# An I/O Separation Model and its Applications to On-Demand I/O on Commodity Platforms

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Department of Engineering

Miao Yu

B.S., Software Engineering, Shanghai Jiao Tong University
M.S., Computer Software and Theory, Shanghai Jiao Tong University

Carnegie Mellon University
Pittsburgh, PA

December 2019

## Acknowledgements

I am grateful to my advisor, Professor Virgil Gligor. He introduced me to the area of on-demand trusted I/O, and has guided me in all my research through my entire Ph.D. His rich experience and dedication in the area of security inspires me to think deeply and broadly. With such great support, I enjoy my research experience, and always hold the belief that I am working on a great problem of which other people desire to see the result. Virgil also offers me a lot of help outside academics. I especially thank Virgil and Carnegie Mellon University for helping me solve my visa issue in 2014, because otherwise I may have had to terminate my PhD studies at that time.

I would also like to thank my thesis committee: Prof. Limin Jia, Prof. Greg Ganger, and Dr. Jonathan M. McCune. They provide me with invaluable feedback, insightful comments and advice on my dissertation, covered researches and future work. I would especially like to express my thanks to Prof. Limin Jia for teaching me about formal verification of models, and co-advising me in my research of I/O separation models. This experience will be invaluable to me in my future projects. I would also like to thank Prof. Adrian Perrig, who is in my prospectus committee but unfortunately cannot join my thesis committee. His discussions and comments also helped me clarify my thesis, and my roads to related research.

I would also like to thank my co-authors, colleagues and friends: Zongwei Zhou, Yueqiang Cheng, Bryan Parno, Vyas Sekar, Maverick Woo, Jun Zhao, Saurabh Shintre, Min Suk Kang, Chen Chen, Youzhi Bao, Yuan Tian, Tianlong Yu, Janos Szurdi, Mahmood Sharif, Soo-Jin Moon, Steve Matsumoto, Mengwen He, Weijing Shi, Yifei Yuan, Zaoxing Liu, Yuchen Yin, Weizhao Tang, Tian Li, Yoshiharu Imamoto, Gihyuk Ko, Michael Stroucken, Xiaolin Zang, and everyone else in CyLab. I am indebted to you for all of the great ideas, discussions and support. And also thanks to all the help from open-source communities: the members of intel-gfx, radeon, nouveau IRC channels.

I would like to express my gratitude to my family. They always support me unconditionally, and alleviate my worries when I meet difficulties. I feel lucky to have grown up in my family, where my parents quietly shaped my characteristics. At last, thanks to my lovely wife, Huilin, and my two kids, Bowen and Steven, for the joy and happiness they bring to me.

The research in this thesis was supported in part by CMU CyLab under the National Science Foundation grant CCF-0424422 to the Berkeley TRUST STC, and the Department of the Navy, Office of Naval Research, under grant N00014-17-1-2892. The views and conclusions contained in this paper are solely those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

## Abstract

A key goal of security architectures is to separate I/O transfers of security-sensitive applications from untrusted commodity OSes and other applications, with high assurance. These architectures provide I/O kernels that assure the confidentiality and authenticity of the transmitted I/O data owned by a security-sensitive application, even when commodity OSes and other applications are compromised. These kernels help eliminate security-sensitive application exposure to drivers they do not need. This is a major security advantage because drivers contribute over half of code size in commodity OS kernels.

However, existing I/O kernels can only enforce I/O separation on *limited* hardware configurations of commodity platforms, if they rely on existing I/O hardware mediation components such as IOMMU, or ignore I/O operations that could be misused to break I/O separation. Commodity I/O hardware designs focus primarily on increasing performance and device connectivity, but often fail to separate I/O transfers of isolated OS and applications code. Even when using the best I/O hardware, commodity systems sometimes trade off isolation assurance for increased performance. Remarkably, to breach I/O separation, device firmware need *not* be malicious, though it is allowed to be so. Instead, any malicious driver can manipulate its device to breach I/O separation. To prevent such vulnerabilities in kernel designs with high assurance, a formal I/O separation model is necessary.

This dissertation defines an I/O separation model for general commodity platforms and proves its soundness. The model defines a precise separation policy based on complete mediation of I/O transfers despite frequent lack of commodity hardware to support it. Thus it can be applied to the I/O designs of *all* commodity platforms, compared to previous kernels that work on limited hardware configurations. Furthermore, this dissertation applies the model to the latest I/O kernels that offer on-demand I/O separation. These kernels allow security-sensitive applications to relinquish and release their devices to and from untrusted commodity OSes on-demand. The dissertation shows how to apply the I/O separation model to one carefully but informally designed on-demand I/O kernel, the Wimpy Kernel, and illustrates how the model enables the discovery of formerly unknown vulnerabilities. The dissertation also shows how to remove these vulnerabilities and obtain a model-based I/O design – an unavailable feature of commodity systems. In addition, the dissertation presents a novel GPU Separation Kernel to allow isolated applications to share display with untrusted OS and other applications, and informally analyzes it against the same vulnerabilities.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

An I/O separation primitive allows Security-Sensitive Applications (SecApps) [1] to communicate with their devices, and to protect the confidentiality and authenticity of their I/O data from other applications. With separate I/O, SecApps can interact securely with users via human-oriented devices such as keyboards, displays, joysticks, etc. For example, users can type passwords into SecApps without worrying about keystrokes being deliberately accessed by other applications. Another example is users can rely on the information displayed by SecApps without worrying about undetectable screen "scrapping" or "painting" by malicious software on commodity systems. These secure I/O channels (also known as "trusted paths" in the Orange book [8]) address the "ultimate insult" towards the end-to-end argument in system designs [26]; i.e., users must have secure I/O channels to protected applications in their own end system in order to securely access a remote system.

In addition to communicating information with users, SecApps can also employ separate I/O to securely control cyber-physical systems and mission critical systems that connect to them. After I/O separation is established between SecApps and their sensors or actuators, no other application can inject commands into these devices. Without I/O separation, attacks to these systems will continue to emerge, such as attacks on power plants [92], the power grid [93], nuclear centrifuges [119], petrochemical plants [81], and unmanned drones [123].

Unfortunately, commodity OSes, which are designed to manage and multiplex hardware resources to applications, can only provide separate I/O to applications with *low assurance* due to large and complex code bases in OS kernels. Commodity OSes implement rich I/O functions in monolithic OS kernels, which now comprise tens of millions lines of code [52]. Given this, it is unsurprising that attackers constantly reveal OS kernels' vulnerabilities [29], eliminating all kernels' security guarantees. Even worse, OS kernels

[1] Also known as Piece of Application Logic (PAL)

continue to bloat [52], due to supporting new hardware and new I/O functions over time. Because of this, the best one can hope for is to protect SecApps' I/O to their devices *without* trusting the OS.

Thus, a key goal of security architectures is to separate I/O transfers of SecApps from untrusted commodity OSes and other applications, with high assurance. To accomplish this goal without enlarging the underlying trusted code base (e.g., security kernels, micro-kernels, micro-hypervisors), existing designs rely on dedicated I/O kernels [35, 121, 125, 21, 134, 128]. They de-privilege device drivers, export them to isolated applications to separate them from each other, and authorize them to access only their own devices. The latest I/O kernels [134, 128] target *on-demand I/O separation*, which enables SecApps to relinquish and release their devices to and from untrusted commodity OSes on-demand. These kernels for on-demand I/O separation allow unmodified commodity OSes to have unrestrained access to all hardware resources, and hence have the advantage of being compatible with unmodified commodity software. All these kernels help eliminate isolated-application exposure to unneeded drivers. This is a major advantage since driver code comprises over half of modern operating system kernel code and accounts for many flaws [25].

## 1.1 Insufficiencies of Hardware Support for I/O Separation

Existing I/O kernels can only enforce I/O separation on *limited* hardware configurations of commodity platforms, as long as they rely on existing I/O hardware mediation support (e.g., IOMMU, PCIe Access Control Service (ACS)) on commodity platforms, or ignore I/O accesses (e.g., I/O multicast/broadcast) that could be misused to break I/O separation. Commodity I/O hardware designs focus primarily on novel features to improve performance [88, 85], increase connectivity [7, 55], and decrease cost [56], but often fail to support the separation of I/O transfers of isolated applications. Concretely, malicious drivers can misuse commodity I/O hardware to break I/O separation despite their isolation in separate address spaces. Five examples illustrate this:

1. Whenever devices are connected by PCI buses, a malicious driver can perform device peer-to-peer (P2P) communications and enable direct data transfers between I/O devices. This enables the malicious driver to avoid P2P transfer *mediation* in I/O configurations where devices are connected by PCIe buses and use Access Control Services (ACS) [99, 132], see Example 1 in Section 2.1.

2. To decrease the number of separate I/O requests, an I/O bus controller interface allows multicast/broadcast transfers by the controller's driver [88, 85] to different devices attached to the bus controller. Thus, a malicious controller's driver can easily breach the separation for I/O devices of different isolated applications; see Example 2 in Section 2.1.

3. A device can issue peer-to-peer transfers to configure another device, such that the latter accesses isolated I/O objects on behalf of the first device. Thus, a malcious driver can issue surrogate/indirect transfers to isolated I/O objects, by directly accessing its devices; see Example 3 in Section 2.1.

4. To decrease the cost of selective DMA-access mediation[2], some commodity IOMMUs [56, 98] mediate accesses at the granularity of PCI *bus controllers* (e.g., PCIe-to-PCI bridges) instead of that of individual devices. This allows isolated drivers to access other drivers' memory data via devices that are connected to a PCI bus and perform DMA transfers [98]; see Example 4 in Section 2.1.

5. To improve the performance of selective DMA-access mediation, IOMMU caches translation results along with access permissions in the IOMMU's translation look aside buffer (IOTLBs). When IOMMU memory mappings are managed dynamically, IOTLB invalidation suffers significant performance penalty because of locking and updating IOTLBs [90, 74]. *Deferring* IOTLB updates reduces this penalty, but enables a malicious isolated driver to manipulate its DMA device, reuse stale IOTLB entries, and access the memory of another isolated driver; see Example 5 in Section 2.1.

Note that a single isolated but compromised driver can breach I/O-transfer separation of other isolated applications by exploiting inadequate I/O hardware mediation of commodity hardware. *No* device firmware needs to be malicious; e.g., in contrast with other work [15, 130, 94, 73], device firmware need *not* contain any malware. (However, adversaries are allowed to corrupt device firmware; see Section 2.2.) Chapter 2 illustrates that *any* malicious driver which is *authorized* to access the I/O data and configuration registers of its non-malicious device can launch an I/O separation attack.

## 1.2 Lack of Formal I/O Separation Model

Central to enabling high-assurance I/O separation primitives is formal I/O separation models. Lacking such models and their interpreted design of I/O separation for isolated applications, current I/O separation kernels cannot match the high assurance of their underlying trusted code base (e.g., micro-kernels, micro-hypervisors [60, 113, 114, 41] and separation kernels [48, 109, 97]). Unless addressed, this imbalance leads to vulnerabilities of isolated applications that perform I/O operations[3]. To be useful, such a formal model must be proved to be sound and must satisfy the following four practical goals.

1. *Hardware-Agnostic Meditation*. It must be possible to instantiate the formal model on *any* commodity I/O hardware configuration regardless of its ability, or lack thereof, to mediate I/O transfers. This implies that it must be possible to instantiate the formal I/O transfer-meditation rules on *any* commodity hardware

[2] Performance of *no-mediation* is higher than *selective mediation* of TCP transfers by about 400 - 1,800% on Intel systems with 1 - 16 cores [74].

[3] Similar vulnerabilities have already been witnessed when isolated VM applications rely on isolated drivers [27] to assure I/O separation. A single driver could still exploit I/O hardware to bypass mediation [132].

platform. Why is this important? The large variety of commodity processors that support application isolation (e.g., via trusted execution environments [91, 84, 13, 112, 69, 51], partitions [48, 109, 97], pieces of application logic [77, 113, 114], and enclaves [30, 41]) will continue to be interconnected to I/O hardware that fails to adequately mediate transfer separation in future applications, for both high performance and low cost reasons; e.g., in CPS [120] and vehicular [120, 63] computing. Some existing application isolation approaches support outdated 32-bit processors [60, 113], in order to support related hardware platforms used in specialized computing environments [39]. In all these systems, our formal model helps identify beneficial hardware-configuration restrictions, and compensates for hardware inadequacy on unrestricted configurations, to protect I/O separation.

The benefits of configuration restrictions are illustrated by SUD [21], which limits transfer separation to the non-selective mediation IOMMUs for PCIe-to-PCI bridges, but not necessarily for individual devices; see Section 2.1, *Example 4*. Nexus RVM [121] uses an commodity-incompatible kernel and must exclude configurations that perform unmediated peer-to-peer (P2P) transfers or allow broadcasts on a bus controller; see Chapter 7. Wimpy Kernel [134] must avoid configurations where devices of different applications share the same PCIe-to-PCI bridge or use unmediated P2P transfers; see Section 5.2.

Although hardware-agnostic mediation can compensate for I/O hardware inadequacy, it also clearly illustrates the fundamental benefits of I/O hardware support for *complete mediation* over inadequate hardware, beyond obvious performance considerations. When I/O hardware for complete mediation is available, mediation enforcement is deferred to the time when transfers are initiated (i.e., late mediation), and *all* legitimate I/O transfers and hardware configurations are allowed. In contrast, when required to compensate for inadequate I/O hardware on unrestricted configurations, complete mediation has to verify potential I/O transfers that can be initiated in the future; i.e., early mediation. Naturally, sound early mediation may conservatively deny some future legitimate I/O transfers or restrict hardware configurations whenever they appear to violate I/O separation; see discussion in Section 3.1.4.

2. *Explicit Separation Policy and Mediation Rules*. Even if a hardware-agnostic formal I/O mediation is provided, a formal separation policy that defines *what must be verified* during mediation does *not* exist. In particular, the policy must include *new* mediation rules that allow unprivileged subjects actions on objects that are disallowed by commodity OSes. For example, in contrast with these OSes where all drivers are privileged, the policy must also allow unprivileged drivers exported to unprivileged applications to access devices directly on demand. Three examples illustrate the need for new rules.

First, unprivileged drivers can modify I/O transfer descriptors, which enable further transfers by other devices (e.g., *indirect* (*surrogate*) transfers) and break application isolation. New mediation rules must verify *both* the presence of the *write* permission to modify a transfer descriptor *and* whether all further actions enabled by modified transfer descriptors (e.g., *read/write* to other I/O devices) are authorized.

Second, when unprivileged drivers or devices are reused by *on-demand* activation from an isolated application to another, hardware mediation is insufficient to guarantee separation. New mediation rules must prevent reuse of multiple I/O objects of drivers and devices independent of the reuse prevented by OS object allocation.

Third, unprivileged (e.g., SMBus) drivers can execute a single broadcast instruction that modifies I/O devices belonging to different isolated applications. New mediation rules must determine when such instruction executions are authorized; e.g., all these devices belong to a single isolated application.

*3. Compatibility*. Traditional OS access control models enforce mandatory rules (e.g., based on security and integrity levels) that often break existing commodity applications code [102, 47, 18, 101]. In contrast, to be useful, the instantiation of the I/O separation model in an I/O kernel must maintain object-code compatibility with existing applications. For example, the model instantiation must allow on-demand I/O of finer granularity subjects (i.e., drivers in isolated applications) to devices, without preventing the coarser granularity subjects (e.g., users or virtual machines) from accessing their (e.g., statically allocated) devices in commodity systems.

*4. Generality*. The I/O separation model must apply to other high-assurance kernels, such as separation kernels [96, 97] – not just to micro-kernels and micro-hypervisors – and to low-assurance OS kernels; e.g., Linux [90, 74].

## 1.3 Thesis Overview

**Thesis Statement:** *Privileged I/O kernels must be designed to provide I/O transfer separation to security sensitive applications on commodity platforms for a wide variety of I/O hardware, with high assurance.*

Concretely, this dissertation first defines a formal *I/O separation model* satisfying the four properties above. Then, the dissertation interprets the model to an I/O kernel design offering on-demand I/O separation, and instantiates this concrete model interpretation to a sound design of real I/O kernel (the Wimpy Kernel [134]), which *time-multiplexes* USB peripheral devices between SecApps and untrustworthy commodity software, and provides I/O separation to SecApps on-demand. At last, the dissertation presents a new I/O kernel, the GPU Separation Kernel, which allows SecApps and untrusted software to *concurrently* access GPU and display. The GPU Separation Kernel and the Wimpy Kernel secure SecApps' communications with *any* devices demanded by the SecApps, whether they are *exclusively* owned by SecApps, or *shared* with an untrusted OS or other applications.

This thesis makes the following contributions:

- This dissertation reviews a variety of transfer-mediation capabilities of commodity hardware platforms (e.g., Intel, AMD, ARM, IBM I/O in Figure 2.1), their security vulnerabilities (in Section 2.1), and threats they pose to I/O separation (in Section 2.2).
- This dissertation defines an I/O separation model and proves its soundness (Chapter 3). The model defines a precise separation policy, and is hardware agnostic. Its application to real systems does not break object-code compatibility with existing applications, and is general; i.e., it can be applied to a variety of both high-assurance and low-assurance kernels; see Chapter 4 and Section 8.1. The model is formally specified and verified in Dafny [66].
- This dissertation shows how to formaly interpret the abstract model and obtain a concrete model for *I/O kernel* design that supports on-demand I/O separation (Chapter 4).
- This dissertation applies the concrete model to two I/O kernels that provide on-demand I/O separation: the *Wimpy Kernel* [134] and the *GPU Separation Kernel* [128]. First, this dissertation *formally* instantiates the concrete model to a sound Wimpy Kernel design (Chapter 5), which supports SecApps exclusively accessing their own peripheral devices. The dissertation illustrates how the concrete model enables the discovery of vulnerabilities in the original Wimpy Kernel design, and how to remove those vulnerabilities. Second, this dissertation presents an informal GPU Separation Kernel (Chapter 6), which supports SecApps sharing display (and hence GPU) with untrusted OS and other applications. The dissertation informally instantiates the concrete model to the design of this I/O kernel, and especially analyzes the kernel design against the same vulnerabilities.

**Note**: This dissertation does not imply any original Wimpy Kernel design contributions. Instead, this thesis produces a sound design of Wimpy Kernel for on-demand I/O separation. The full Dafny specifications of the I/O separation model, the concrete model, and the sound Wimpy Kernel design can be downloaded from *https://github.com/superymk/iosep_proof*.

# Chapter 2

# I/O Vulnerabilities and Threats

Commodity I/O hardware designs focus primarily on novel features to improve performance [88, 85], increase connectivity [7, 55], and decrease cost [56], but often fail to support the separation of I/O transfers of isolated applications. Even the best hardware for separating high throughput memory accesses of I/O transfers (e.g., via concurrent IOMMU updates) exposes interfaces (e.g., IOTLBs invalidation) that encourage trading off isolation assurance for substantially increased performance [90, 74]. Figure 2.1 illustrates a variety of commodity I/O interconnection hardware ranging from devices that do not mediate any I/O transfers, to those that mediate transfers non-selectively, and finally the best available devices that can provide selective mediation.

## 2.1 Common Vulnerability Examples

Whenever devices are connected by PCI buses, a malicious driver can manipulate a device to perform unmediated *direct* and *indirect* peer-to-peer (P2P) communications with other devices. This allows a malicious driver to avoid P2P transfer *mediation* in I/O configurations where devices are connected by PCIe

| | Intel / AMD | ARM | IBM | External buses |
|---|---|---|---|---|
| *No mediation* | • **PCI, no ACS**<br>• **SMBus** | • **AHB (PCI-like), no ACS**<br>– early version: **ASB** | • **PCI, no ACS** | • **CAN**<br>• **I2C** |
| *Non-selective mediation* | • **PCIe-to-PCI bridge** | • **AXI-to-AHB bridge** | • **PCIe-to-PCI bridge** | • **Firewire (w/ OHCI)** |
| *Selective mediation* | • **IOMMU**<br>• **PCIe ACS** | • **SMMU**<br>• **TZ Normal vs. Secure** | • **IOMMU / CAPI**<br>• **PCIe ACS** | |

Figure 2.1: Examples of mediation levels in commodity I/O hardware.

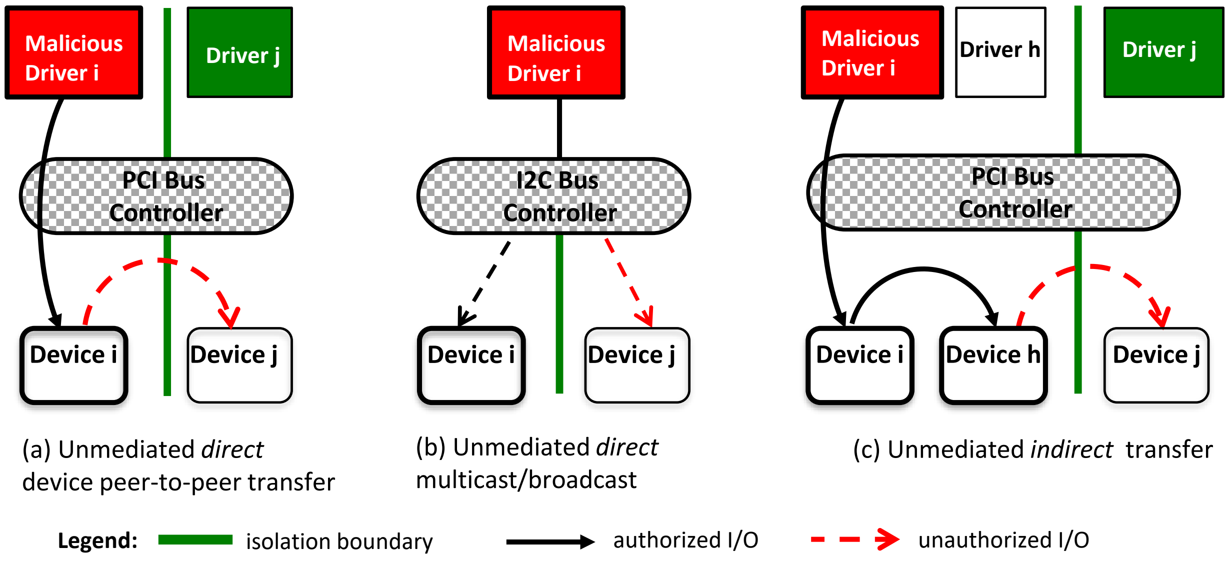


Figure 2.2: Separation vulnerabilities of no-mediation I/O hardware.

buses and use Access Control Services (ACS) [99, 132]; see Examples 1 and 3 below. Furthermore, malicious drivers can issue I/O multicast/broadcast transfers unmediated by bus controllers to isolated I2C devices, see Example 2. These and all other examples use x86/x64 and ARM configurations, which also apply to other platforms with small differences.

*Example 1. Lack of direct device peer-to-peer transfer mediation*. Common I/O transfer mechanisms enable a device to access another device without providing its identity in the I/O transfers. For example, PCI buses, the System Management Buses, CAN buses, and ARM Advanced High-performance Bus (AHB) buses do not require I/O requests to include device identities for senders' device authentication [103, 110, 63, 17], and yet allow device peer-to-peer (P2P) communication. When connected to these buses, sender devices always have the same privilege as the bus controllers in accessing configuration and data registers of recipient devices. This enables unauthorized transfers; i.e., both a *write* over the recipient device data registers or a *read* from the recipient device. As shown in Figure 2.2(a), a malicious (red) driver $i$ can configure its device $i$ to perform device P2P communications, and successfully access another device $j$ that belongs to another isolated application, without any mediation. In this way, the malicious driver breaks I/O isolation.

*Example 2. Lack of direct I/O multicast/broadcast mediation*. I/O multicasts/broadcasts allow a driver to access multiple devices with one atomic transfer. For example, a driver can configure an I2C bus controller to issue write transfers to the general call address, in order to broadcast writes to all I2C devices connected to the bus controller. Bus controllers and devices do not mediate these transfers by drivers, because of unabling to distinguish different drivers. As a result, malicious drivers can access their devices and isolated devices in one transfer, and hence breaks I/O isolation, as shown in Figure 2.2(b).

*Example 3. Lack of indirect (surrogate) transfer mediation*. A device can execute a P2P transfer that configures another device such that the latter *reads/writes* an isolated I/O object on behalf of the first device. For example, a PCI device $i$ can configure another PCI device $h$ via P2P transfers and enable device $h$ to *read/write* I/O objects of otherwise isolated device $j$ on $i$'s behalf. Here the device $i$ can embed the identity of an I/O object of device $j$ into its maliciously configured device $h$, instead of issuing *reads/writes* to the I/O object directly. Thus, the direct transfers of device $i$ to device $h$ do not break I/O separation. Instead, the transfers of device $h$ to device $j$ do. Figure 2.2(c) shows how the malicious driver of device $i$ configures non-malicious device $h$ to unwittingly *read* or *write* an object of device $j$ across a separation boundary. If device $h$ *writes* over a descriptor $TD_j$ of device $j$, then a device $j$'s transfer via the modified $TD_j$ may breach another application-isolation boundary.

*Example 4. Non-selective mediation*. Some [1] commodity IOMMUs [56, 98] mediate accesses at the granularity of PCI *bus controllers* via PCIe-to-PCI bridges instead of individual devices; i.e., *non-selective* mediation. Also, when issuing requests on behalf of their individual DMA devices, USB host controllers [55] use their own identities. In both cases, the IOMMU between the I/O bus controller and main memory regards all DMA device transfers as originating from the I/O bus controller, and hence cannot mediate transfers selectively at the granularity of individual devices. This allows isolated drivers to access each other's memory data via devices that are connected to a PCI bus and perform DMA transfers [98].

As shown in Figure 2.3, non-malicious device $i$ operating under the control of a malicious driver $i$ can *read* or *write* a DMA memory region of another device $j$ across an isolation boundary, even though the IOMMU is correctly configured and isolated. For selective mediation, devices of different isolated applications must be connected to different PCIe-to-PCI bridges yielding restricted hardware configurations [21]. ARM TrustZone requires stronger configuration restrictions because it identifies device transfers at the granularity of the *world*; i.e., normal versus secure world.

*Example 5. Exploiting a performance-I/O isolation trade-off*. Selective mediation mechanisms based on PCIe ACS and IOMMU (see Figure 2.1) reduce performance when memory is dynamically mapped. This requires locking and updating the IOMMU's translation look aside buffer (IOTLBs), which impose a large performance penalty [90, 74]. *Deferring* IOTLB updates substantially reduces this penalty, but enables a malicious isolated driver to manipulate its DMA device, reuse stale IOTLB entries, and access

[1]This is in corresponding to IOMMU specifications [56].

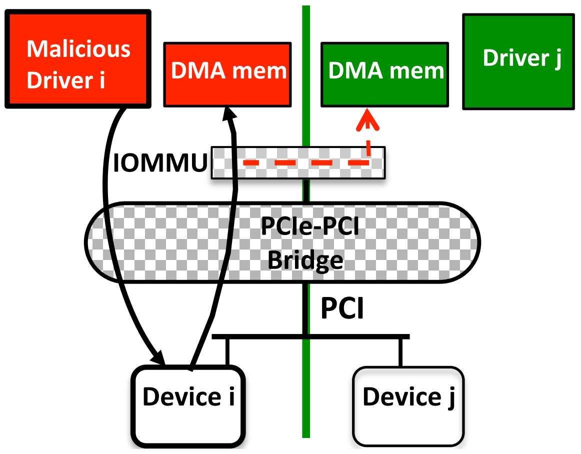


Figure 2.3: Separation vulnerability of non-selective mediation.

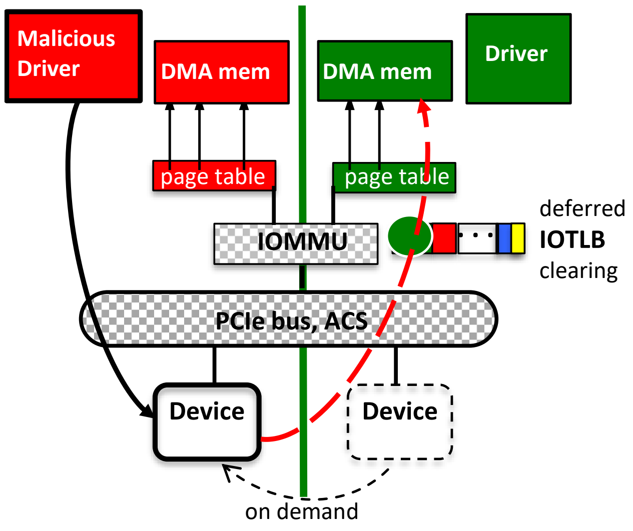


Figure 2.4: Separation vulnerability of a performance-isolation trade-off.

the memory of another isolated driver. Figure 2.4 illustrates negative consequence of this performance-isolation assurance trade-off; i.e., an attack that breaches the address spaces of isolated applications by exploiting the deferred clearing of IOMMU's IOTLB entries up to 10 *ms* [90, 74]. During this delay, the DMA device is allocated to a malicious driver, which is isolated in another application, on demand. The malicious driver legitimately instructs the DMA device to issue a transfer whose target virtual address is translated using the undeleted IOTLB entry (green circle), breaching red-green address-space isolation.

While the recently proposed countermeasure to this attack [74] works well for low-assurance (e.g., Linux) kernels, it would add 15% or more complex code for the management of the single *shared* buffer-pool by a high-assurance micro-hypervisor [77, 113, 114] or micro-kernel [60, 41] and significant performance degradation due to switches to/from the micro-hypervisor/kernel. In contrast, the interpretation of our model in I/O kernels of such systems (Section 4.3) shows that *separate* (non-shared) drivers and their local buffers can be de-privileged and safely exported to isolated applications. This substantially simplifies both micro-hypervisor and I/O kernel verification and naturally avoids the performance penalties of concurrent access to a single shared buffer pool and micro-hypervisor/ kernel switches (Section 9).

Similar examples of isolation breaches as those above can be caused by green applications; e.g., breaches of other isolated green applications, and of isolated red applications.

## 2.2 Threat Model

Section 2.1 reviewed several types of inadequate mediation for separation of I/O transfers, and showed how an adversary can breach *confidentiality* and *integrity* of sensitive I/O data across isolated-application boundaries. The emerging attack pattern is simple: either the adversary compromises a driver or provides their own isolated application containing a deliberately compromised driver. To launch an attack the adversary's malicious driver can misconfigure devices to set up unauthorized I/O transfers either directly or indirectly. These attacks appear in all commodity OSes – even in those that benefit from the use of formally verified micro-hypervisors and micro-kernels that provably isolate drivers.

Note that the adversary need *not* corrupt device firmware to launch the I/O separation attacks, but is allowed to do so; e.g., surreptitiously modify the device controller's firmware by re-flashing [130, 94, 73] or by supply-chain compromise [15, 14, 80, 28]. Techniques to *provably verify* the correct device firmware and register content after re-flashing are known [45], and our model allows such verification before device activation (Section 3.2). However, the latency of controller verification (e.g., microseconds [67, 68, 62, 45]) suggests that device activation should be infrequent, if done on demand, or even static. It is also required that legitimate backdoors, such as hardware debugging interfaces for privileged access to I/O devices, be disabled before system operation. Furthermore, as is often the case, device hardware (not firmware) is assumed to be non-malicious. As a result of trusted hardware and verified firmware, devices function as specified. For example, PCIe devices must issue transfers with translated addresses after the addresses are translated by IOMMU/PCIe Address Translation Services (ATS), or otherwise the transfers claimed being translated bypass mediation by IOMMUs [73]. Finally, denial-of-service and covert-channel attacks in I/O transfers are irrelevant to the separation model.

# Chapter 3

# I/O Separation Model

This chapter provides an overview of I/O subjects, objects and accesses, mediation of subject accesses to objects, and properties of the I/O separation model. Then, this chapter describes the state-transition model, which includes the notions of the state, state invariants, operations, transition constraints, and security theorems. The theorems show that the access checks of the model operations yield sound security properties. The model has been formally specified and verified in Dafny.

## 3.1 Model Overview

### 3.1.1 Subjects, Objects, and I/O Accesses

The set of *subjects* comprises *Drivers*, which can be viewed as arbitrary programs that read/write devices, and *Devices*. *Devices* include common peripherals as well as I/O bus controllers routing I/O transfers between devices and CPUs/memory. These components can access each other via different types of I/O communications; e.g., CPUs *read* and *write* devices via Memory-Mapped I/O (MMIO) and Port I/O (PIO)[1], devices *read* and *write* memory via Direct Memory Access (DMA), and request CPUs' attention via interrupts. All subjects have unique (e.g., non-reusable) names.

Device drivers run on CPUs and interact with devices via *I/O objects*, which comprise *Data Objects (DOs)* and *Descriptor Objects* both of which can be read/written by drivers and devices. All I/O objects have unique (e.g., non-reusable) names. Data objects store devices' input and output data; e.g., device status registers, data buffers in drivers and devices. Descriptor objects define device functions, such as transfers to be issued, power management, performance management, and are further partitioned into function and transfer descriptors.

[1]ARM platforms support MMIO only [31].

*Function descriptors* (FDs) define configurations of device functions other than transfers; e.g., frame buffer format registers in GPUs define the data format of pixels, and power control registers are used to configure power of certain GPU components and/or display components. *Transfer descriptors* (TDs) define direct I/O transfers to be issued by devices. Every TD contains a list of entries, each comprising a pointer to an I/O object, requested access modes, and new values to write. For example, the frame-buffer base registers of GPUs are TDs, and comprise read transfers to be issued to data objects; i.e., to frame buffers. Also, the Interface Data Structures in drivers of USB host controllers are more complex TDs, which not only include descriptions of transfers to data objects, but also include *read* transfers to other TDs, which specify the next interface data structures to be processed by USB host controllers.

Each device owns a *Hardcoded TD*, which defines local transfers issued to the device's own *internal objects*, based on the device's hardware/firmware implementation. For example, a GPU's hardcoded TD defines *read* transfers to TDs of frame-buffer base registers to enable the GPU poll their configurations. Hardcoded TDs are *read* only by their containing device and do not define *read* transfers to themselves. However, they define either *read* or *write* transfers to other TDs. This enables a device to perform arbitrary transfers via these TDs. Finally, hardcoded TDs cannot be *written* as they are *immutable* between (trusted) re-flashing operations of device firmware.

Each internal I/O object is owned by exactly one subject, as opposed to the *external* I/O objects, which are not owned by any subject. In practice, I/O object ownership is defined by I/O kernels and isolated applications at initialization time. For example, I/O kernels may isolate external I/O objects from drivers to mediate driver access to these objects; e.g., interface data structures for protected USB devices are stored in an I/O kernel, not in their drivers. Also, external I/O objects may be owned by applications, not only by kernel drivers, to eliminate data copy overhead between applications and these drivers.

Drivers and devices perform different types of transfers to I/O objects. Drivers issue arbitrary transfers to any I/O object except to devices' hardcoded TDs. Devices *read* only their own hardcoded TDs but not those of other devices. A device *can issue* a transfer to an I/O object if and only if the transfer is defined by the device's hardcoded TD or by a TD that can be *read* by the device via its hardcoded TD. For example, the device of Figure 3.1(a) can issue a transfer to the I/O object 2 because (1) the device can *read* its hardcoded TD, (2) hardcoded TD defines a *read* (R) transfer to TD1 and thus the device can read TD1, and (3) TD1 defines a *read/write* (R/W) transfer to the I/O object 2. However, the device cannot issue transfers to the I/O object 3 since they are not defined by *any* TD that can be *read* by the device. The device can do so, only after a driver modifies TD1 directly, or indirectly, as shown in Figure 3.1(b).

Modification of TDs enables *indirect* (i.e., *surrogate*) transfers by devices. For example, a driver can configure a device to perform a direct *write* over a TD that can be *read* by another device, which in turn

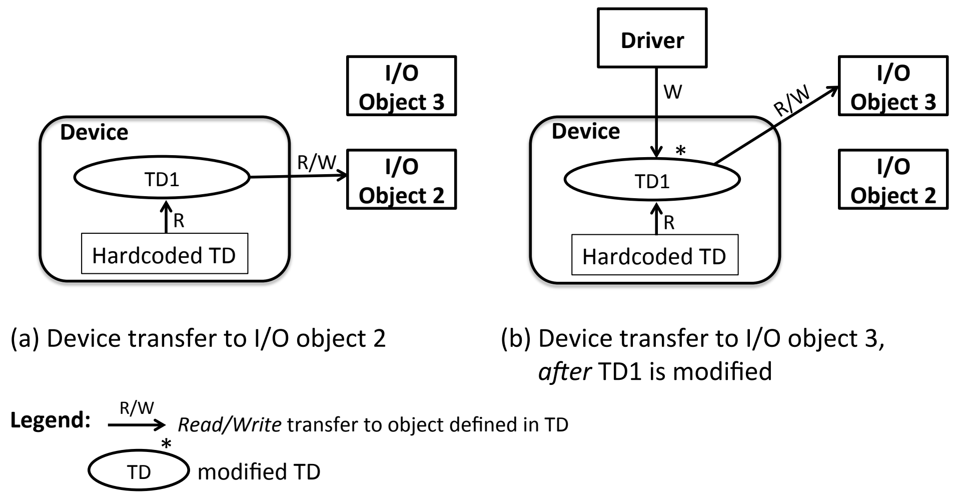

Device
TD1
R/W
I/O Object 3
I/O Object 2
R
Hardcoded TD
Driver
W
*
(a) Device transfer to I/O object 2
(b) Device transfer to I/O object 3, *after* TD1 is modified
Legend:
*Read/Write* transfer to object defined in TD
TD
modified TD

Figure 3.1: Device transfers controlled by TDs.

performs a direct I/O transfer to a target object. In Figure 2.2(c) of Section 2.1, malicious driver $i$ configures device $i$ to *write* the $\text{TD}_h$ of device $h$ to perform a direct *write* to target device $j$. Here, device $i$ performs an indirect I/O transfer to device $j$ via device $h$, which acts as a surrogate for device $i$ and accesses I/O objects of device $j$ on behalf of device $i$. *Surrogate/indirect transfers are mediated by a new transitive-closure mechanism described below.*

### 3.1.2 Transfer Mediation by I/O kernels

All devices' transfers are mediated by I/O kernels which verify *direct* and *indirect* transfers to single I/O objects, such as single TDs, or to multiple I/O objects when an instruction initiates an atomic multicast transfer[2] to multiple devices.

I/O separation kernels can use as much hardware support for mediation as available in a system. For example, in Figure 3.1(b), an I/O kernel is assumed to have mediated the driver's direct or indirect *write* access to TD1. This means that an I/O kernel knows what values can be written to TDs directly and indirectly and checks those values; i.e., by computing the *transitive closure* of possible values of TDs caused by a driver's *write*, as described below. Then the kernel denies a driver's *write* attempt, if any of the values to be written could enable transfers that break the I/O separation policy; see Section 3.1.3 below.

**Transitive closure**. Intuitively, a device is *active* if it is not in an idle state and thus can perform I/O transfers. Also, a TD is *active* if it defines a transfer by an active device. (In contrast, inactive TDs are those corresponding to devices that have not been activated yet; i.e., are idle and thus cannot perform any

[2] Emulating an atomic (i.e., all-or-none) I/O transfer to multiple devices with a chain of separate transfers would require additional hardware constraints to ensure no I/O object is modified on a denied transfer in the middle of the chain. Otherwise, all allowed modifications prior to transfer denial would have to be reversed to preserve atomicity.

transfer). Let a *TD_state* comprise the values of all active TDs at a point in time; i.e., the contents of all TDs in use at that point in time. The *transitive closure* of a *TD_state* is the set of all possible active TDs contents created by any direct and indirect TD *writes* that *could be* issued by active devices. Naturally, the transitive closure does not contain inactive-TD values, since they cannot be accessed by subjects and hence are left unchanged by I/O transfers.

**An example**. Indirect (surrogate) *writes* by devices may target a set of TDs that may *not* be directly referenced by any TDs in a given state. Absence of direct references does *not* imply that mediation of the corresponding I/O transfers need not be performed. For example, in Figure 2.2(c) of Section 2.1, device *i*'s indirect *write* on device *j* targets $\mathrm{TD}_j$, and yet the initial system state does not contain any *direct write* reference to $\mathrm{TD}_j$, which would require mediation. To ensure that mediation is performed for all TD targets of indirect *writes*, and hence for all I/O transfers they cause, a transitive-closure computation converts indirect *writes* to TDs into sequences of *direct transfers* to TDs by devices. Thus, it discovers all *TD_states* whereby devices *could issue* direct TD *writes*. For example, Figure 3.2 illustrates the initial *TD_state* where device *i* can *read* $\mathrm{TD}_i$, which was already written by driver *i*, device *h* can *read* $\mathrm{TD}_h$, and device *j* can *read* its own $\mathrm{TD}_j$. However, no direct *write* reference to $\mathrm{TD}_j$ exists in this state. The transitive-closure of this initial *TD_state* converts device *i*'s indirect *write* to $\mathrm{TD}_j$ into the following sequence of four *direct transfers* to TDs: device *i reads* $\mathrm{TD}_i$ and *writes* $\mathrm{TD}_h$, whereas device *h reads* $\mathrm{TD}_h$ and *writes* $\mathrm{TD}_j$[3]. Thus, the transitive closure of the initial *TD_state* includes three states of active TDs: the first is the initial state itself, the second is obtained when device *i* performs the direct *write* to $\mathrm{TD}_h$, and the third is obtained when device *h* performs the direct *write* to $\mathrm{TD}_j$.

**Transitive-closure computation**. The transitive closure computation of a given *TD_state* comprises two functions. The first function discovers all direct TD *writes* that can be issued by a device. To do this, the function uses a *breadth-first-search* (BFS) algorithm, which starts from the device's hardcoded TD, and constructs and traverses the graph of TDs that can be *read* by the device. After discovering all TDs that can be *read* by the device, this function iterates over all their entries and outputs all TD *writes* enabled by these entries. Hence, the function outputs all direct TD *writes* that *could be* issued by the device in the given *TD_state*.

The second function utilizes this reachability relationship between states established by the first function, constructs and traverses a graph of *TD_states* with the BFS algorithm, and outputs all potential states that enable I/O transfers. The result of this function always includes the initial *TD_state*.

[3] Implementation of the transitive-closure computation in an I/O kernel must account for the fact that indirect *writes* may not finish at the time of the next TD *write* is issued by a driver since indirect I/O transfers may be interleaved with any direct I/O transfers. Consequently, indirect TD *writes* may not be caused only by the latest direct TD *writes* by drivers but also by unfinished prior indirect *writes* by devices. Also, indirect *writes may not finish* at the time of devices' *reads* of these TDs. Hence, a device may *read* an old transfer's parameters, which may not have been overwritten yet by indirect *writes*.

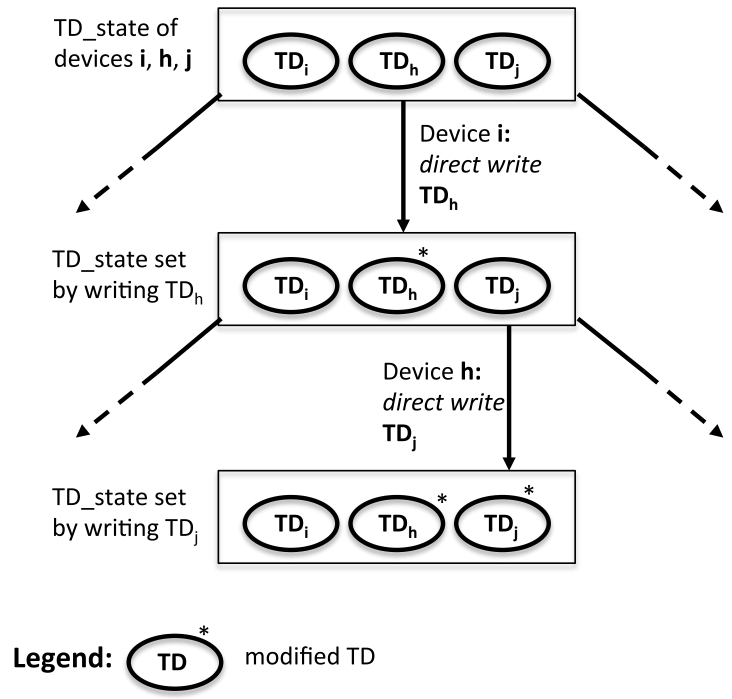

TD_state of devices **i**, **h**, **j**
$TD_i$ $TD_h$ $TD_j$
Device **i:** *direct write* $TD_h$
TD_state set by writing $TD_h$
$TD_i$ $TD_h^*$ $TD_j$
Device **h:** *direct write* $TD_j$
TD_state set by writing $TD_j$
$TD_i$ $TD_h^*$ $TD_j^*$
Legend: TD* modified TD

Figure 3.2: Transitive closure of initial TD_state of Figure 2.2(c), Section 2.1.

Note that the transitive-closure computation always terminates since the set of TDs is finite, and each TD has a finite set of values. Thus, the number of all possible *TD_states* is finite. Also, the computation terminates even when TD graphs are cyclic; i.e., BFS algorithms traverse cyclic graphs.

### 3.1.3 I/O separation policy

Having informally established that an I/O kernel can perform the mediation of all direct and indirect I/O transfers via the transitive closure, the remaining question is what type of I/O separation policy and mediation rules should be enforced.

**Partitions**. One of the simplest I/O policies that formally supports application isolation comprises I/O partitions. Partitions encapsulate the I/O subjects (i.e., drivers and devices) and I/O objects of isolated applications, and prevent cross-partition access, which would violate application isolation. Partitions have unique (e.g., non-reusable) names. A special NULL I/O partition encapsulates all inactive subjects and objects where the subjects are in an idle state and cannot perform any I/O transfer. In contrast, active subjects can issue I/O transfers, and together with active objects can only be in non-NULL I/O partitions.

At any given time, each I/O subject and object belongs to *one and only one* partition. Of course, subjects and objects can move from one partition to another, by deactivation and activation operations. When a subject is (de)activated, all I/O objects owned by it are also (de)activated and moved to(from) the NULL partition. Note that commodity OSes can activate/deactivate various I/O devices on-demand and our policy preserves this feature.

**Properties**. At a high level, the goal of I/O separation is that the non-immutable I/O subject and I/O objects' values of an isolated application in an I/O partition cannot be observed and/or tampered with from other partitions. Specifically, this goal is satisfied by the following two properties:

- (SP1) *No I/O transfer crosses a partition*;
- (SP2) *Only hardcoded TDs can be reused in a new non-NULL partition*.

SP1 prevents drivers and devices from accessing I/O objects in a different I/O partition, while SP2 ensures I/O objects moved between non-NULL partitions do not store any sensitive I/O information belongs to the old partition. These two security properties are sufficient for I/O separation in practice, because each I/O object always belongs to one partition at any time. If an I/O object is in one partition, subjects in other partitions cannot access the object, by property SP1. If not, the object must be moved (i.e., by object deactivation and reactivation) into a new partition, and hence it may be accessed by all subjects of the new partition, but not by the old ones. The new-partition subjects cannot *read* any value written by any subjects of the old partition. To do otherwise, would either violate SP2 or the new partition is the NULL partition. In the later case, subjects cannot issue any transfers, and consequently cannot *read* objects. The reuse of hardcoded TDs between partitions is secure, because hardcoded TDs are immutable between any two trusted device re-flashings.

### 3.1.4 Late versus early (conservative) mediation

Mediation of all I/O transfers based on transitive-closure computation is *hardware agnostic*, and hence the (un)availability of adequate hardware support impacts a system beyond performance considerations. Recall that the transitive closure computation converts all indirect *writes* to and *reads* from TDs into sequences of direct transfers.

*Late mediation*. Late mediation means that runtime execution traces initiate some of these direct transfers, and these can be completely mediated using adequate I/O hardware.

*Early (conservative) mediation*. In contrast, early mediation is performed when only inadequate I/O hardware support is available, and consequently it must be *conservative*. That is, it may restrict I/O hardware configurations and/or may deny *some* legitimate transfers that appear to violate I/O separation in some traces that may not occur in a given runtime execution session. However, conservative mediation is sound: it never misses a transfer that might leave the system in a vulnerable state. This is because the transitive closure outputs all potential *TD_states* whereas *actual* execution traces of I/O operations may yield transfers that reach only a subset of these states. Naturally, if mediation of I/O transfers is performed early, *without* the benefit of examining all and only actual execution traces starting in a *TD_state*, then it has no choice but to deny all potentially insecure I/O transfers which could be issued in that *TD_state*. For instance, in *Example 3* and Figure 2.2(c), the *write* from device $i$ to $h$ doesn't mean that device $h$ will write to $j$, which would violate separation, in the future. However, mediation without adequate I/O hardware support needs to stop this *write* early; otherwise, it invites successful attacks.

## 3.2 Model Definition

This section presents a formal version of the I/O separation model and prove that it satisfies security properties SP1 and SP2. The full formal model is specified in Dafny.

### 3.2.1 I/O Separation Model as a Labeled Transition System

**States.** A system state, denoted by $k$, consists of subjects, objects (which include permissions), and a set of non-NULL partitions as described in Section 3.1. Each driver is associated with a partition ID (i.e., the partition holding the driver) and a set of objects IDs (i.e., objects owned by the driver), and each device additionally is associated with an ID of hardcoded TD; i.e., the hardcoded TD owned by the device. Objects are also associated with partition IDs. FDs and DOs have string values, whereas the value of each TD is a list of tuples comprising a pointer to an I/O object, requested access modes, and new values to write, as defined in Section 3.1.

**Initial State**. The initial state comprises a set of inactive subjects, $s$, and inactive objects $o$, and an empty set of non-NULL partition IDs. The partition ID of all $s$ and $o$ is NULL.

**Operations, State transitions and Traces.** Operations in I/O systems are categorized into four groups: I/O transfers, partition creations and destructions, subjects/objects activations, and subjects/objects deactivations. Operations enable us to define state transitions and traces. A *state transition* occurs when an operation modifies a system state $k$ into an new state $k'$, as specified by the operational semantics. The semantics of each operation are defined using a Dafny method, which takes the current state $k$ and the operation as arguments and returns the resulting state $k'$ as well as a boolean (decision) value $d$, indicating whether the operation is successful. When $d$ is true, the operation is completes successfully; when $d$ is false, the operation is denied and $k' = k$. A *trace* $t$ is a sequence of transitions starting from initial state $k_0$. This section presents the transition defined by each operation below.

**(1) Driver Write**: $DrvWrite(drv\_id, td\_id\_val, fddo\_id\_val)$. The driver with ID *drv_id* attempts to update TDs with a new values *td_id_val* and the FDs and DOs with new values *fddo_id_val*. The I/O kernel performs the following two checks: (1) the driver must be in the same partition as all the objects specified by *td_id_val* and *fddo_id_val*; and (2) for the transitive closure, *tc* of the *TD_state* in the updated system state, $k_{new}$, all the objects to which transfers could be issued by an active device in *tc* must be in the same partition as the device and must not include a hardcoded TD.

Note that driver *writes* might update a state in such a way that cross-partition transfers would be enabled; e.g. see Figure 2.2(c) of Section 2.1. Thus the mediation performed using the transitive closure prevents such cross-partition transfers.

**(2) Device Write:** $DevWrite(dev\_id, td\_id\_val, fddo\_id\_val)$. Like drivers, a device can also issue *write* requests. The state is updated without any checks. This is because device *writes* can only occur either as defined by the device's hardcoded TD, which references the device's objects only, or after appropriate driver *write* operations, which have already been allowed by the I/O kernel; see Section 3.1. Axiom **Ax4** shows this in Section 3.2.2.

**(3) Driver Read:** $DrvRead(drv\_id, read\_objs\_id, obj\_dst\_src)$. The driver with ID *drv_id* attempts to *read* objects identified by their IDs in *read_objs* and stores a subset of the *read* values to objects with IDs specified by the last parameter *obj_dst_src*, which maps destination object IDs to source object IDs. The I/O kernel checks that (1) the driver is in same partition as all objects in *read_objs*; and (2) the *writes* to the destination objects are allowed by the same checks as those specified in the *Driver Write* operation.

**(4) Device Read:** $DevRead(dev\_id, read\_objs, obj\_dst\_src)$. *Device read* performs similar functions as *Driver Read*, and is always allowed under the same conditions for *Device write*.

**(5) Create Empty Partition:**

$EmptyPartitionCreate(new\_pid)$. To create a new I/O partition with ID *new_pid*, the ID *new_pid* must be a fresh ID; i.e., an ID that has not been used before.

**(6) Destroy Empty Partition:** $EmptyPartitionDestroy(pid)$. This operation destroys an empty I/O partition *pid*. The partition ID *pid* must not be NULL and in the set of existing partition IDs in the current state. Furthermore, no subject or object can exist in the partition *pid*. Note that this check ensures that no objects have dangling references to non-existent partition IDs.

**(7) Driver Activation:** $DrvActivate(drv\_id, pid)$. This operation activates a driver with ID *drv_id* into the partition with ID *pid*. The I/O kernel checks that in the current state: (1) The driver is inactive, and (2) the partition *pid* exists in the set of non-NULL partition IDs. Then, it clears all the driver's objects to prevent object reuse between partitions. Finally, the driver's partition ID is updated to *pid*.

**(8) Device Activation:** $DevActivate(dev\_id, pid)$. This operation activates a device with ID *dev_id* into the partition *pid*. The I/O kernel performs similar checks and object clearing to prevent object reuse as in the *Driver activation* operation, without modifying the device's (immutable) hardcoded TD[4].

**(9) External Objects Activation:**
*ExternalObjsActivate*(*obj_ids*, *pid*). This operation activates a set of external objects *obj_ids* into the partition *pid*. For this operation to be allowed, the external objects must be inactive and the partition *pid* must exist in the set of non-NULL partition IDs in the current state. The new state includes these objects with their partiton ID set to *pid* and values cleared.

**(10) Driver Deactivation:** *DrvDeactivate*(*drv_id*). This operation deactivates a driver with ID *drv_id*. The driver must be active. The I/O kernel also computes the transitive closure (*tc*) of the *TD_state* in the current system state and checks that no active device can issue transfers to any objects of the driver *drv_id* in *tc*. The partition ID of the driver and all its objects are set to NULL in the resulting state.

Note that the above check is necessary because otherwise, when the driver is re-activated, active devices could enable cross-partition accesses. For example, consider the case where a partition 1 contains device *i* and driver *h*, in which device *i*'s TD defines a transfer to an object of driver *h*. After driver *h* is deactivated and reactivated into another partition 2, device *i* can issue cross-partition transfers to driver *h* and break I/O separation. In this example, merely clearing all objects of driver *h* does not remove the possible transfer, and hence cannot prevent this attack.

**(11) Device Deactivation:** *DevDeactivate*(*dev_id*). Deactivation operation of a device *dev_id* has the same checks and updates as those of the *Driver Deactivation* operation.

**(12) External Objects Deactivation:**
*ExternalObjsDeactivate*(*obj_ids*, *pid*). The deactivation operation of external objects has the same checks and updates as those of the *Driver Deactivation* operation, except that the operation takes an extra argument *pid* to identify the partition that contains these objects.

Note that the activation operations (i.e., *DrvActivate*, *DevActivate*, and *ExternalObjsActivate*) always clear all objects except hardcoded TDs, as required by security property SP2. Deactivation operations (i.e., *DrvDeactivate*, *DevDeactivate*, and *ExternalObjsDeactivate*) always compute the *transitive closure* of the *TD_state* for the current system state, and check if any remaining active devices after a deactivate operation can issue transfers to any objects to be deactivated. The check is necessary to satisfy security property SP1. Otherwise, active devices after the deactivate operation can issue transfers to inactive objects, and further issue cross-partition transfers after these objects are activated into new partitions. In other words, the check of deactivate operations ensures that *the lifetime of configuration of transfers to an object must be*

[4] If the threat model considers device firmware to be malicious, the correct firmware is re-flashed and provably verified here, as mentioned in Section 2.2.

*strictly less than the lifetime of the object in a partition.*

### 3.2.2 Axioms and Formal Properties

To prove that the I/O separation model is secure, the dissertation uses the following five intuitive axioms (formally defined in Dafny specifications).

**Ax1** All TDs have finite range of values.

**Ax2** Only active subjects can issue transfers to objects.

**Ax3** Hardcoded TDs cannot be accessed by drivers.

**Ax4** A device performs a transfer to an object only if the device *can issue* a transfer (Section 3.1).

**Ax5** The set of all subjects and objects are known *a priori*; i.e., in either active or inactive state.

Axiom **Ax1** is required by the termination of the transitive-closure computation. Axiom **Ax2** states a basic tenet of all state-transition models, while **Ax3** reflects a common access restriction on drivers; Axiom **Ax4** defines the basic device ability to issue transfers. Axiom **Ax5** fixes the domain of the model operations, and is standard in all state-transition models. Note that this axiom does not prevent the modeling of device and driver *install* and *uninstall* operations, which are now represented by *activation* and *deactivation* operations.

**State Invariants.** Each state reachable from the initial state, including the initial state itself, satisfies a number of properties, which are called *state invariants*. These invariants define the security guarantees of the states of the I/O separation model. In this dissertation, a *state k is secure* if and only if it satisfies all the state invariants. For simplicity, this dissertation illustrates one of the invariants below. The model specification in Dafny includes all other state invariants.

**(SI1)** For any *TD_state* of a transitive closure in a system state *k*, if a TD can be *read* by an active device, then objects referenced in that TD (**i**) must be in the same partition as its referenced objects, and (**ii**) must not be hardcoded TDs.

Invariant **SI1(i)** implies that a device must be in the same partition as the objects to which the device can issue direct or indirect transfers; i.e., no cross-partition access possible. This is because devices are always in the same partition as their hardcoded TDs and TDs are objects themselves. Thus, if invariant **SI1(i)** holds, an active device must be in the same partition as all the TDs and other objects to which the device can issue direct or indirect transfers (via TDs). **SI1(ii)** together with **Ax3** reflects the fact that hardcoded TDs are local to devices and are immutable between firmware re-flashings.

**Transition Constraints.** In addition to state invariants, this dissertation also proves a set of properties for every possible state transition. Each *transition constraint* (aka., transition property) specifies properties of the starting state, resulting state, and the operation. Below, this section illustrates the constraint used to prove SP2.

**(TC1)** Only hardcoded TDs can be reused in a new partition with non-NULL partition IDs.

The general properties for a transition from $k$ to $k'$ that this thesis proves are: (1) If $k$ is secure (i.e., satisfies all the state invariants), then $k'$ is also secure; and (2) $k$ and $k'$ satisfy all the transition constraints (e.g., TC1); and (3) if the operation is from the set (*DrvRead*, *DrvWrite*, *DevRead*, or *DevWrite*) and is successful, then the subjects in that operation must be in the same partition as the objects accessed by that operation.

Note that (3) can be straightforwardly derived from the combination of the state invariant **(SI1)**, which ensures only same-partition transfers from *DevRead* and *DevWrite* are allowed, and the semantics of *DrvRead* and *DrvWrite*, which check the partition IDs of objects being accessed; see Section 3.2.1. The lemmas stating that the above transition properties hold are directly used in the inductive proofs of the security theorems, which are shown next.

**Security properties of the I/O separation model.** The following two theorems show that all reachable model states and state-transition sequences are secure. Following tradition ([79]), this thesis proves them separately.

**Theorem 1** (State Invariants)**.** *If state $k_n$ is the resulting state after application of a sequence of $n$ transitions on state $k_0$ and $k_0$ satisfies all state invariants, then $k_n$ also satisfies all state invariants.*

*Proof Sketch.* Proof by induction over $n$. In the base case, when $n = 0$, $k_n = k_0$ satisfies all state invariants. In the induction step, that is $k_i$ transitions to $k_{i+1}$ whereas $i \in (0, n-1])$, and $k_i$ is assumed to fulfill all state invariants, it is obvious that $k_{i+1}$ must fulfill all state invariants due to the property (1) of transitions.

**Theorem 2** (Transition Properties)**.** *If state $k_n$ is the resulting state after application of a sequence of $n$ transitions on state $k_0$ and $k_0$ satisfies all state invariants, then all transitions in the trace satisfy all transition constraints.*

*Proof Sketch.* Proof by induction over $n$ and apply the induction hypothesis and lemmas for the transition properties.

The security guarantees of the I/O separation model follow from the above theorems.

**Corollary 1.** *The I/O separation model satisfies SP1 and SP2.*

Chapter 4

# The Concrete I/O Model for On-Demand I/O Separation

Figure 4.1 illustrates the notions of the concrete interpretation of the I/O separation model in a kernel design; i.e., the *concrete I/O model*, and its instantiations in different I/O kernels. Different concrete I/O models are obtained by selecting different types of separation units (e.g., partitions, red/green partitions, security levels, OS partition), different types of I/O hardware mediation, different types of device activation (e.g., on demand, static), and different policy rules for ephemeral devices (defined below). This chapter describes a concrete I/O model that separates I/O channels of untrusted (red) OS and applications from those of isolated (green) applications, uses existing I/O hardware mediation support, and captures different ephemeral-device activation policies.

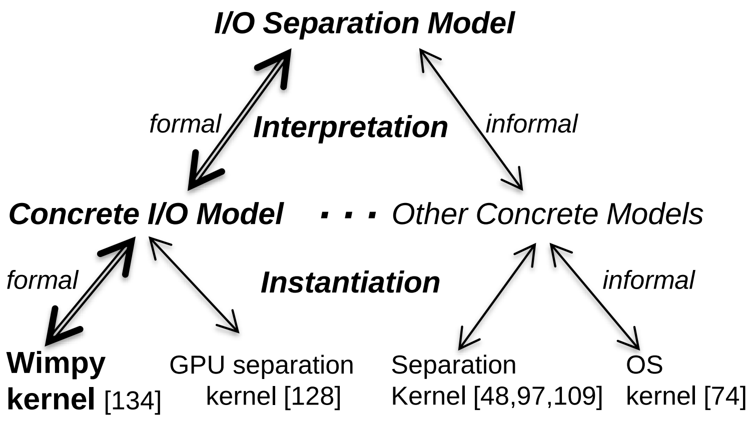


Figure 4.1: Model interpretation and instantiation

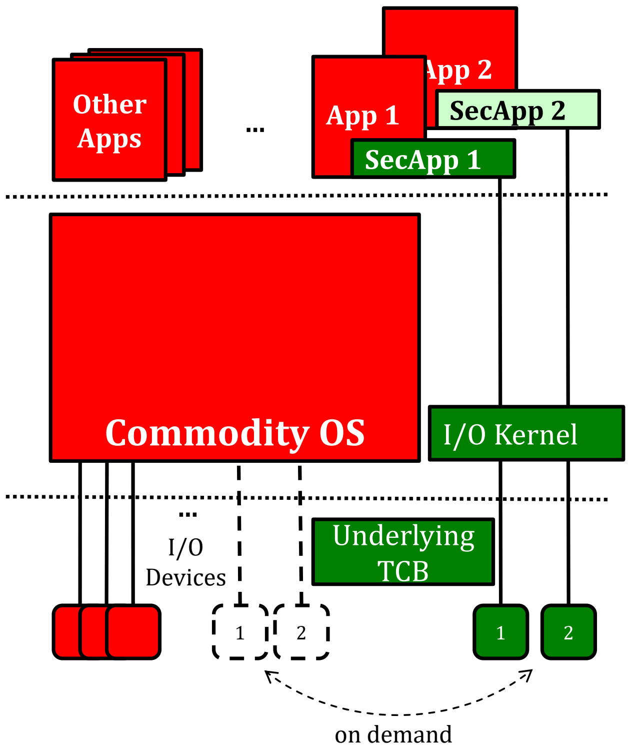


Figure 4.2: Architecture of I/O kernels for on-demand I/O separation.

## 4.1 On-Demand I/O Separation

In the on-demand I/O separation, the untrusted commodity OS manages all commodity hardware resources and devices on the platform most of the time. When a SecApp requires separate I/O to devices, untrusted drivers of the commodity OS release their access to these devices. Then the I/O kernel takes devices, establishes separate I/O channels from SecApp's drivers to these devices, and then runs these drivers in the isolated execution environment of the SecApp (which are provided by the underlying trusted computing base (TCB)). Once the SecApp finishes its execution, the I/O kernel tears down the SecApp's I/O channels, and returns devices to the untrusted OS drivers.

Figure 4.2 shows the architecture of I/O kernels enforcing on-demand I/O separation. The I/O kernel is an add-on trustworthy component, which is isolated from the untrusted commodity OS by the underlying trusted computing base. It executes at the OS's privilege level, dynamically controls hardware resources necessary to establish separate I/O channels between SecApps and I/O devices, and prevents the untrusted OS and applications from interfering with these channels, and vice-versa.

The on-demand I/O separation approach has four significant advantages. First, it enables SecApps to obtain separate I/O channels to *any* devices needed during their sessions, not just to a few devices statically selected at system and application configuration [132]. And with I/O separation, cryptographically enabled channels, which are supported by limited hardware [78, 116, 50, 61], become unnecessary.

Second, it enables trusted audit and control of physical devices without stopping and restarting applications not sharing devices with SecApps. Long-running applications preserve their normal execution when SecApps are loaded and in execution, if they do not share devices with SecApps.

Third, it allows unmodified commodity OSes to have unfettered access to all hardware resources and preserve the entire application ecosystem unchanged. Relinquishing and reclaiming hardware resources for on-demand I/O isolation is handled by non-intrusive OS plug-ins (e.g., loadable kernel modules), without requiring any OS re-design or re-compilation.

Fourth, it offers a significant opportunity for the reduction of the trusted I/O kernel size and complexity, and hence for enhanced verifiability. An I/O kernel can outsource many of its I/O functions to an untrusted OS and use them whenever it can verify the results of the outsourced functions correctly and efficiently. In addition, I/O kernels can de-privilege device drivers, export them to SecApps and mediate accesses to their devices.

## 4.2 Ephemeral Devices

In addition to supporting virtualization of physical devices for use in different virtual machines (VMs), the interpretation of the separation model in different I/O kernels supports a variety of ephemeral devices. These devices are created by multiplexing shared physical devices to create the illusion of separate physical devices[1]. They differ from virtual devices in several ways. For example, unlike virtual devices [19], they have a short lifetime, often as short as a few I/O transfers. They can be virtualized whereas virtual devices may not always be multiplexed. Furthermore, virtual devices support more device functions than ephemeral devices since they support multiple applications in different virtual machines. In comparison, ephemeral devices only implement necessary functions to support individual applications. Also, different I/O kernels implement different policies for ephemeral-device activation, whereas virtual device activation is uniform across different VMs.

[1] An I/O kernel can multiplex a shared bus controller to create separate *channels* [134] and a shared interrupt controller/registers to create separate *devices*. A GPU separation kernel [128] creates separate *windows* on the same GPU. However, the creation of separate ephemeral devices does *not* address I/O separation.

## 4.3 I/O Kernel Partitions

### 4.3.1 Red-Green I/O Partitions

Let an I/O kernel design separate I/O partitions of untrusted (red) OS and application software and isolated (green) applications. An I/O partition can either contain *one driver and its device* and hence a *single* I/O channel, as in a simple isolated application, or *many drivers and devices*, and hence *many* I/O channels as in commodity OS and applications. Thus, commodity OS drivers run and access all their devices in an untrusted I/O partition without any code modification; aka., the “red partition." Note that the red partition must exist because the I/O kernels implementing the green partitions are isolated from the red partition and loaded on-demand, often by a micro-hypervisor or micro-kernel.

When an isolated application requires I/O separation on-demand, the I/O kernel creates a new partition for its drivers, aka., a “green partition," deactivates the required devices from the red partition, and activates them into the green partition, along with the drivers and necessary external objects. Once the isolated application finishes its execution, it invokes the I/O kernel to deactivate its drivers, external objects and devices, activates corresponding devices in the red partition, and destroys the green partition. The I/O kernel may create multiple green partitions to enforce I/O separation for multiple isolated applications, while only one red partition exists.

### 4.3.2 I/O Policies

The I/O kernel enforces different mediation policies for drivers and devices in different partitions, via different mediation mechanisms. Only drivers of a green partition (green drivers) can write to external TDs of devices associated with the partition in the I/O kernel. Then, the (ephemeral) bus controllers configured by isolated-application drivers can read these TDs, and issue transfers to other I/O objects in green partitions; e.g., DOs in green drivers, and FDs/DOs in peripheral devices. The I/O kernel mediates the transfers issued by the bus controllers when green drivers write TDs, such that bus controllers cannot issue direct transfers to objects in a different partition. In the red partition, untrusted red drivers and their devices can issue transfers to objects in various ways supported by different hardware platforms; e.g., device peer-to-peer transfers, I/O multicast and broadcast. The I/O kernel mediates all transfers issued by red drivers and devices, by taking control of the commodity hardware security mechanisms; e.g., setting the IOMMU page tables and resetting the IOTLB. It also prevents configurations that allow P2P transfers on PCI buses, which cannot mediate transfer selectivity.

The I/O kernel uses different mediation mechanisms for red and green drivers and devices, in order to employ hardware mediation mechanisms for performance and mediation benefits. The I/O kernel cannot employ hardware mediation mechanisms for subjects in green partitions, because these mechanisms do not distinguish *ephemeral* green bus controllers that are mapped to the same *physical* bus controller. Consequently, the I/O kernel mediates accesses by green drivers and devices by software, but not via hardware mediation mechanisms.

In practice, green I/O partitions support the export of unprivileged drivers and their local buffers to separate isolated applications. Driver and buffer removal from I/O kernels avoids the performance penalties caused by concurrent access to a single, shared kernel buffer pool and buffer copying to isolated applications, as required to avoid trading off I/O separation for increased performance [74].

## 4.4 Concrete Interpretation of the I/O Separation Model

The concrete model enforces I/O separation on more concrete I/O components than the abstract I/O separation model; e.g., there are more operations using specific hardware mechanisms and additional device and driver activation/deactivation conditions, there are additional state variables that operations must verify, extra state transitions, and more state invariants and transition constraints. As the proofs of the I/O separation soundness theorems become more elaborate here, their machine-checkable specification and verification become indispensable; e.g., see the Dafny specifications and verification.

This section below illustrates the I/O kernel interpretation of the I/O separation model defined in Section 3.2.

*State.* In addition to all state variables of the I/O separation model, the concrete-model states comprise two additional variables: the partition ID of the red I/O partition, and a map that shows which ephemeral devices sharing the same physical device can be active in what partition; e.g., which ephemeral devices can share a partition, which can be in either the red or green partition concurrently, and which could be only in the red or one of the multiple green partitions. This map helps enforce separation policy: it is checked by the concrete model operations before activating a device into a partition.

*Initial State.* In the initial state of the concrete model, the red partition is the only non-NULL partition. Subjects and objects are either active in the red partition, or inactive.

*Operations.* Each operation of the concrete model maps to one and only one operation of the I/O separation model, as shown in Figure 4.3. Among these operations, the red drivers and their external objects do not have activation or deactivation operations. This is because they are never moved to a green partition. Instead, they are activated/deactivated by default at untrusted OS initialization/shutdown. The reason is that green and red drivers for the same device differ; i.e., green drivers are smaller and simpler

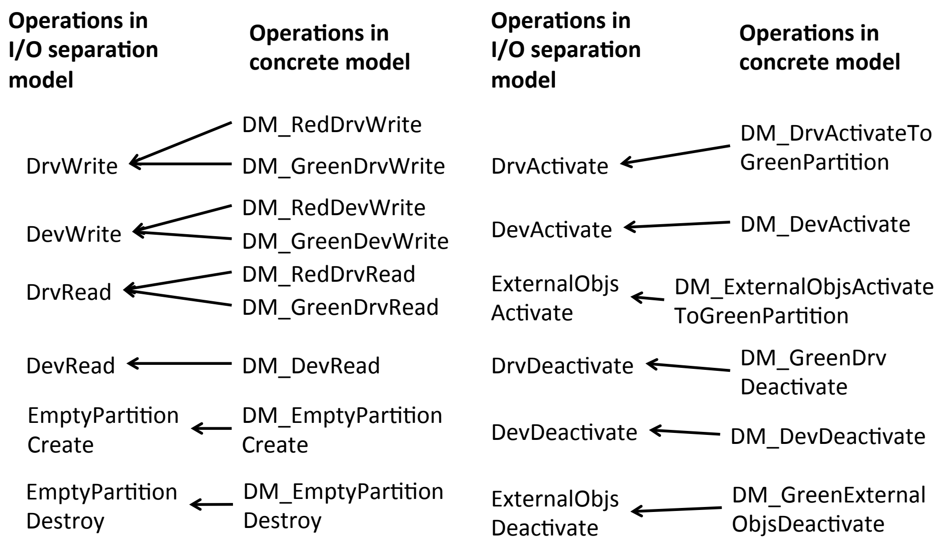


Figure 4.3: Mapping operation labels of the concrete model to the operations of the I/O separation model.

than their red counterparts and rely only on the I/O kernel, which is minimized for formal verification reasons, whereas the larger red drivers can rely on the bloated services of the untrusted OS. Their external I/O objects differ since they are located in their respective partitions and hence isolated from each other; e.g., green drivers and external objects are directly activated into green partitions on demand, but never in a red partition.

The driver *write* operation *DM_RedDrvWrite* also differs from its I/O separation model counterpart. Unlike the *DM_GreenDrvWrite* operation, which relies only on I/O kernel code, the *DM_RedDrvWrite* relies on available I/O hardware mechanisms to block any direct or indirect cross-partition transfers issued by red devices. These mechanisms mediate selectively and include IOMMU and/or PCIe ACS.

Another operation whose interpretation differs from its I/O separation model counterpart is *DM_DevDeactivate*. When a red device is being deactivated, this operation requires no other red device can issue direct or indirect transfers to the device. This requirement can be satisfied by using commodity hardware mechanisms, which selectively mediate transfers.

Finally devices that can be activated in red and green partitions can differ both in capabilities and activation procedures. For example, the same type of ephemeral devices (e.g., ephemeral bus controllers) can provide different capabilities to the red OS drivers and isolated application drivers.

For simplicity of exposition, this thesis illustrates only the interpretation of the state invariant SI1 and transition constraint TC1 (see below) of the separation model. The complete set of invariants and constraints is in the Dafny specifications.

*State Invariants.* A state *ck* is secure if it satisfies the following state invariants:

(SI1c) In any *TD_state* of a transitive closure in system state *ck*, if a red device can *read* a TD, then the objects referenced by the TD must be (i) not hardcoded TDs, and (ii) in the red partition.

(SI2c) All TDs in green partitions (i) only reference objects in the same partition with the TDs, and (ii) do not define direct TD *write* transfers.

The state invariant SI2c is stronger than the property SI1 of the I/O separation model applied to green devices; i.e., no indirect transfer is allowed due to SI2c(ii). This invariant does not forbid green drivers' use of I/O functions; e.g., only green drivers configure transfers by USB host controllers, USB devices, and (ephemeral) interrupt controllers, if any.

The invariant SI2c is important in all concrete models, yet it is ignored in mediation policies of *real* I/O kernel designs. Section 5.2 will illustrate vulnerabilities in the real wimpy kernel design, due to missing critical checks and/or assumptions of this invariant.

*Transition Constraints.* The concrete model defines the following additional transition constraints.

(TC1c) (Same with TC1) Only hardcoded TDs can be reused in a new partition with non-NULL partition IDs.

*Initial State Security.* The initial state is assumed to satisfy state invariant SI1c; e.g., by invoking operations of the underlying system security components. Thus, the initial state maps to a secure state of the I/O separation model, but not to this model's initial state. Nevertheless, it is easy to prove the initial state is secure.

*Theorems and Proofs.* The high-level I/O separation properties for the concrete model are: first, no I/O transfer from the red partition to any green partition; second, no I/O transfer crosses green partitions; and third, only hardcoded TDs can be reused in a new green or red partition. Given the way green and red partitions are set up, these fall out of SP1 and SP2. To prove that the concrete model enforces I/O separation, this thesis needs to prove the interpretation of **Theorem 1** and **2**, and **Corollary 1** of the (abstract) I/O separation model; see Section 3.2.1. The theorem statements remain the same; however, the underlying definitions of state invariants and transition properties are replaced by the ones defined specifically for the concrete model; i.e., SI1c, SI2c, TC1c, etc.

This thesis leverages the simulation relation between the abstract and concrete model in proving these theorems. More concretely, this dissertation formally defines a mapping $f$ from concrete states, operations, and transitions to abstract states, operations, and transitions as illustrated in Figure 4.3. (For brevity, this thesis overloads $f$ for all of the mappings.) This thesis then proves the following lemma.

**Lemma 1.** *If $ck_1$ makes a successful transition to $ck_2$ under operation co (d = true) and $f(ck_1) = k_1$, then exists o and $k_2$ such that $k_1$ can make a successfull transition to $k_2$ under o, s.t. $f(ck_2) = k_2$ and $f(co) = o$.*

The key part of proving the theorems is to show that a successful transition from a secure state in the concrete model always results in another secure state and that the transition properties hold on this transition. Instead of directly proving this statement on the concrete model, we first separate state invariants and transition properties of the concrete model into two groups: one that can be proved with the abstract model's security invariants and transition properties, and the other cannot. For example, SI1c belongs to the first group, while SI2c belongs to the second group. The benefit of separating these properties is to reduce proof effort by using proved properties in the abstract model. Once the two groups of properties are proven, the key part of proving the theorems holds. To prove the first group of properties, we only need to show three detailed lemmas hold: (1) a secure concrete state always maps to a secure abstract state, (2) If a concrete state *ck* maps to a secure abstract state, then *ck* fulfills state invariants in the first group, and (3) If a concrete transition *co* maps to an abstract transition that satisfies transition properties, then the concrete transition satisfies transition properties in the first group. Proof of (1), (2), and (3) is straightforward, because concrete states only differ from abstract states in the treatment of green and red partitions and ephemeral devices, and fulfill stronger invariants; e.g., SI2c. A direct proof of the second group of properties is trivial, because concrete operations enforce these properties in their specifications. Consequently, proving the theorems in the concrete model takes fewer Dafny code than proving them in the abstract model, i.e., 15733 lines of Dafny code for the concrete model, versus 24934 lines of Dafny code for the abstract model.

The thesis presents a more elaborated proof sketch to the first group of properties as follows.

**Lemma 2.** *For the abstract I/O separation model, let K denotes its states, $\xrightarrow{o}$ denotes its successful transitions, $\phi$ denotes all state invariants and $\omega$ denotes all transition properties. For the concrete model, let CK denotes its states, $\xrightarrow{co}$ denotes its successful transitions, and $\Phi$ denotes all state invariants. $\forall ck_1 \xrightarrow{co} ck_2, ck_1, ck_2 \in CK$, if $\Phi(ck_1)$, then $\Phi_1(ck_2), \Omega_1(ck_1, ck_2)$, in which $\Phi_1$ are the state invariants of the concrete model in the first group, $\Omega_1$ are the transition properties in the first group.*

*Proof Sketch.* The three detailed lemmas that can be trivially shown in the previous paragraphs can be written as follows:

$\forall ck \in CK, k \in K, f(ck) = k, \Phi(ck) \implies \phi(k)$ (Proved detailed lemma 1)

$\forall ck \in CK, k \in K, f(ck) = k, \phi(k) \implies \Phi_1(ck)$ (Proved detailed lemma 2)

$\forall ck_1, ck_2 \in CK, k_1, k_2 \in K, co, o, ck_1 \xrightarrow{co} ck_2, f(ck_1) = k_1, f(ck_2) = k_2, k_1 \xrightarrow{o} k_2,$
$\omega(k_1, k_2) \implies \Omega_1(ck_1, ck_2)$ (Proved detailed lemma 3)

By proved detailed lemma 1, and $\mathit{\Phi}(ck_1)$, we have $\phi(k_1)$ (1)

By $ck_1 \xrightarrow{\text{co}} ck_2$, Lemma 1 and definition of $f$, we have $\exists k_2 \in K, o, k_1 \xrightarrow{\text{o}} k_2, f(ck_2) = k_2$ (2)

By $k_1 \xrightarrow{\text{o}} k_2$ in (2), (1), and Theorem 1 and 2, we have $\phi(k_2), \omega(k_1, k_2)$ (3)

By $f(ck_2) = k_2$ in (2), (3), and proved detailed lemma 2, we have $\mathit{\Phi}_1(ck_2)$ (4)

By $ck_1 \xrightarrow{\text{co}} ck_2$, definition of $f$, (2), (3), and proved detailed lemma 3, we have $\mathit{\Omega}_1(ck_1, ck_2)$ (5)

With (4) and (5), Lemma 2 holds □

Chapter 5

# Application: On-Demand I/O Separation For Peripheral Devices Exclusively Owned By SecApps

This chapter provides an overview of the Wimpy Kernel [134] and describes an instantiation of the concrete model to the Wimpy Kernel design. This enables us to identify heretofore unknown vulnerabilities of Wimpy Kernel, and note that even careful but informal designs can benefit from formal analyses.

The Wimpy Kernel aims to enforce on-demand I/O channel separation for SecApps, and enable SecApps' drivers to use I/O channels to *exclusively* access their devices. When a SecApp requires I/O separation, untrusted drivers of the commodity OS release their access to any devices requested by the SecApp's drivers. Then the Wimpy Kernel takes these devices, establishes separate I/O channels from the SecApp's drivers to these devices, and then runs these drivers in the isolated execution environment of the SecApp (which is provided by the underlying *micro-hypervisor*). Once the SecApp finishes its execution, the Wimpy Kernel tears down the SecApp's I/O channels and returns devices to the untrusted OS drivers.

The goal of Wimpy Kernel is to separate the I/O channels used by untrusted OS drivers from those of the SecApp drivers. That is, any sensitive I/O data in I/O channels of SecApps *cannot* be observed or altered by drivers and devices of the untrusted OS, or other SecApps. Yet, these data can be *arbitrarily* accessed by any drivers and devices associated with different I/O channels, as long as they belong to the same SecApp. In other words, no channel separation is enforced inside a SecApp and the untrusted OS.

| **Operations in (correct) WK design** | | **Operations in concrete model** | **Operations in (correct) WK design** | | **Operations in concrete model** |
|---|---|---|---|---|---|
| WS_OSDrvRead | = | DM_RedDrvRead | WS_WimpDrvWrite | = | DM_GreenDrvWrite |
| WS_WimpDrvRead | = | DM_GreenDrvRead | WS_OSDevWrite | = | DM_RedDevWrite |
| WS_DevRead | = | DM_DevRead | WS_WimpDevWrite | = | DM_GreenDevWrite |
| WS_OSDrvWrite | = | DM_RedDrvWrite | | | |

| **Operations in (correct) WK design** | | **Operations in concrete model** |
|---|---|---|
| WK_WimpDrvsDevs_ Registration_CreatePartition | = | DM_EmptyPartitionCreate |
| WK_WimpDrvsDevs_ Registration_AssignWimpDrvsDevsObjs | = | DM_DevDeactivate \|\| [DM_DevActivate, …] \|\| [DM_DrvActivateToGreenPartition, …] \|\| DM_ExternalObjsActivateToGreenPartition |
| WK_WimpDrvsDevs_Unregistration | = | DM_GreenExternalObjsDeactivate \|\| [DM_GreenDrvDeactivate, …] \|\| [DM_DevDeactivate, …] \|\| [DM_DevActivate, …] \|\| DM_EmptyPartitionDestroy |

Figure 5.1: Wimpy Kernel operations represented as concatenations of concrete-model operations.

## 5.1 Instantiation of the Concrete Model in Wimpy Kernel

A sound Wimpy Kernel design (i.e., whose Dafny specs are proved to be sound) reuses the state, hardware mediation mechanisms, state invariants, transition constraints, and initial state definitions of the concrete model. Also, it implements specific operations as *concatenations* of concrete-model operations; see Figure 5.1. Furthermore, activation properties of ephemeral devices (e.g., ephemeral USB bus controllers) that are not common to instantiations of the same concrete model in other I/O kernels are specified here. That is, ephemeral devices of the Wimpy Kernel that are activated in green partitions can differ from those deactivated from the red partition since they can provide different functions to SecApps and the OS. Thus, the Wimpy Kernel can only activate ephemeral devices into either green or red partitions (but not both), and only if the corresponding ephemeral devices are inactive in the red or green partitions, respectively. Thus, a sound Wimpy Kernel design instantiates the concrete model, and hence I/O separation becomes fairly simple[1], taking 4450 lines of Dafny code in total.

[1] A sound Wimpy Kernel *implementation* of a device move between red and green partitions flushes the IOTLB, just as any OS kernel would context switch an IOMMU. Similarly, other cache entries of hardware mediation mechanisms in the micro-hypervisor are also invalidated. Otherwise, caches would store stale access permission, and hence cause address-space isolation to fail.

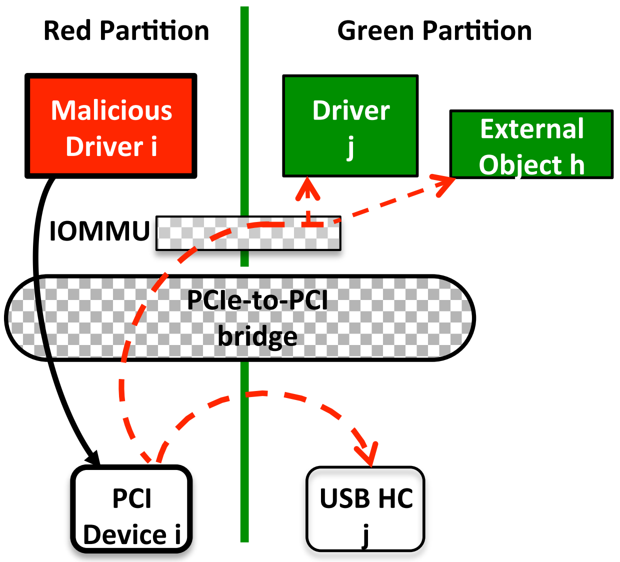


Figure 5.2: Wimpy Kernel vulnerability 1.

## 5.2 Vulnerabilities of the Wimpy Kernel

To prove the soundness of Wimpy Kernel design specs we had to formally specify I/O hardware separation properties and configuration restrictions, which are same with the ones in the concrete model. Then we had to compare the *sound* kernel specifications[2] with the original design [134] to discover vulnerabilities. The thesis illustrates the two vulnerabilities we found below.

*Vulnerability 1: A violation of red-green separation. Red devices can issue inadequately mediated transfers to green objects; i.e., to driver/device and external objects.*

This vulnerability is caused by a violation of the state invariant SI1c of the sound Wimpy Kernel design described above. In Figure 5.2, let a red PCI device *i* share a PCI bus with a green USB Host Controller (USB HC) device *j*, and let that bus be connected to the IOMMU via a PCIe-to-PCI bridge. Now malicious red driver *i* *writes* configuration values to a TD of its device *i* denoting a transfer to green driver *j* or external I/O object *h*. This driver *write* operation is allowed since the Wimpy Kernel assumes that device *i*'s transfers outside the red partition are *always* blocked by the IOMMU. However, as shown in Section 2.1, *Example 4* (and Figure 2.3), some PCIe-to-PCI bridges prevent the IOMMU from distinguishing device transfers originating from the same PCI bus, and hence allow red device *i*'s transfers to the green partition, violating the red-green separation. Alternatively, the malicious red driver *i* can configure its device *i*'s TD to perform a peer-to-peer transfer to green USB HC device *j* on the shared PCI bus. The

[2] Analysis of failed soundness proofs for kernel design specs is insufficient as they do not always cause exploitable vulnerabilities, even when they violate I/O separation. For example, objects can be reused when moved from the red partition to a green partition, as there is no sensitive I/O data in the red partition that cannot be observed in a green partition.

Wimpy Kernel incorrectly assumes that PCI buses mediate such transfers; see Section 2.1, *Example 1* and Figure 2.2(a).

To remove this vulnerability, the sound Wimpy Kernel design (proved in Dafny specs) requires restrictions of hardware configurations to prevent such transfers. For example, specified I/O hardware restrictions are satisfied when green USB host controllers are on PCIe (not PCI) buses only, and enable IOMMU and PCIe ACS to mediate transfers by red devices. Then, the Wimpy Kernel must invoke operations provided by this hardware configuration to implement secure initial state, and operations of the sound design outlined above.

*Vulnerability 2: A violation of green-green separation. Green drivers can issue indirect writes to external TDs and cause their devices initiate transfers to other green partitions after reading these TDs.*

This vulnerability is caused by a violation of state invariant SI2c (ii) of the sound Wimpy Kernel design described above, which denies *indirect* transfers in green partitions. Figure 5.3 illustrates how to exploit this vulnerability in practice.

At time (1), malicious driver *i writes* to an external TD. The new TD value defines a *write* transfer to the TD itself.

At time (2) the same driver *writes* to its device *i*'s (USB HC *i*'s) transfer descriptor TDi enabling it to *read (only)* the external TD. Wimpy Kernel allows driver *i* to perform both operations because all direct transfers defined in the modified TDs reference objects in the local green partition 1, and their new values enable *only* local-partition transfers.

At time (3), under the control of malicious driver *i*, the USB HC device *i reads* the value of the external TD, which now allows device *i* to *write* new configuration values for this TD. Next, device *i overwrites* the external TD thereby enabling transfers to driver *j* and device USB HC *j*. The Wimpy Kernel allows USB HC device *i* to *write* the configuration values to external TD since this operation is direct and local to the green partition 1. Note that the Wimpy Kernel does *not* ensure that the values written by the device prevent indirect transfers.

At time (4), the USB HC device *i* issues an *indirect* transfer to device USB HC *j* and driver *j* of partition 2 via the external TD, which violates the green-green partition separation. This indirect transfer is similar to the one illustrated in Section 2.1, *Example 3*; see Figure 2.2(c).

To remove this vulnerability, the sound Wimpy Kernel design specs prevent all direct TD *write* transfers by devices in their local green partition. This can be implemented as a simple check in the Wimpy Kernel; e.g., interface data structures submitted by wimp drivers must not define write transfers to any interface data structures or device registers that define transfers. This type of mediation check is not included in the original Wimpy Kernel design [134].

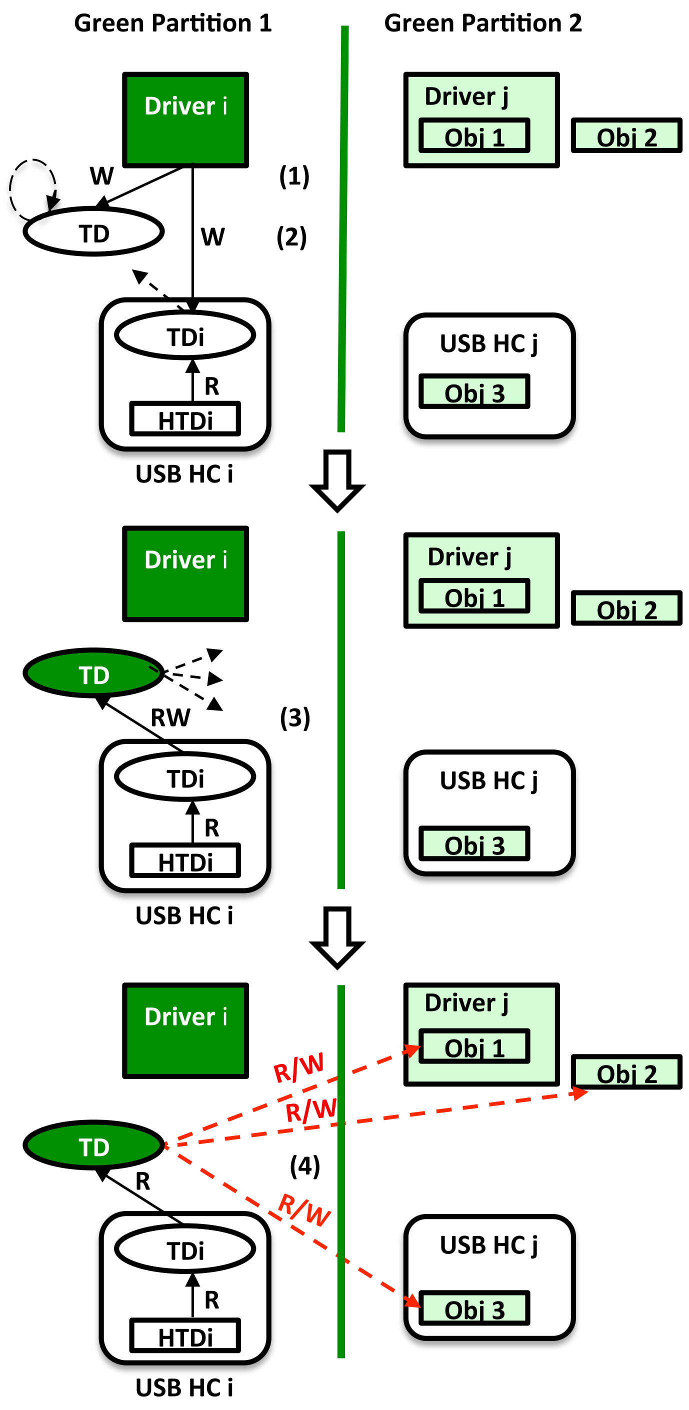


Figure 5.3: Wimpy Kernel vulnerability 2.

Chapter 6

# Application: On-Demand I/O Separation For Display Shared Among Applications

This chapter describes the GPU Separation Kernel [128], which provides the trusted display service to SecApps, by sharing a GPU between commodity applications and SecApps. The GPU Separation Kernel fulfills the goal by accomplishing two *orthogonal* tasks: creating separate ephemeral GPUs, and enforcing I/O separation for SecApps and their ephemeral GPUs. Among these two tasks, separating ephemeral GPUs provides *necessary* input to enforcing I/O separation, i.e., defining the set of subjects and objects.

Unfortunately, creating separate ephemeral GPUs in the GPU Separation Kernel is non-trivial. Past approaches have failed to separate ephemeral GPUs and provide trusted display on commodity platforms that use modern GPUs, even when taking exclusive control of physical GPUs. For example, full GPU virtualization encourages the sharing of GPU address space with multiple virtual machines *without* providing adequate hardware protection mechanisms such as address-space separation and instruction execution control.

This chapter describes how the GPU Separation Kernel *informally* creates separate ephemeral GPUs to enable a new trusted display service with minimal trusted code base and maintains full compatibility with commodity computing platforms. The GPU Separation Kernel (1) defines different types of GPU objects, (2) mediates access to security-sensitive objects, and (3) emulates objects whenever required by commodity-platform compatibility. The kernel employs a new address-space separation mechanism that avoids the challenging problem of GPU instruction verification without adequate hardware support.

After separating ephemeral GPUs, the GPU Separation Kernel uses the same I/O separation approach between SecApps and ephemeral GPUs as in the Wimpy Kernel [134], and hence has the risk of suffering the same vulnerabilities as described in Chapter 5. In the end of this chapter, the dissertation informally analyzes the GPU Separation Kernel with respect to these vulnerabilities.

The overall implementation of the I/O kernel has a code base that is two orders of magnitude smaller than other similar services, such as those based on full GPU virtualization. Performance measurements show that the trusted-display overhead added above that of the underlying trusted system is fairly modest.

## 6.1 Motivation

A trusted display service provides a protected channel that assures the confidentiality and authenticity of content output on selected screen areas. With it users can rely on the information output by a security-sensitive application (SecApp) without worrying about undetectable screen “scraping" or “painting" by malicious software on commodity systems, i.e., when the display output is surreptitiously read or modified by a compromised commodity operating system (OS) or application (App).

Security architectures that isolate entire SecApps from untrusted commodity OSes and Applications (Apps) [77] implement trusted display functions via trusted path [132, 134]. That is, a user's explicit activation of the trusted-path effectively removes all untrusted OS and Apps access to the display device and assigns the device to a SecApp for the entire duration of a session. Unfortunately, the exclusive use of display devices via trusted path does not allow *both* untrusted OS/Apps and SecApps to output content concurrently on a user's screen, because untrusted output cannot be displayed until after the trusted path releases the screen at the end of the SecApp session. As a consequence, it would not be possible to maintain the typical multi-window user experience for applications that comprise both trusted and untrusted components and use the same display screen.

**Problem.** Some past approaches that allow trusted display of output with different sensitivity on the same screen concurrently have been based on encapsulating and protecting graphics cards within high-assurance security kernels [36, 104, 42]. In addition to requiring changes of commodity OSes, adopting such an approach for the entire graphics processing unit (GPU) would not work since the complexity of modern GPU functionality[1] would rule out maintaining a small and simple code base for the security kernel, which is a prerequisite for high assurance. For example, the size of Intel's GPU driver for Linux 3.2.0 - 36.57 has over 57K SLoC, which is more than twice the size of a typical security kernel [131]. Furthermore, GPU functions operate asynchronously from the CPUs [111, 129] to improve graphics performance and introduce concurrency control for multi-threading in the trusted code base. This would invalidate all existing proofs that assume single-thread operation [60, 113].

[1] Modern GPUs include graphics/computation accelerators [87, 40]. They are equipped with hundreds of processors [72] to provide complex functions of 2D/3D hardware rendering, general-purpose computing on GPU (GPGPU), and hardware video encoding/decoding.

Full GPU virtualization [111, 108] can be used to enable concurrent display of both trusted and untrusted output on a user's screen without requiring commodity OSes/Apps modification. However, full GPU virtualization, which is largely motivated by improved performance, relies on address-space sharing between different virtual machines (VMs) and the GPU *without* providing adequate hardware mechanisms (e.g., address-space separation and instruction execution control) for protecting different VMs' code and data within the GPU. As a concrete example, this chapter illustrates a class of new attacks that exploit the inadequacy of address-space separation on fully virtualized GPUs; viz., Section 6.2.2. Moreover, full GPU virtualization intrinsically requires a large trusted code base. For example, supporting native GPU drivers/Apps requires emulating *all* accesses to *all* GPU configuration registers for the VMs scheduled to access the GPU. Thus, adopting full GPU virtualization for high-assurance trusted display would be impractical.

**Solution.** The trusted display design presented in this chapter satisfies the following four requirements:

- It allows the confidentiality and authenticity of display contents to be assured to whatever degree of rigor deemed necessary by minimizing and simplifying the trusted-display code base.
- It avoids redesign and modification of underlying trusted-system components, and preserves their existing properties such as proofs of high-assurance micro-kernels and micro-hypervisors [60, 113].
- It preserves full compatibility with commodity platforms; i.e., it does not require any modification of commodity OS/Apps code and GPU hardware or reduce their functionality.
- It maintains a typical user's perception and use of application output and relies on easily identifiable screen geometry; e.g., it uses different windows for trusted and untrusted screen areas.

The central component of our trusted display design is a GPU Separation Kernel (GSK) that (1) distinguishes different types of GPU objects, (2) mediates access to security-sensitive objects, and (3) emulates object access whenever required by commodity-platform compatibility. The GSK employs a new address-space separation mechanism that avoids the challenging problem of GPU instructions verification without adequate hardware support. The implementation of the trusted display service has a code base that is two orders of magnitude smaller than other similar services, such as those based on full GPU virtualization.

**Outline.** Section 6.2 provides a brief overview of GPU functions to enable the reader understand the vulnerabilities of GPU virtualization to adversary attacks, and challenges of trusted display on commodity platforms. Section 6.3 defines the adversary threats, security properties that counter them, and an informal security model that satisfies these properties. Section 6.4 describes the detailed design and implementation

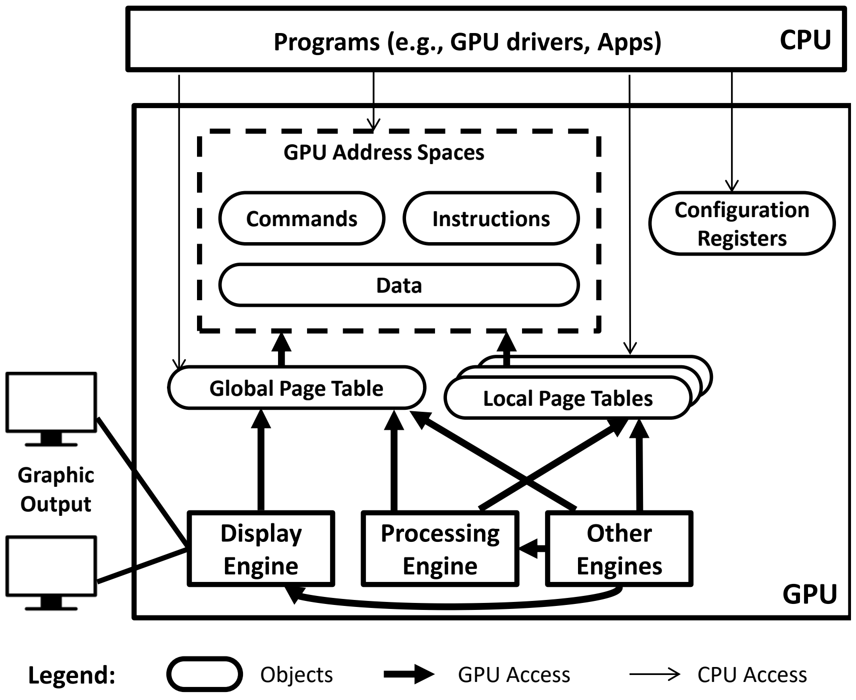


Figure 6.1: Overview of a modern GPU architecture.

of the trusted display system, and Section 6.5 evaluates the implementation. Section 6.7 informally maps the GPU Separation Kernel to the concrete model, and presents the vulnerabilities of the GPU Separation Kernel.

## 6.2 Commodity GPU Architecture and Security Vulnerabilities

This section presents an overview of common architecture features of modern GPUs to enable an unfamiliar reader understand their vulnerability to attacks. The GPU architecture described herein is common to widely available commodity devices from vendors such as Intel [54, 2], AMD [6], Nvidia [4], and ARM [3, 16].

### 6.2.1 GPU Architecture Overview

CPU programs (e.g., GPU drivers and Apps) control GPU execution via four types of *objects*, namely data, page tables, commands, and instructions that are stored in GPU memory, and GPU configuration registers; viz., Figure 6.1.

CPU programs produce the instructions and commands that are executed by GPU hardware. For example, instructions are executed on GPU processor cores, process input data, and produce results that are used by display engines. In contrast, commands are executed by dedicated command processors and are used to configure the GPU with correct parameters, e.g., to specify the stack base address used by instructions. Groups of commands are submitted for processing in dedicated command buffers; for example, they are received in input (*ring*) buffers from drivers and (*batch*) buffers from both applications and drivers.

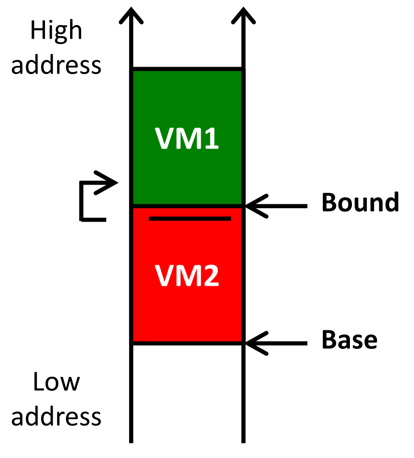
As shown in Figure 6.1, a GPU also contains several *engines* [111, 54, 9], such as the *processing engine* and *display engine*. The processing engine executes instructions on multiple GPU cores for computation acceleration. It references memory regions known as the *GPU local address space* via the *GPU local page tables*. The display engine parses screen pixel data stored in *frame buffers* according to the engine's configurations, and outputs images for display. Other engines perform a variety of functions such as device-wide performance monitoring and power management. Other engines could access the processing engine and the display engine, e.g., to sample the performance of the latter two engines.

The display engine defines several basic configurations for *frame buffer* presentation such as geometry and pixel formats. Furthermore, it provides the data paths from *frame buffers* to external monitors. For example, the screen output may comprise a combination of multiple screen layers, each of which contains a separate *frame buffer*. In this case, GPUs support a *hardware cursor* as the front layer of the screen and display it over the primary image. Since a single GPU may be connected to multiple screen monitors, a monitor may consume the *same* frame buffers as another monitor, which implies that GPU memory protection requires a controlled-sharing mechanism. Furthermore, an image presented on a screen may be torn as the result of frame-buffer updates by CPU programs during screen refreshing. To address this synchronization problem, display engines of modern GPUs also provide a *V-Sync interrupt* to notify CPU programs of the time when it is safe to update a frame buffer [118].

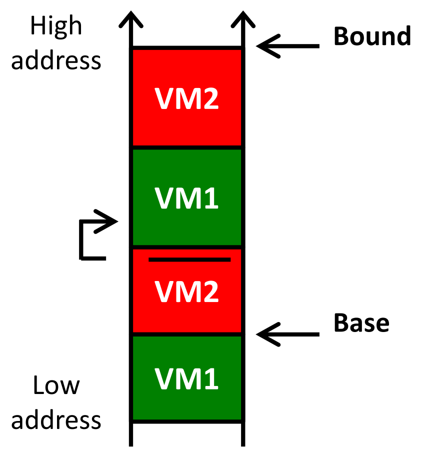
Although the GPU architecture illustrated in Figure 6.1 is common to many commodity GPUs, some of these GPUs differ in how memory is accessed and managed. For example, Intel's GPUs use a *global page table* (GGTT) for memory access in addition to local page tables. The GGTT maps the memory region referred to as the *GPU global address space*, which includes *frame buffers*, command buffers, and GPU *memory aperture*, which is shared between CPU and GPU. In contrast, AMD and Nvidia GPUs do not have a GGTT and allow direct access to GPU physical memory address space[2]. Because of this, GPU memory access also differs in different GPUs; e.g., the processing engine of Nvidia's GPU can access only the local address space [108, 59], whereas Intel and AMD[3] GPUs can also access the global address space [54, 6, 2].

[2] To simplify presentation, this thesis considers that these GPUs use a GGTTs with flat mappings (e.g. virtual addresses are identical with physical addresses) even though the GGTT does not exist in these GPUs.

[3]Although this feature is undocumented by AMD's GPUs, it is supported in the open source GPU driver provided by AMD [6].



(a) Simple



(b) General

Figure 6.2: GPU address-space separation attacks.

### 6.2.2 Address Space Separation Attacks

A fully virtualized GPU shares its global address space with multiple virtual machines (VMs) to support *concurrent* accesses to its memory [111]. For example, while the GPU's display engine fetches a VM's *frame buffer* to display its content, the GPU's processing engine generates content for other VMs' *frame buffers*. Furthermore, the hardware design of Intel and AMD GPU's processing engines allows instructions to access the global address space. Because full GPU virtualization supports native drivers, any malicious VMs can submit GPU instructions that access another VM's GPU data for screen output.

Figure 6.2(a) illustrates this simple attack. Here, a malicious VM2 submits valid GPU instructions that ostensibly address GPU memory inside VM2's address space but in fact access victim VM1's GPU memory. For example, VM2 can submit malicious instructions that contain large address offsets which fall into VM1's GPU address space[4]. Unless an additional "base-and-bound" mechanism for address space protection is supported by GPU address translation, the GPU's processing engine would allow the malicious VM2 to access victim VM1's GPU output data thereby violating confidentiality and authenticity.

Note that some fully virtualized GPUs support a single "base-and-bound" pair of registers for address space protection; e.g., Intel GPUs limit memory access range of GPU instructions by correct setting of the "base-and-bound" register pair for GPU command execution [54]. These GPUs can mediate memory accesses and deny address-space violations by GPU instructions and commands issued by malicious VMs [111].

Unfortunately, a *single pair* of base and bound registers is insufficient to counter all address-space separation attacks mounted by malicious VMs. These attacks are enabled by another important perfor-

[4]Other full GPU virtualization approaches [108] are also subject to such attacks.

mance optimization of full GPU virtualization. That is, address space "ballooning" [111] allows the GPU to directly access virtual memory at addresses provided by guest VMs. This optimization improves GPU memory-access performance and greatly reduces complexity of GPU programming. Without it, trusted code would have to translate the referenced GPU virtual addresses for every object, and even recompile GPU instructions on the fly. For example, AMD's GPU instructions perform register-indirect memory accesses, and hence would require such recompilation for address translation.

However, address space ballooning allows the GPU memory of a guest VM to be mapped into two or more non-contiguous blocks in GPU global address space; e.g., one in GPU memory aperture for exchanging data between CPU and GPU, and the other in non-aperture space for holding GPU data. As a consequence, the separated memory blocks cannot be protected by the setting of the single pair of "base-and-bound" registers in the GPU commands such as with the Intel GPU. As illustrated in Figure 6.2(b), malicious VM2 uses the simple attack of Figure 6.2(a) but this time it can access victim VM1's GPU memory despite base-and-bound protection, because one of VM1's GPU memory blocks falls between two of VM2's non-contiguous memory blocks. It should be noted that the simple attack succeeds for other GPUs, not just Intel's; e.g. some instructions in AMD GPUs can perform register-indirect memory accesses, without specifying added address-space protection [9].

### 6.2.3 Challenges of Commodity Platforms

Implementing a trusted display service on untrusted commodity OS and hardware platforms that support SecApp isolation faces three basic challenges.

*Incompatibility with commodity platforms.* The goal of maintaining object-code compatibility with untrusted OSes that directly access GPU objects in an unrestricted manner poses a dilemma. If one redesigns and re-implements GPU functions on commodity OSes to block memory accesses that breach address space separation, one introduces object-code incompatibility. If one does not, one forgoes trusted display. To retain compatibility, accesses to GPU objects by untrusted commodity OS/Apps code must be emulated by the trusted-system, yet emulating accesses to *all* GPU objects inevitably increases the trusted code base and makes high-assurance design impractical.

*Inadequate GPU hardware protection.* The inadequacy of the hardware for memory protection has already been noted in the literature for Intel GPUs [111]. The address-space separation attack by malicious GPU instructions of Section 6.2.2 illustrates another instance of this problem and suggests that simplistic software solutions will not work. For example, verifying address offsets of GPU instructions before execution does not work because operand addressing cannot always be unambiguously determined due to indirect branches [54] and register-indirect memory accesses [54, 9].

*Unverifiable code base.* Even if, hypothetically, all the OS/Apps functions that access GPU objects could be isolated and made tamper-proof, their code base would be neither small (that is, tens of thousands of SLoC) nor simple, and hence the formal verification of their security properties would be impractical. A large number of diverse GPU instructions and commands spread throughout different drivers and application code provide access to a large number of GPU objects; e.g., a GPU can have 625 configuration registers and 335 GPU commands, as shown in Section 6.5. Furthermore, since the underlying trusted base (e.g., micro-kernel or micro-hypervisor) must protect different SecApps on a commodity platform, the functions that access GPU objects directly must be implemented within the trusted base. Hence, these functions' code would have to preserve all existing assurance of the underlying trusted base. In other words, their security properties and proofs must compose with those of the trusted base. These challenges have not been met to date.

## 6.3 Security Model

This section defines the threats posed by an adversary to trusted display and present security properties that counter these threats. Furthermore, this section presents an *informal GPU security model* to separate ephemeral GPUs by satisfying those properties in commodity systems.

### 6.3.1 Threats

An adversary can leak a SecApp's security-sensitive output via *screen scraping* attacks whereby the content of display output in a GPU's memory is read by a malicious program of a compromised commodity OS/App or SecApp. The adversary can also modify the SecApp's output content, configuration (e.g., geometry, pixel format, *frame buffer*'s base address) via *screen painting* attacks whereby a malicious program modifies GPU memory and configuration registers. To breach the separation of ephemeral GPUs, the adversary does not need to break the I/O separation for the physical GPU; e.g., accessing it via adversary-controlled devices on the same bus. Instead, the adversary only needs to access his/her own ephemeral GPUs. For example, to launch both attacks the adversary can breach the separation of GPU's address spaces. These breaches can be implemented by unauthorized access to GPU objects, either directly by CPU programs (e.g., drivers, applications), or indirectly by GPU commands and instructions that cause the GPU to access other GPU objects in an unauthorized manner. Furthermore, the adversary can manipulate the display engine's data paths and overlay a new *frame buffer* over a SecApp's display thereby breaking the integrity of SecApps' display output without touching its contents.

In this work, the thesis ignores I/O separation attacks, which have already been addressed in the literature [134, 132]. Instead, the thesis focuses on creating and separating ephemeral GPUs, which is orthogonal from enabling I/O separation.

In this work, the thesis does not consider hardware, side-channels, and shoulder-surfing attacks [49]. The thesis does not assume GPU firmware is benign, because it can use existing approaches to verify the correct GPU firmware and register content [45]. The thesis also omits denial of service (DoS) attacks, such as manipulation of GPU configurations to disable screen output; e.g., disable-then-resume GPU and color shifts. For a well designed SecApp, it would be difficult for an adversary to launch a DoS attack that would remain unnoticed by an observant user.

### 6.3.2 Security Properties

A security model for trusted display on commodity systems must satisfy three abstract properties (defined below) that are intended to counter an adversary's threats. To express these properties, this thesis partitions the GPU objects into two groups: security *sensitive* and *insensitive* objects. Intuitively, the security-sensitive GPU objects are those that can be programmed by untrusted software (e.g., malicious drivers, applications) to break the confidentiality or authenticity of trusted display output, and those which can be tainted by access to other sensitive GPU objects. For example, sensitive GPU objects include directly accessible objects, such as *frame buffers*, page tables, configuration registers, and objects that can affect the security of other objects, such as GPU commands, and instructions, which can modify GPU page table structures. Furthermore, because GPU objects are mapped into GPU address spaces, the corresponding virtual and physical GPU memory regions are regarded as *sensitive*. In contrast, the security-insensitive GPU objects cannot affect the confidentiality and authenticity of trusted display even if they are manipulated by malicious software.

The three security properties that must be satisfied by trusted display designs and implementations are expressed in terms of the separation of sensitive-insensitive objects and their accesses, complete mediation of accesses to sensitive objects, and minimization of the trusted code base that implements the separation and mediation properties.

**P1. Complete separation of GPU objects and their accesses.** The trusted display model must partition *all GPU objects* into security-sensitive and security-insensitive objects and must define all *access modes* (e.g., content read, write, configuration modification) for the security sensitive objects and their memory representations.

**P2. Complete mediation of GPU sensitive-object access.** The trusted display model must include a

*GPU separation kernel* that must satisfy the following three properties. The kernel must:

(1) mediate *all* accesses to the security-sensitive objects according to a defined GPU access mediation policy;

(2) provide a GPU access-mediation policy that defines the access invariants for security-sensitive objects; and

(3) be protected from tampering by untrusted OS/Apps and SecApps[5].

**P3. Trusted code base minimization.** The GPU separation kernel must: (1) have a small code base to facilitate formal verification, and (2) preserve the existing assurance of the underlying Trusted Computing Base (TCB) necessary for its protection.

### 6.3.3 Separation of GPU Objects and Accesses

All security properties of trusted display ultimately rely on the complete separation of GPU objects and their accesses by GPU design, implementation and GPU separation kernel (GSK) discussed below. This requires the analysis of all interfaces between the software components (untrusted commodity OS/Apps and SecApps) and GPU objects. The results of this analysis, which are abstracted in Figure 6.3 and Table 6.1, enable the design of the access mediation mechanism, policy, and access emulation. For example, this analysis shows that SecApps may provide only display content and geometry configurations to the GPU without sending any commands or instructions, and hence do *not* need direct access to GPU objects. Hence, all their direct accesses to GPU objects are blocked by the GPU separation kernel.

### 6.3.4 GPU Separation Kernel

#### Access Mediation Mechanism

The mediation mechanism of the GSK distinguishes between two types of access outcomes, namely direct access to security-insensitive GPU objects, and verified (mediated) access to security-sensitive GPU objects. Every CPU access to sensitive objects must be mediated even though GPU hardware lacks mechanisms for intercepting all GPU commands and instructions individually at run time. Since most GPU commands and instruction references cannot be mediated at run time, they must be verified *before* submitting them to the GPU. Although previous work [111] illustrates how to verify GPU commands, it does not address the verification of GPU instructions, which is more challenging, as argued in Section 6.2.3.

[5]The isolation of the GPU separation kernel can be easily achieved using the services of existing micro-kernel or micro-hypervisor security architectures [60, 134]

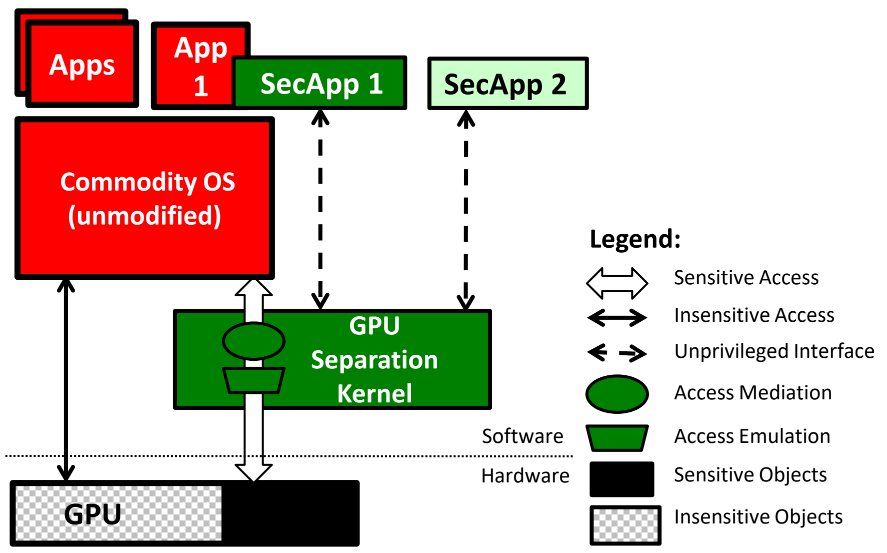


Figure 6.3: GPU separation kernel architecture.

GPU instructions are limited to three types of operations: arithmetic/logic operations, control flow operations, and memory access operations [54, 9, 122, 87]. Arithmetic/logic operations run on GPU cores only and do not affect GPU memory or other GPU objects. However, an adversary may exploit control flow or memory access operations to break the confidentiality and authenticity of trusted display contents. Mediating each of these operations individually without hardware support would be prohibitive since it would significantly enlarge and add complexity to the GSK code base and hence diminish its security assurance. This would also add significant overhead to the OS's graphics performance.

To resolve the above access-mediation problem, this dissertation uses an efficient *address-space separation* mechanism. Instead of verifying individual instruction access, this mechanism confines the memory access of GPU instructions: it limits memory accesses only to those allowed by *local* GPU page tables. As a consequence, GPU address-space separation attacks no longer succeed since GPU instructions can no longer reference GPU memory via the shared GGTT. As the result, the mediation mechanism does not require any GPU instruction modification.

The mediation mechanism must also protect command buffers from modification by malicious GPU instructions and prevent TOCTTOU attacks. For example, some command buffers must be mapped into the local address space of untrusted OS/Apps in GPU memory. However, malicious GPU instructions can modify the GPU commands *after* command verification and invalidate the verification results for GPU commands at run time. Nevertheless, GPU address-space separation hardware can still protect the integrity of GPU command buffers via *write* protection. The confidentiality of command-buffer content

Table 6.1: GPU object and access separation.

| GPU Objects | Untrusted OS/Apps | SecApps |
|---|---|---|
| Data | Mediated sensitive access and direct insensitive access | Submitted via GPU separation kernel |
| Configuration Registers | | No access |
| Page Tables | | |
| Commands | | |
| Instructions | Confined by address space separation | |

does not need extra protection[6] and hence the accesses to GPU instructions that read command buffers need not be mediated.

### Access Mediation Policy

GPU access mediation policy comprises a set of “access invariants" that are enforced by the GPU separation kernel. These invariants are designed to ensure the security of the SecApps' display output and must hold at all intermediate points during trusted-display operation. They yield both secure-state invariants and transition constraints in a state-transition model of security policy [44].

**Access invariants.** As argued in Section 6.3.1, an adversary's attacks may either breach the confidentiality and authenticity of trusted display content (i.e., *content security*), or destroy the integrity of its configurations (i.e., *configuration integrity*). For example, the adversary can modify the configurations of both SecApps' display and sensitive GPU memory content. Hence, our access mediation policy is defined in terms of *invariants* for GPU object accesses that must be maintained for both content security and configuration integrity.

- *GPU data.* Content security requires the following invariants: (1) no untrusted *read* of the trusted display's *frame buffer*; and (2) no untrusted *write* to sensitive GPU data.

- *GPU page tables.* The following invariants must hold for GPU address space separation: (1) no untrusted OS/Apps can map sensitive GPU memory to be writable in any GPU local page tables; (2) no untrusted OS/Apps can map the trusted display's *frame buffer* to be readable in any GPU local page tables; (3) untrusted OS/ Apps must have a *single mapping* to sensitive GPU memory in GPU global address space; and (4) GPU instructions uploaded by untrusted OS/Apps cannot reference the GPU's global address space.

[6] Our adversary model omits side channels and inference analyses that may deduce sensitive output from command content.

- *GPU configuration registers.* Configuration integrity requires the following invariants: (1) no untrusted re-configuration of SecApps' display; and (2) no untrusted re-configuration of sensitive GPU memory. Content security requires the following invariant: no untrusted *read* of the trusted display's *frame buffer*, and no untrusted *write* to sensitive GPU memory.

  In addition, the invariant that untrusted access to configuration cannot violate the access invariants of GPU page tables must also be enforced.

- *GPU commands.* Content security requires the following invariants: (1) no GPU command can *read* trusted display's *frame buffers*; and (2) no GPU command can *write* sensitive GPU memory.

  In addition, the invariant that GPU commands cannot violate (1) any GPU configuration register invariants, and (2) GPU page table invariants must also be enforced.

**Soundness.** In order to show these invariants are sound, this thesis needs to show that if they hold, malicious access to and configuration of the above sensitive GPU objects is prevented. To show this, it should be noted that an adversary can perform only four types of attacks that represent combinations of two basic methods (unauthorized access to trusted-display content, and alteration of its configurations) exercised either directly or indirectly. To prevent direct manipulation of trusted-display content, GPU mediation policy ensures no untrusted access to the trusted display's *frame buffers* is possible. And to prevent indirect manipulation of trusted-display content, GPU mediation policy ensures that no untrusted write to other sensitive GPU memory is possible.

The protection of the trusted-display configuration protection is similar. To prevent direct tampering of output, untrusted re-configuration of trusted display is disallowed. Indirect re-configuration is prevented by requiring the same access invariants as those for indirect access to the display content. Finally, additional invariants are required to ensure that GPU address space separation is maintained in order to avoid complex GPU instructions verification.

#### Access Emulation

The GSK maintains full object-code compatibility with commodity platforms by retaining the common access methods (e.g., memory-mapped and port I/O) of commodity GPUs and by emulating the expected returns when untrusted commodity OS/Apps perform security-sensitive object operations. This thesis uses the security model to identify the minimal set of object accesses and functions that require emulation, viz., Section 6.4.5. This enables us to minimize the GSK code base; viz., Table 6.2 in Section 6.4.4, which shows that only a small number of mediated GPU objects require function emulation. For example, direct access to security-insensitive objects and security-sensitive access to objects used only by commodity OS/Apps (i.e., outside trusted display) do not need emulation. In contrast, full GPU virtualization has to emulate accesses to *all* GPU objects of the VMs scheduled to access the GPU. In particular, it has to emulate a wide variety of accesses to *all* GPU configuration registers[7] and thus requires a large trusted code base.

When object-access emulation is required by the security model, the GSK returns the expected access results as defined in the GPU specifications without actually accessing the real GPU objects. It does this by "shadowing" the real objects, viz., Section 6.4.5. For example, to locate OS's frame buffer, GSK emulates accesses to frame buffer base by accessing this register's shadow. Furthermore, for sensitive-object accesses that violate security invariants, the GSK simply drops *write* accesses and returns dummy values for *read* accesses[8].

### 6.3.5 Verifiable code base

The GSK code base is both small and simple, and hence verifiable, for the following three reasons. First, as shown in Section 6.5.1, the number of security-sensitive GPU objects is very small. Most of the GPU objects are security-insensitive, and can be directly accessed without kernel mediation.

Second, the GSK outsources most GPU functions (including all GPU functions used by commodity software and GPU objects provisioning for trusted display) to untrusted OS/Apps because it can verify all untrusted-code results very efficiently. The verification is driven by the policy invariants. Furthermore, only a small number of sensitive GPU objects require function emulation and this takes only a small amount of code, as shown in Section 6.5.1. Thus, implementing the GPU functions themselves (e.g., the large and complex native GPU drivers) within the GSK becomes unnecessary. The GSK also exports GPU driver code to SecApps using standard techniques [134]; that is, the traditional GPU software stack already de-privileges *frame buffer rendering* functions and management logic and exports them to user libraries. The GSK uses a similar approach, except that it requires SecApps to provide their own display contents. (Recall that SecApps cannot directly access any GPU objects.)

Third, GSK preserves existing assurance of the underlying trusted code bases. This is because GSK relies on existing security primitives and services already provided by the underlying trusted code bases, e.g., CPU physical memory access control [57, 11], and Direct Memory Access control [10, 56].

[7] Failure to emulate all accesses causes incompatibility with commodity OS/Apps; e.g., Tian *et al.* [111] virtualize GEN6_PCODE_MAILBOX register without emulating its functions, which causes GPU initialization errors in VMs.

[8] To date, this work has not found any malware-free commodity OS/Apps code that would be denied accesses to security-sensitive objects.

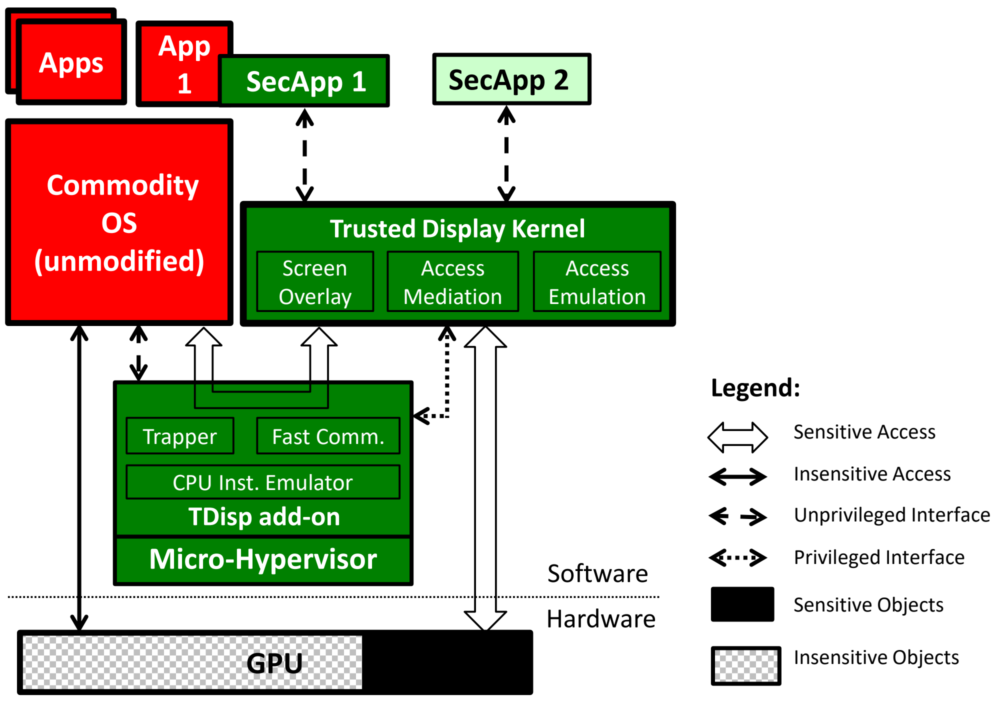


Figure 6.4: Architecture of the trusted display service.

## 6.4 Design and Implementation

This dissertation designs the GSK as an *add-on security* architecture [46] based on two components: a *Trusted Display Kernel (TDispK)* and a *trusted display (TDisp) add-on* to the underlying micro-hypervisor (*mHV*). This section first describes the system architecture, and then presents its detailed design for the trusted display.

### 6.4.1 Architecture Overview

As illustrated in Figure 6.4, mHV runs underneath all the other software components, protects itself, and hosts a TDisp add-on component. The TDisp add-on extends mHV and takes advantage of the mHV primitives to isolate its execution in a similar manner as used in past work [77, 134]. The TDisp add-on notifies Trusted Display Kernel (TDispK) about untrusted OS/Apps' requests to access sensitive GPU objects, since the TDispK is the only software component in the trusted-display service that is allowed to access these objects directly. The TDispK runs at the OS privilege level and provides trusted-display services to user-level SecApps that generate sensitive display content via CPU rendering. The TDispK also mediates accesses to sensitive GPU objects by native drivers of untrusted OS/Apps and emulates these accesses whenever necessary.

**TDispK.** The trusted display kernel (TDispK) includes three components. The first is the screen overlay component, which displays SecApps output over that of untrusted OS/Apps (Section 6.4.2).

The second component mediates access to all GPU sensitive objects that reveal or modify SecApps's overlaid display (Sections 6.4.3 and 6.4.4). The access mediation mechanism uses the CPU's protection ring 0 to prevent direct SecApps access to GPU objects and uses the privileged mHV interfaces to program the TDisp add-on to intercept sensitive accesses to GPU objects by untrusted OS/Apps.

The third component emulates access to security-sensitive objects to assure object-code compatibility with untrusted OS/Apps by shadowing sensitive GPU objects (Section 6.4.5). To emulate untrusted accesses, this component either configures CPU/GPU to operate on the shadow GPU objects or simulates object accesses.

**TDisp add-on.** The trusted display (TDisp) add-on supports the TDispK in the mediation of security-sensitive accesses to GPU objects by untrusted OS/Apps. It implements traditional hypercalls to receive TDispK-initiated communications, and fast communication channels to notify the TDispK of access requests received from untrusted OS/Apps. The hypercalls enable the TDispK to define the security-sensitive GPU objects so that the TDisp add-on knows what access requests from untrusted OS/Apps to trap. Once an access is trapped, the TDisp add-on uses its CPU instruction emulator to identify the object-access requested, access mode, and access parameters; e.g. the new value to be written. This information is sent to the TDispK for the mediation and emulation of the requested access to the object, via a fast communication channel.

The TDisp add-on is implemented using the TrustVisor [77] extension to XMHF, which isolates SecApps and enables them to request on-demand isolated I/O operations [134].

### 6.4.2 Screen Overlay

The screen overlay component of TDispK displays SecApps' output over that of untrusted commodity software. The screen overlay provides interfaces to SecApps and performs *frame buffer merging* in response to SecApps' requests. Frame buffer merging can be done either purely by software (i.e., by *"software overlay"*) or by hardware acceleration (i.e., by *"hardware overlay"*). In software overlays, TDispK shadows the screen frame buffer when TDispK is initialized. During the trusted-display session, the TDispK merges the SecApps' display contents with the untrusted *frame buffer* using their geometry information, and outputs the resulting image to the shadow frame buffer. Then, the TDispK programs the display engine to present the shadow *frame buffer* on the display. In comparison, hardware overlays are supported by some modern GPUs that layer one *frame buffer* over others. This improves the SecApps' CPU performance by eliminating frame-buffer merging by the CPU.

Frame buffer merging does not necessarily ensure that the SecApps display content is layered over all untrusted OS/Apps content. The hardware cursor can still be shown over the merged image, and hence TDispK must ensure that the hardware cursor operation is trustworthy. To do this, the TDispK provides and protects its cursor image and the corresponding GPU configurations[9]. Similarly, the TDispK emulates all hardware overlay not used by SecApps. Thus, SecApps can display over all untrusted contents.

SecApps also provide their display geometries to the TDispK to prevent SecApp-output overlaps. Furthermore, the TDispK provides a V-sync interrupt to SecApps to prevent image tearing. To enhance human perception of trusted screen areas, a SecApp always starts its display at the center of the screen. The current trusted-display implementation supports hardware overlays only, and allows a single SecApp to execute at a time, though a multi-window version implementation does not pose any additional security problems.

### 6.4.3 Access Mediation Mechanism

In general, the access mediation mechanism of TDispK interposes between commodity software and GPU by intercepting Memory-Mapped I/O (MMIO) and Port I/O (PIO), in a similar manner to that of previous systems [111, 108]. The access mediation mechanism also performs two tasks, namely GPU address space separation, and GPU command protection, as described in Section 6.3.4.

**GPU Address Space Separation.** GPU address space separation mediates instructions accesses by limiting their scope to GPU local page tables. Nevertheless, GGTT may also contain mappings to security-insensitive objects to be used by GPU instructions. To satisfy these requirements, the TDispK shadows GGTT to separate GPU address space as follows:

(1) TDispK shadows GGTT in a GPU local page table (GGTT′), and updates GGTT′ whenever GGTT is modified.

(2) TDispK verifies the access invariants for GGTT′.

(3) TDispK forces GPU instructions execution to use GGTT′ for all GPU engines except the display engine, which uses GGTT.

Note that in Step 3, the TDispK forces GPU instructions to use GGTT′ instead of GGTT in two steps. First, the TDispK wraps related GPU commands into a batch buffer. Second, it sets the batch buffer to use

[9] The cursor location needs protected only if the mouse device is a trusted-path device. This issue is orthogonal to the design of the trusted-display service.

GGTT′. As a result, GPU instructions preserve both their original functionality and security of the trusted display.

Forcing GPU instructions to use a new GPU local page table poses some implementation challenges. For example, it is possible that no spare slot exists to load GGTT′. Our solution is to randomly kick out a GPU local page table, switch to GGTT′ to execute the GPU instructions, and then switch back the original GPU local page table after the GPU instruction execution finishes. In principle, it is also possible that a single GPU command group uses all GPU page tables. Although this work has never encountered this situation in normal GPU driver operation [2, 6, 4], our TDispK splits the command group into smaller pieces and reuses the solution to the first challenge described above. The TDispK also pauses the verification of new GPU command submission in this case and resumes when this command group is executed.

**GPU Command Protection.** The TDispK also protects GPU command buffers from malicious GPU instructions. As GPU page tables support *read-write* access control in many modern GPUs (e.g. Nvidia GPUs [4], AMD GPUs [6] and recent Intel GPUs [54, 2]), the TDispK can protect GPU command buffers by mapping their *read-only* accesses to GPU local page tables. However, some Intel GPUs provide different hardware protection mechanisms. For example, GPU privilege protection disallows execution of security sensitive GPU commands from the batch buffers provided by applications. Using this feature, the TDispK can enforce the security invariants for GPU commands by de-privileging these commands from the batch-buffer accesses mapped in GPU local page tables.

### 6.4.4 Access Mediation Policy

The access mediation policy of the TDispK enforces the security invariants for sensitive GPU objects references, viz., Section 6.3.4. The TDispK identifies a GPU object by its address and verifies the validity of the object access by enforcing the corresponding invariants. The TDispK enhances the performance of GPU command mediation without sacrificing trusted-display security [111]. In particular, the TDispK monitors specific GPU configuration registers and performs the batch verification of an entire group of submitted GPU commands instead of trapping and mediating single GPU commands individually. Furthermore, the TDispK protects the verified GPU commands from malicious modification in TOCTTOU attacks, by shadowing the ring buffers and *write*- protecting batch buffers as illustrated by Tian *et al.* [111].

The access mediation policy has a small code base, since sensitive GPU objects comprise only a small group among all the GPU objects. As illustrated in Table 6.2, the TDispK needs to mediate accesses to only a few GPU objects of each type and all accesses for GPU global page tables. This saves TDispK from

Table 6.2: Trusted display kernel minimization. Legend: Bold letters denote object categories that contribute a significant number of GPU objects. (*) denotes categories where objects need mediation. The underline denotes mediated objects that do not require function emulation.

| Category | | Examples | Mediation | | Func. |
|---|---|---|---|---|---|
| | | | Virt. | TDK | Emu. |
| GPU Data | Display Engine Data | Shadow Framebuffer | Yes | Yes | Yes |
| | **Processing Engine Data** | **Other VM's GPU Data** | **Yes** | **No** | **No** |
| | Data of Other Engines | Performance Report | Yes | No | No |
| GPU Configu-ration | Config Registers Used By TDisp | *Presentation*: SecApps' Geometry, V-Sync Enable | Yes | Yes | Yes |
| | | *Data Path*: Framebuffer Bases, Target Display | Yes | Yes | Yes |
| | | *Others*: Ring Buffer Base | Yes | Yes | Yes |
| | Other Access Sensitive GPU Objects | Performance Report Buffer Base | Yes | Yes | No |
| | **Others** | | **Yes** | **No** | **No** |
| GPU Page Tables | Global Page Table | GGTT | Yes | Yes | Yes |
| | **Local Page Tables** | | **Yes** | **Yes** | **No** |
| GPU Comm-ands | Access TDisp Config Regs./ Page Tables | Update GGTT | Yes | Yes | Yes |
| | Other Access Sensitive GPU Objects | Batch Buffer Base | Yes | Yes | No |
| | **GPU Processing** | **3D Commands** | **Yes*** | **No** | **No** |
| | Others | | No | No | No |
| **GPU Instructions** | | | **Yes*** | **No** | **No** |

mediating accesses to all bolded types of objects except local page tables. The mediation of page-table access is simple due to their limited formats and sizes. In addition, special GPU protections built in hardware, such as GPU privilege protection, further reduces the mediation effort. It should also be noted that the TDispK forces GPU commands to use GGTT' instead of GGTT whenever possible to reduce the mediation overhead of GPU command accesses. Section 6.5.1 presents the TDispK code base minimization results.

Table 6.2 also shows that TDispK has to mediate access to far fewer GPU objects than full GPU virtualization [111, 108]. This is largely due to fine-grained object separation provided by our security model. For GPU data, full GPU virtualization must mediate all the accesses to other VMs' GPU data, whereas TDispK needs to mediate the accesses for only a small set of trusted-display sensitive GPU data and GPU configuration registers. In contrast, full virtualization approaches need to mediate all accesses to GPU configuration registers to isolate the GPU configurations among different VMs. Full GPU virtualization also requires mediation of accesses to GPU instructions by design, whereas the TDispK uses address space separation to avoid verifying individual GPU instructions. In addition, full GPU virtualization needs to mediate accesses to more GPU commands than the TDispK, due to the virtualization of the GPU processing engines. In Section 6.5.1, the thesis will provide a detailed evaluation on the mediation-policy trusted base.

### 6.4.5 Access Emulation

The TDispK emulates accesses to four categories of GPU objects (viz., Table 6.2) operating on the shadow objects (ephemeral objects) instead of the originals, as follows.

- For GPU data accesses, the TDispK allocates dummy memory with equal size of the sensitive GPU data, and invokes the TDisp add-on to remap the sensitive GPU data to the dummy memory for untrusted OS/Apps.
- For GPU configuration registers, the TDispK maintains shadow registers, and updates their values on untrusted accesses and chronology events (e.g. V-Sync) according to their function definitions. TDispK also keeps all the pending updates if the corresponding register requires *stateful* restoring during trusted-display finalization; e.g., register update relies on previous updates.
- For GPU page tables, the TDispK updates the shadow GGTT' whenever the original GGTT is updated.
- For GPU commands, the TDispK modifies their parameters or results to access shadow objects.

Table 6.2 shows that only a small number of mediated GPU objects require function emulation. The result is unsurprising since many sensitive GPU objects are not shared between untrusted OS/Apps and various SecApps. Section 6.5.1 shows the results of access emulation minimization.

### 6.4.6 TDisp add-on

**Trapper.** The trapper intercepts sensitive GPU object accesses by untrusted OS/Apps, as specified by the TDispK during the trusted display session. It use the CPU instruction emulator to obtain the trapped instruction's access information, e.g., the object being accessed, access mode, and access parameters such as a new value to be written. Then it notifies the TDispK with the access information and busy waits for TDispK's access decision. When TDispK returns the access decision, the trapper resumes the execution of the untrusted OS/Apps.

**CPU Instruction Emulator.** CPU instruction emulation is needed only for instructions that access security-sensitive GPU objects, and hence its complexity is lower than that of general purpose instruction emulators and disassemblers. For example, only a small subset of CPU instructions can be used to access GPU objects, e.g., instructions for MMIO and PIO. Also, the only instruction functions that need emulation are those operating in protected mode.

**Fast Communication Channel.** The fast communication channels facilitate communications initiated by the TDisp add-on with the TDispK on multi-core platforms. They employ shared memory to hold the communication data and use Inter-Processor Interrupts (IPI) for cross-core notification [11, 57, 134]. However, these channels differ from previous approaches in two ways. First, the communication initiator, namely the TDisp add-on, busy waits for TDispK's response after sending a request. Second, the communication receiver notifies *ready* responses via shared memory, instead of issuing IPI back to the TDisp add-on. Thus, these channels avoid expensive context switches between mHV and TDispK, and improve the system performance. More importantly, fast communication channels preserve the mHV's sequential execution and verified security properties [113], since the TDisp add-on neither receives or handles inter-processor interrupts.

### 6.4.7 Life-Cycle

As is the case with previous secure I/O kernel designs [134], the TDispK boots on-demand. This requires the mHV to isolate the execution of TDispK from commodity OS/Apps and the TDispK to isolate its memory from SecApp access.

**Initialization.** The TDispK configures the trusted display services when invoked by a SecApp. The untrusted OS/App provisions GPU objects (e.g. shadow frame buffer, V-Sync interrupt) and pins the related GPU memory in GPU global address space. Then the OS/App registers the configuration via an OS-hypervisor interface. After configuration, the TDisp add-on switches execution to the TDispK. The TDispK disables interrupts, pauses GPU command execution, and calls the TDisp add-on to specify and enable the interception of sensitive GPU objects accesses from untrusted OS/Apps. Next, the TDispK initializes GPU access emulation and verifies all GPU objects accesses according to the security invariants. Lastly, the TDispK configures shadow memories (e.g. shadow ring buffer, shadow frame buffer) to start

Table 6.3: Number of GPU objects requiring access mediation.

| GPU Object | Mediation in | | Total |
|---|---|---|---|
| | TDispK | Full GPU Virtualization [111] | |
| GPU Data | ~6 MB | All other VM's data | 2 GB |
| GPU Configuration Registers[10] | 39 | 711 | 625 |
| GPU Page Tables | All | | |
| GPU Commands | 21 | 43 | 269 |
| GPU Instructions | 0 | 14 | 66 |

the trusted display service, resumes GPU command execution, enables interrupts, and and returns to the SecApp. Trusted display initialization is not needed for a SecApp unless all previous SecApps that used the trusted display terminated. Finalization reverses these steps and zeros shadow memories without verifying GPU objects again.

**Untrusted Code Accesses to Sensitive GPU Objects.** The TDispK mediates the sensitive GPU object access using the access information received from the TDisp add-on, allows the access if the relevant security invariants are satisfied, and emulates the access if necessary, as described in Section 6.4.5. Then it returns the access decision to the TDisp add-on.

## 6.5 Evaluation

This thesis implements and evaluates our system prototype on an off-the-shelf HP2540P laptop, which is equipped with a dual-core Intel Core i5 M540 CPU running at 2.53 GHz, 4 GB memory and an integrated Intel 5$^{th}$ generation GPU (IronLake) with screen resolution of 1200 * 800. The laptop runs 32-bit Ubuntu 12.04 as the commodity OS with Linux kernel 3.2.0-36.57. This thesis implements a test SecApp that outputs a still image in all experiments.

### 6.5.1 Trusted Code Base

**Access Mediation Comparison.** The number of GPU objects that require access mediation by TDispK is much smaller than the number of GPU objects mediated in full GPU virtualization approaches [111]; viz., Table 6.3. This comparison is based on the Intel 7$^{th}$ generation GPUs (Haswell), which has an open-source driver (released by Intel) and detailed documentation. For GPU data, Haswell GPU maps a 2 GB GPU memory into the GGTT. Full GPU virtualization hosts the bulk of other VM's GPU data in the global

[10] The thesis counts registers using the same functional clustering as in Intel's documentation. This differs slightly from Tian *et al.*'s count [111], which lists registers individually.

Table 6.4: Code base size of trusted display service.

(a) Micro-hypervisor

| Modules | SLoC |
|---|---|
| XMHF[11] + TrustVisor | 28943 |
| CPU Instruction Emulator | 1090 |
| Fast Communication Channel | 144 |
| Trapper | 66 |
| **Total** | **30243** |

(b) GPU code in TDispK

| Modules | SLoC |
|---|---|
| Screen Overlay | 177 |
| Access Mediation | 2865 |
| Access Emulation | 1571 |
| Utility Code | 973 |
| **Total** | **5586** |

address space, whereas in our system the sensitive GPU memory is mapped in only about 6 MB. The memory used for sensitive GPU objects includes the shadow *framebuffers* (3750 KB for screens with 1200 * 800 resolution and 4 bytes per pixel), GGTT' (2052 KB), and other very small sensitive GPU memory areas, e.g., shadow ring buffers (128 KB). Note that the ratio of sensitive GPU objects to all GPU objects may vary, since the protection of multiple local GPU page tables requires more mediation of GPU data accesses and also increases the mapped memory space.

The TDispK has to mediate access to far fewer GPU configuration registers than full GPU virtualization: access to 39 out of 625 GPU configuration registers require mediation, 13 of which are needed for hardware overlays. In contrast, full GPU virtualization must mediate accesses to all GPU configuration registers to share all GPU functions securely among the different VMs that access the GPU. It also mediates access to more GPU commands than the TDispK since it needs to perform additional tasks such as the virtualization of the GPU 2D/3D processing engine. In addition, the TDispK does not need to mediate accesses of individual GPU instructions due to its use of the address-space separation mechanism.

**Access Emulation Minimization.** The TDispK has to emulate the functions of only 20 configuration registers and 12 GPU commands since the untrusted OS/Apps and various SecApps only share a subset of the sensitive GPU objects. As is the case with access mediation, full GPU virtualization needs to emulate the functions of *all* configuration registers to support all VMs that access the GPU, and more GPU commands than the TDispK to virtualize the GPU 2D/3D processing engine.

**Code Base Minimization.** The thesis uses the SLOCCount tool to measure the code base of our trusted-display service. As shown in Table 6.4a, the TDisp add-on modules (i.e., trapper, CPU instruction emulator, fast communication channel) add a total of 1300 SLOC, which is a minuscule number compared to existing code bases of similar functionality. For example, the CPU instruction emulator contributes

[11]XMHF with fine-grained DMA protection takes 24551 Source Lines of Code (SLoC) [134]

Table 6.5: Access mediation overhead of TDispK.

| GPU Configuration Registers | GPU Page Tables | GPU Commands |
|---|---|---|
| 2.61 $\mu$s | 2.69 $\mu$s | 8.86 $\mu$s |

most of this code, and yet it is an order of magnitude smaller than the code base of general purpose CPU instruction emulation tools; e.g., diStorm3.3[12] has 11141 SLOC. To implement this emulator we stripped the CPU instruction emulator of the Xen hypervisor (i.e., 4159 SLoC of Xen-4.5.0) of unnecessary code and shrank its size by 73.80% (from 4159 to 1090 SLOC). Were we to use diStorm3.3 instead, we would have bloated our micro-hypervisor's code size by over 38% (11141/(30,243-1,090)) and undoubtedly invalidated the existing formal assurances of XMHF.

Table 6.4b shows the code size of the TDispK. The access mediation code for the GPU uses most of the code base since it contains both the mediation mechanism (1741 SLoC) and policy (1124 SLoC). A large portion of the code in these modules (2227 SLoC) can be reused across different GPUs, including all the utility code and helper functions of other modules in the TDispK. In particular, supporting different GPUs of the same vendor only requires minimal code modification because the GPU object changes are incremental; e.g., IronLake specific code in TDispK takes only 178 SLoC.

In contrast, the code size of full GPU virtualization approaches [111, 108] is much larger. It contains a Xen hypervisor of 263K SLOC [134] and a privileged root domain that has over 10M SLOC.

### 6.5.2 Performance on Micro-Benchmarks

This section measures the performance overhead of commodity OS/Apps and SecApps during a trusted-display session. Specifically, this section measures the overhead of the access mediation and access emulation components of the TDispK to evaluate their impact on untrusted commodity OS/Apps. This section also evaluates the overhead added by screen overlays to SecApps. Finally, this section illustrates the overhead incurred by the TDisp add-on component of the trusted-display service.

**Access mediation.** The run-time overhead of access mediation is small, though its magnitude varies for different GPU objects. The mediation of access to GPU configuration registers, GPU page tables, and GPU data adds a modest performance penalty to commodity OS/Apps during SecApp run-time. As shown in Table 6.5, the TDispK spends 2.61 $\mu$s on mediating access to a GPU configuration register, and 2.69 $\mu$s on mediating access to a new GPU page table mapping, on average. However, the TDispK does not spend any time on GPU data mediation on the fly, because the sensitive GPU data has been remapped by TDisp

[12]https://code.google.com/p/distorm/

Table 6.6: Overhead of GPU address space separation.

| Initialization | Run-time | | |
|---|---|---|---|
| GGTT Shadowing | Modify GGTT′ | Apply Invariants | Page Table Switch[13] |
| 40.25 ms | 0.04 $\mu$s | 0.10 $\mu$s | 11.60 $\mu$s |

add-on. On average, access mediation for GPU commands takes 8.86 $\mu$s. It should be noted that GPU commands mediation overhead may vary; e.g. mediation of the GPU batch-buffer *start* command may require access verification for the entire GPU command batch buffer. However, in most cases, batch-buffer mediation by TDispK code is unnecessary since the GPU hardware protection mechanisms can be used instead.

The thesis also measured the performance of GPU address-space separation and GPU command protection since they are important components of the access-mediation mechanism. Table 6.6 shows the overhead of GPU address space separation added to untrusted OS/Apps. GGTT shadowing incurs the overwhelming portion of the overhead since it needs to parse every mapping in GGTT and construct the mapping in GGTT′. Fortunately, this overhead is incurred only at TDispK initialization, and subsequent SecApps display operations do not require GGTT shadowing. In contrast, the run-time overhead incurred by untrusted OS/Apps is small, as shown in Table 6.6. For GPU command protection, the TDispK implementation uses the GPU privilege protection mechanism and takes 0.07 $\mu$s on the average to de-privilege a single GPU batch buffer.

**Access emulation.** Similar to access mediation, TDispK's access emulation has a small runtime overhead, which varies for different GPU objects. For example, TDispK's overhead for emulating access to GPU page tables is 0.24 $\mu$s, on average. This thesis does not measure access emulation costs of GPU data, GPU commands, nor GPU configuration registers. The reason for this is that GPU data accesses are not intercepted by TDispK or TDisp add-on and their access emulation is done by memory remapping. Although the overhead of GPU command-access and configuration-register emulation varies widely with the types of objects and functions used, accesses to these objects are either infrequent or cause a small overhead.

**Screen overlay.** As anticipated, experiments confirm that hardware overlays have much better performance than software overlays. The test SecApp uses a 100 * 100 display area size with a screen resolution of 1200 * 800. Software overlays take 4.10 ms to process a display request of this SecApp. In contrast, hardware overlays take only 0.03 ms, which decreases the overhead of handling a display request in software

[13]This work evaluates Haswell GPUs instead, because Intel's open source GPU drivers support local page tables on GPUs newer than IronLake [2]

Table 6.7: TDisp add-on overhead.

| Trapper | CPU Instruction Emulator | Fast Communication Channel |
|---|---|---|
| 11.79 $\mu$s | 5.43 $\mu$s | 3.60 $\mu$s |

by 99.27%. Software overlays decrease CPU performance for SecApps as the screen resolution increases. For example, software overlays take 42.59 ms on displays with 4096 * 2160 resolution, which are becoming increasingly popular. Such an overhead would cause excessive frame rate drops at 60Hz or higher display refresh cycles, which would result in visually choppy display images. It should be noted that the overhead of software overlays increases when performed in GPU memory due to the reading and writing of *frame buffers* by the CPU.

**TDisp Add-on.** The TDisp add-on component takes 20.82 $\mu$s to trap and operate on a single sensitive GPU object access by untrusted OS/Apps. Table 6.7 illustrates the overhead breakdown.

First, the trapper takes 11.79 $\mu$s, on average, to intercept an MMIO access to GPU object and resume untrusted OS/Apps execution. (Note that this measurement does not include the PIO access interception.) This overhead is large due to the round-trip context switch between the mHV and untrusted OS/Apps. However, new hardware architectures make the trapper's overhead negligible.

Second, the CPU instruction emulator takes 5.43 $\mu$s, on the average, to parse the trapped CPU instruction due to accessing sensitive GPU objects from untrusted OS/Apps. However, this emulation overhead is well amortized since native GPU drivers [2, 6, 4] tend to use a single unified function for configuration updates of each type of GPU objects. Thus, the CPU instruction emulator can cache the emulation result for future use.

Third, our evaluation results show that the fast communication channels are much more efficient than switching between untrusted OS and TDispK. In our implementation, the fast channel takes 3.60 $\mu$s for the communication round-trip between mHV and TDispK when they run on different cores. In contrast, when the mHV needs to switch to TDispK and back on the same core, the overhead is 722.42 $\mu$s. This suggests that fast communication channels on multi-core systems can reduce the communication overhead by 99.50%.

### 6.5.3 Performance on Macro-Benchmarks

This thesis uses Linux GPU benchmarks to evaluate the performance impact of the trusted-display service on commodity OS/Apps software. Specifically, this thesis uses 3D benchmarks from Phoronix Test Suite [5], including OpenArena, UrbanTerror, and Nexuiz. The thesis also uses Cairo-perf-trace [1]

2D benchmarks, including firefox-scrolling ("Firefox"), gnome-system-monitor ("Gnome"), and midori-zoomed ("Midori"). However, the thesis did not run LightsMark in 3D benchmark or firefox-asteroids in 2D benchmark as in previous studies [111], because these workloads have not been available to us.

The evaluation uses three settings that are designed to distinguish the overhead caused by the underlying mHV from that caused by the trusted display service. These settings are: Ubuntu 12.04 with no security component added (“native"), mHV running *without* the trusted-display service ("TDisp_off"), and a SecApp using the trusted-display service running on the top of mHV ("TDisp_on"). The native setting does not load any of our trusted code, i.e., neither the mHV nor trusted-display service code. In the TDisp_off setting, both the mHV and TDisp add-on code are loaded, but the TDisp add-on code is never invoked because the TDispK is not loaded. Thus, whatever performance overhead arises in the TDisp_off setting is overwhelmingly caused by the security services of the *unoptimized* mHV [14]. The TDisp_on setting measures the overhead of the trusted-display service over and above that of mHV, and hence is of primary interest.

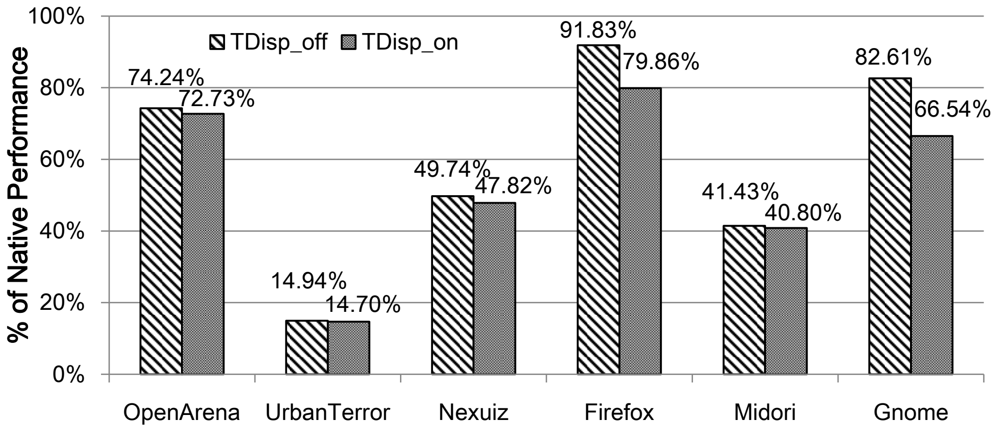
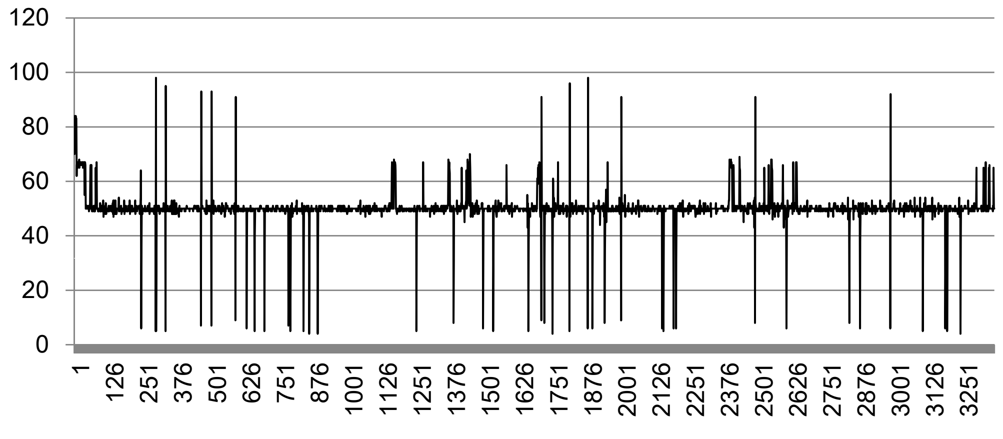


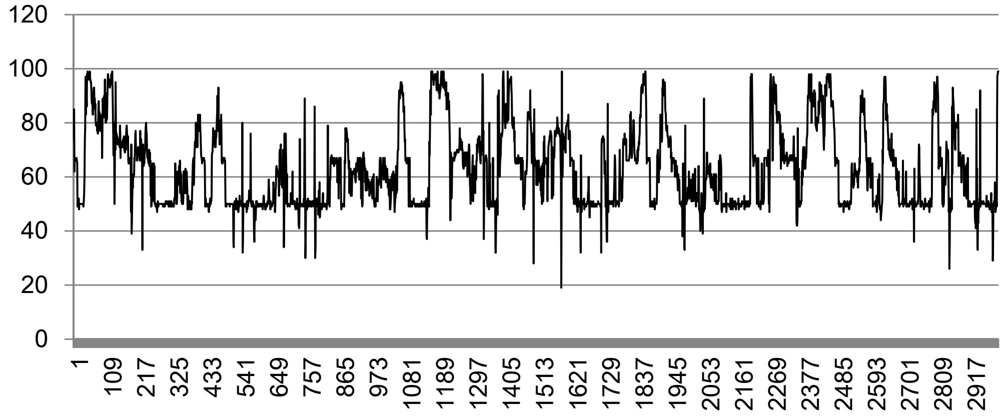
Figure 6.5: Performance of the trusted display service on 2D and 3D benchmarks.

Figure 6.5 shows that the TDisp_on setting achieves an average 53.74% of native performance, while TDisp_off achieves 59.13%. Thus, the trusted-display service is responsible for only 10% of the overhead whereas the *unoptimized* mHV is largely responsible for the rest. We believe that the performance of the mHV (i.e., XMHF) can be improved significantly with a modest engineering effort, e.g., mapping large pages instead of small pages in the nested page table for CPU physical memory access control, and hence decreasing the page-walking overhead and frequency of nested page table use. We also believe that new hardware architectures, such as Intel’s SGX, will make the hypervisor overhead negligible.

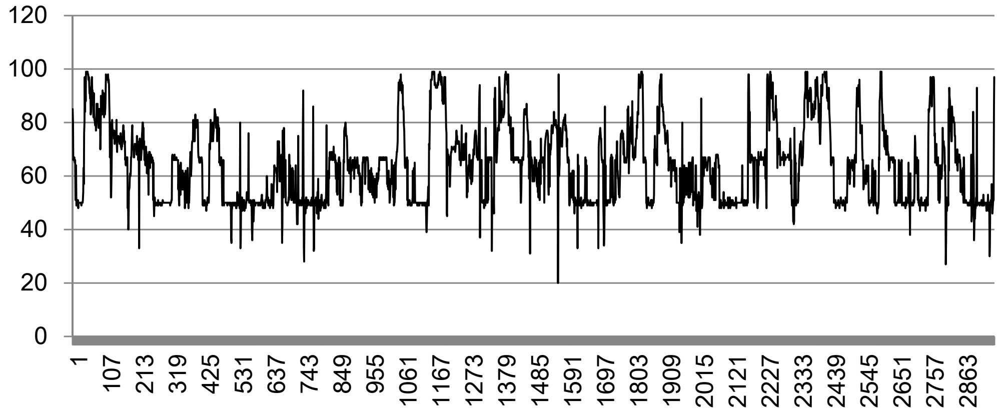
[14]For example, when isolating SecApps from untrusted OS/Apps, XMHF does not reduce page table walk overhead with pages larger than 4KB.



(a) Native



(b) TDisp_off



(c) TDisp_on

Figure 6.6: Latency evaluation of OpenArena. The vertical axis represents latency in milliseconds and the horizonal axis represents frame index.

Furthermore, the data of Figure 6.6 show that most frame jitter is caused by the unoptimized mHV, and the trusted-display service does not increase the amount of frame jitter. The thesis obtained these data by measuring the frame latencies of the OpenArena workload using the tools provided by the Phoronix Test Suite. These data show that the frame latencies of TDisp_on and TDisp_off settings are similar, whereas those of the native and TDisp_off settings are different. Specifically, the standard deviations are 6.62, 14.69, and 14.49 for Figure 6.6(a), Figure 6.6(b), and Figure 6.6(c), respectively.

## 6.6 Discussion

**Direct GPU Access by SecApps.** This chapter proposes a trusted display design that can be used by the vast majority of SecApps, and has few basic GPU requirements. Most SecApps can render their display content on the CPU and then submit it to the trusted display kernel for output. To support SecApps that require GPU commands and instructions, the use of either GPU pass-through mechanisms [115, 124] or GPUs that have added hardware isolation features [83] is suggested. Full GPU virtualization [111, 108] does not provide sufficient security for SecApps that require direct GPU access.

**Recommendations for GPU Hardware Enhancement.** Separating sensitive and insensitive GPU registers and memory into different aligned pages would help reduce the trap-and-mediate overhead in commodity OSes and improve OS's run-time performance. GPU hardware overlays [54] provide dedicated memory buffers inside the GPU for programs to enable independent rendering of images and videos on top of the main display screen. The system in this thesis – and other trusted display solutions – could leverage these features to further reduce the size of the trusted code base. The address-space separation attacks described in Section 6.2.2, which are enabled by inadvertent side-effects of a GPU optimization, should serve as a warning to GPU designers that serious security analyses are required when introducing new features that are intended to enhance performance in GPUs.

**Uniprocessor Support.** On uniprocessor platforms, the trusted display service pauses the untrusted OS/Apps when executing the SecApp. This simplifies the design of GPU isolation since SecApps cannot run concurrently with untrusted commodity software.

## 6.7 Vulnerabilities of the GPU Separation Kernel

Except separating ephemeral GPUs, the design of the GPU Separation Kernel is similar to the design of the Wimpy Kernel, i.e., SecApps exclusively own their ephemeral GPUs. Thus, the GPU Separation Kernel also enforces I/O separation between red and green partitions, and between green partitions. The red partition consists of commodity drivers, the ephemeral GPU used by them, and other devices. A green partition comprises SecApp's GPU drivers, their ephemeral GPU, and ephemeral objects. Ephemeral objects include ephemeral frame buffers associated with a SecApp, which are stored in the GPU Separation Kernel. The operations are similar to the ones in the Wimpy Kernel. The difference is that ephemeral GPUs are active in both the red and green partitions, whereas ephemeral USB host controllers are not active in both partitions in the Wimpy Kernel design. Because the GPU Separation Kernel relies on previous approaches [134] to provide separate I/O channels between SecApps and their ephemeral GPUs, the kernel has the risk of suffering from similar vulnerabilities as described in Chapter 5.

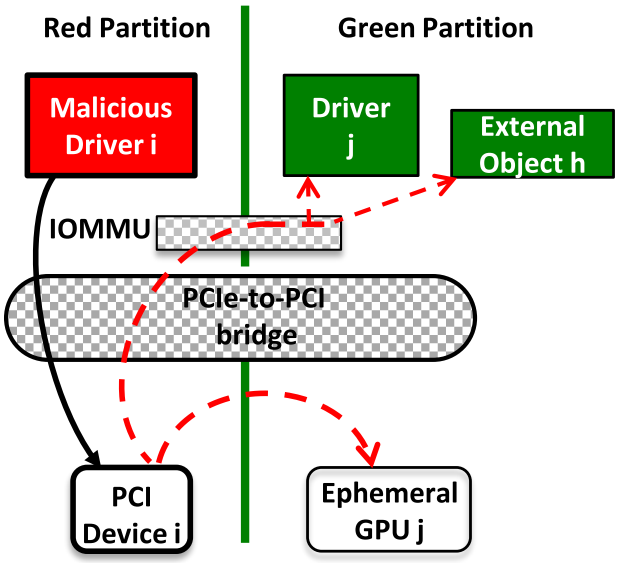


Figure 6.7: GPU Separation Kernel vulnerability 1.

*Vulnerability (same with vulnerability 1 of the Wimpy Kernel): A violation of red-green separation. Red devices can issue inadequately mediated transfers to green objects, i.e., to driver/device and external objects.*

The vulnerability shown in Figure 6.7 is the same as the one in Figure 5.2. If the GPU is a PCI device, then a malicious red driver *i* can misconfigure its PCI device *i*, such that the device can issue cross partition transfers to I/O objects in green via device peer-to-peer transfers and DMA accesses. To remove this vulnerability, the GPU Separation Kernel also needs to restrict the hardware configurations in order to prevent these transfers. For example, the GPU Separation Kernel must map ephemeral GPUs to GPUs on PCIe buses (not PCI buses), and enable IOMMU and PCIe ACS to mediate transfers by red devices.

**No similar vulnerability as Vulnerability 2 of the Wimpy Kernel.** This is because SecApps cannot write any transfer descriptors in green partitions. According to Section 6.3.3, SecApps only need to provide display content (which is the value of DO), and geometry configurations (which are values of FDs). Thus, green drivers cannot issue indirect accesses, i.e., cannot cause their ephemeral GPUs to initiate transfers to other green partitions.

Note that, by applying a formal interpretation/instantiation of the concrete model, one may discover more vulnerabilities in the GPU Separation Kernel design. In this thesis, the informal analysis of the GPU Separation Kernel design above demonstrates how a formally verified concrete model helps to reveal vulnerabilities in careful but informal GPU Separation Kernel design.

# Chapter 7

# Related Work

## 7.1 I/O Separation with Limited I/O Device Support

Some isolation kernels [89] and micro-hypervisors [23, 132, 133] support I/O separation for a limited set of I/O devices. NGSCB [89] isolates I/O for a very small collection of devices in order to provide an administrator console for system configuration. It cannot isolate general I/O devices, because it would have to include their drivers, and hence bloat the trusted code base. Guardian [23] provides a similar administrator console to establish trusted path between users and the Guardian micro-hypervisor.

Filyanov *et al.* [43] isolates a few user-centric I/O devices (i.e., keyboard, display and optionally a mouse) to build a trusted path between users and isolated applications using these devices. SUD [21] de-privileges existing commodity device drivers as untrusted user-level processes, by mediating I/O transfers by PCIe devices with commodity hardware security components. Thus, SUD cannot enforce I/O separation for devices on other buses, if hardware security components cannot mediate their transfers.

In constrast, the I/O separation model addresses *all* commodity devices and functions, and further reduces the system's trusted code base.

## 7.2 I/O Separation with Limited I/O Function Support

Separation kernels [96, 109] and some micro-kernels [121] provide I/O separation at the cost of restricting I/O functions that can be used by drivers and devices. Separation kernels [96, 109] isolate drivers and devices in different I/O partitions, which are defined prior to system boot. When the system runs, I/O partitions cannot be modified; e.g., the drivers and devices cannot move between partitions on demand.

Nexus RVM [121] implements reference monitors in the Nexus micro-kernel, and mediates I/O transfers by devices and de-priviledged drivers. Nexus RVM mediates drivers' accesses to TDs, and checks that direct transfers in TDs target only these drivers. This fails to prevent device peer-to-peer transfers, because malicious drivers can write TDs *indirectly*, and configure their devices to read these TDs; viz. the vulnerability 2 in Section 5.2. Furthermore, Nexus RVM fails to support hardware platforms that allow multiple-device broadcasts on a bus controller.

*seL4* [60] is a formally verified micro-kernel that allows drivers to run as user-mode applications and access their devices. The proof of I/O separation assumes devices cannot issue transfers, except interrupts [60, 32]. Consequently, *seL4* limits the I/O functions that can be used by drivers, and offers little guidance regarding I/O separation for commodity OSes.

DriverGuard [24] protects I/O communications between sensitive drivers in untrusted OS kernels and their devices. This approach must disable device peer-to-peer transfers when ACS or equivilant functions is unavailable, to prevent untrusted OS kernels from accessing sensitive I/O data in devices.

In contrast, the I/O separation model in this thesis can be used with a variety of commodity hardware architectures without limiting I/O hardware functions or restricting system configurations such as forcing zero or one device under each PCIe-to-PCI bridge.

## 7.3 Ephemeral Device Support

Device virtualization [125, 111] focuses on separate ephemeral commodity devices, and allows each virtual machine to access only their own ephemeral devices. This approach yields a much larger trusted code base than is suitable for micro-kernels or micro-hypervisors, and hence would lower their assurance significantly.

In contrast, the I/O separation model enforces security properties of I/O transfers across isolation boundaries and *assumes* the existence of secure ephemeral-device support in micro-kernel and micro-hypervisor based systems.

## 7.4 Hardware Supported Enclave Isolation

Intel Software Guard Extensions (SGX) uses trusted hardware to protect enclaves from any non-enclave memory accesses, including ones issued by untrusted OS, hypervisors and DMA devices [30]. Consequently, commodity I/O devices cannot encrypt/decrypt traffic with enclave secret keys and direct it to/from the encrypted enclave memory. Instead, they separate I/O transfers to isolated drivers, which then establish crypto channels with the enclaves [117].

SGX-USB [58] and Fidelius [38] provide protected I/O communicatons between enclaves and USB devices and display. These approaches build hardware dongles, which connect to USB devices [58, 38] and display [38] and communicate with enclaves via encryption. When providing protected I/O, hardware dongles encrypt I/O data received from peripheral devices and send them to enclaves, and decrypt I/O data received from enclaves and send them to peripheral devices.

Unfortunately, these hardware dongles cannot easily support general I/O devices. For the instance of PCIe devices, users have to unplug PCIe devices from computers, and then plug them in hardware dongles in order to allow enclaves to communicate with these PCIe devices. Similarily, the hardware dongles cannot support trusted path for laptops or tablets, where the peripheral devices cannot be unplugged easily.

## 7.5 Trusted Display And Related Attacks

### 7.5.1 GPU isolation

Several previous approaches provide trusted display services using security kernels. For example, Nitpicker [42], EWS [104] and Trusted X [37] support a trusted windowing system. Glider [100] could also be used to provide a trusted display service since it isolates GPU objects in the security kernel. However, these approaches are unsuitable for *unmodified* commodity OSes, because security kernels are object-code incompatible with native commodity OSes. Past research efforts to restructure commodity OSes to support high-assurance security kernels have failed to meet stringent marketplace requirements of timely availability and maintenance [70, 46]. In contrast, our system does not require any modification of widely available commodity OSes.

Other approaches provide trusted display by exclusively assigning GPU to SecApps. Recent work [115, 124] uses the device pass-through feature of modern chipsets [10, 56] for this assignment. Other work [75, 22] isolates the GPU with a system's TCB. Recent implementations of trusted path [132, 134] also isolate communication channels from SecApps to GPU hardware. However, once assigned to a SecApp, the GPU cannot be accessed by untrusted commodity OS/App code until the device is re-assigned to that code. Thus, a commodity OS/App cannot display its content during a SecApp's exclusive use of the trusted display, unless the OS/App trusts SecApps unconditionally. Our system solves this problem by allowing untrusted OS/Apps and SecApps to use GPU display function *at the same time*. As a result, commodity OS/Apps do not need to rely on external SecApps for display services.

### 7.5.2 GPU virtualization

GPU virtualization can provide trusted-display services by running SecApps in a privileged domain and untrusted OS/Apps in an unprivileged domain. The privileged domain can emulate the GPU display function in software [106, 64] for the untrusted OS/Apps. However, other GPU functions, such as image-processing emulation, are extremely difficult to implement in software and take advantage of this setup due to their inherent complexity [111, 34]. As a result, GPU emulation cannot provide all GPU functions to the untrusted OS/Apps, and hence this approach is incompatible with commodity software. Smowton [105] paravirtualizes the user-level graphics software stack to provide added GPU functions to untrusted OS/Apps. Unfortunately, this type of approach requires graphics software stack modification inside untrusted OS/Apps, and hence is incompatible with commodity OS software.

Full GPU virtualization approaches [111, 108] expose all GPU objects to unprivileged VMs access, and hence allow untrusted OS/Apps to use unmodified GPU drivers. However, these approaches share all GPU functions between privileged domain and unprivileged domain, and hence require complex mediation and provide only low assurance. Existing full GPU virtualization approaches are subject to GPU address space isolation attacks, and hence are inadequate for trusted-display services. Furthermore, full GPU virtualization requires extensive emulation of accesses to a large number of GPU objects, in order to retain compatibility with the VMs that share the GPU. Our system solves the problem by sharing only the GPU display function between the untrusted OS/Apps and the SecApps. Thus, it needs to mediate only accesses to GPU objects that affect trusted display's security. Hence it needs to emulate accesses to a much smaller set of GPU objects, which helps minimize the trusted code base for high assurance system development.

### 7.5.3 Special devices

High-bandwidth digital content protection (HDCP) [71, 95] employs cryptographic methods to protect display content transmitted from the GPU to a physical monitor. HDCP requires encryption/decryption circuits, and hence hardware modification of both GPUs and physical monitors. Similarly, Hoekstra *et al.* [50] also require crypto support in GPU to provide trusted display. Intel Identity Protection Technology with Protection Transaction Display is also reported to rely on special CPU and GPU features [53]. In contrast, our system does not require any modification of existing commodity hardware. It could also use HDCP to defend against certain hardware attacks, e.g., malicious physical monitors.

Other cryptography-based approaches [127, 82] decode concealed display images via optical methods, e.g., by placing a transparency, which serves as the secret key, over concealed images to decode them. These approaches are similar in spirit to the use of *one time pads*, and hence need physical monitor modification for efficient, frequent re-keying. Other systems [86] add decryption circuitry to displays, and hence also require commodity hardware modification, which fails to satisfy our design goals.

### 7.5.4 Related GPU Data Leakage Attacks

Recent research [33, 65] shows that GPU data leakage may occur because security-sensitive applications may not zero their freed GPU data memory in a timely manner. Thus, malicious applications can probe GPU data of other applications by using native GPU drivers (including user-level libraries) on commodity GPUs. Other recent work [76] also focuses on GPU data leakage in virtualized environments and shows that the untrusted OS can manipulate GPU configuration registers to leak GPU data. An early instance of the data leakage problem [111] shows that full GPU virtualization approaches could solve the problem in principle by mediating access to GPU data, configuration registers, commands and page tables. Nevertheless, the GPU address space separation attack of Section 6.2.2 shows that malicious GPU instructions can still break GPU data confidentiality and authenticity of full GPU virtualization. Our system prevents these attacks since the GSK mediates accesses to sensitive GPU objects and emulates unsafe accesses.

Chapter 8

# Discussion and Future Work

This chapter discusses I/O hardware modifications that could reduce complexity of I/O kernels and increase their performance. Then this chapter discusses the limitation of the I/O separation model and the future work in composing multiple I/O kernels to protect I/O between SecApps and various devices.

## 8.1 Informal Separation Model Interpretation in Other I/O Kernels

Informally, I/O partitions of traditional separation kernels [97, 109, 48] differ from the I/O kernels of our concrete model, and thus require a new concrete model; i.e., a new interpretation of the I/O separation model. First, these I/O partitions are separation-kernel partitions, and thus there is no distinction between red and green I/O partitions. Second, the new concrete model leverages the unidirectional communication channels provided by separation kernels to mediate transfers issued by drivers, which are loaded into isolated applications partitions prior to system boot, to I/O kernels located in different I/O partitions. Third, the allocation of devices to these I/O kernels cannot be modified on demand; e.g., it is static, as devices cannot move between partitions on demand. Nevertheless, these I/O kernels can support DMA accesses that are excluded by traditional I/O partitions of separation kernels [96, 97].

The I/O separation model can also be instantiated in low-assurance kernels of commodity OSes, such as the Linux kernel, via a new concrete model. To prevent device access to non-DMA memory, the new concrete model has two I/O partitions: one comprises all active subjects and objects, and the other is the NULL I/O partition for inactive ones. Thus, active devices cannot access non-DMA memory areas because these areas comprise non-I/O objects; i.e., inactive I/O objects that are unmapped in DMA memory. Unlike previous I/O separation designs [74], the new concrete-model instantiation in OS kernels would support device P2P communications.

## 8.2 Suggested I/O Hardware Modifications

Commodity hardware can reduce the complexity of I/O kernels and improve their performance by providing more hardware mediation support of I/O transfers. For example, commodity hardware can implement selective mediation of transfers in IOMMUs, and mediation of device peer-to-peer accesses in all I/O buses that support this feature. Thus, I/O kernels can employ these hardware mediation mechanisms and enforce I/O separation by computing transitive closures of smaller sets of TDs, i.e., for TDs that define I/O transfers which cannot be mediated by hardware, instead of all active TDs.

Note that hardware mediation mechanisms cannot easily separate I/O for ephemeral devices, even after modification. First, different SecApps require different device functions, and hence require physical devices to be separated differently. Second, ephemeral devices mapped to the same physical device could access each other via the internal fabric of the physical device, instead of issuing transfers via I/O buses. Consequently, these transfers bypass all hardware components that mediate I/O transfers on buses.

## 8.3 Limitations of the I/O Separation Model

The I/O separation model in this thesis is not specified in a modular way, which hinders one from straight forwardly establishing new models by replacing I/O separation properties with other security properties. Specifically, a modular specification allows new models to mediate devices' access without relying on any commodity I/O hardware architecture by computing transitive closures of active TDs' states. For example, a modified model could fulfill multi-level secrecy properties, such that devices are subjects instead of objects as in the BLP model [20].

## 8.4 Composition of I/O Kernels

As described in Chapter 5 and Chapter 6, the Wimpy Kernel provides separate I/O between SecApps and their USB devices, and the GPU Separation Kernel provides separate I/O between SecApps and their display. More I/O kernels could be introduced to support SecApps with additional devices. Consequently, a question arises: *How may each I/O kernel be composed with the others?* The answer to this question helps to reduce the trusted code base of each I/O kernel, because an I/O kernel would not need to trust others to secure SecApps' communications with its supported devices. This question remains open and its solution is left as future work.

# Chapter 9

# Lessons Learned

Formal I/O separation models did not exist until now. Several practical lessons arise from the use of such a model in the formal verification of a real I/O kernel design, which are summarized below.

First, the model requirement for *complete mediation* of I/O transfers illustrates the fundamental drawback of inadequate I/O hardware use (e.g., denied legitimate transfers), independent of performance considerations which arise from transitive closure computation. This suggests that hardware such as PCI, PCIe-PCI bridges, and similarly inadequate devices (Figure 2.1) should be deprecated for secure I/O use, and ARM TrustZone and similar techniques must mediate transfers at the granularity of devices instead of worlds, to support I/O separation between green partitions. Even with the best hardware mediation support, OS designers encourage performant applications [12] to disable PCIe ACS and IOMMU to improve device P2P access performance [126, 107]. For example, Xilinx [126] notes that routing PCIe P2P transfer through IOMMU significantly degrades access performance. Once disabled IOMMU or related configuration in ACS, hardware is limited in mediating device P2P transfers.

Second, formal modeling enforces rigorous reasoning. Even for small I/O kernels (e.g., fewer than 4K SloC) that were carefully but informally designed, the soundness of these designs is difficult to check without formal specifications of I/O hardware separation properties. Lack of such specifications leads to subtle vulnerabilities (viz., Section 5.2) that can be easily missed by informal designs, even when carefully created.

Third, the performance of formal I/O separation need not incur any penalty beyond that of secure IOMMU context switches in existing OS kernels [74]. Just the opposite: the use of *non-shared*, de-privileged drivers and buffers in isolated applications, which don't rely on *shared* buffer pools in OS kernels[1], *and* of small and simple dedicated I/O kernels, naturally offer added performance benefits [134].

[1] Substantial performance improvements that preserve intra-OS kernel I/O separation by *selective mediation* can still incur up to 25% throughput overhead and 20% increased CPU utilization by using shared buffer pools [74].

Chapter 10

# Conclusion

General I/O separation models that enable isolated applications running on untrusted commodity OSes to protect their I/O operations do not exist to date. Such models must include a precise separation policy based on complete mediation despite lack of comprehensive hardware support on commodity computer systems. In particular, these models must enable proofs of correctness for I/O designs where drivers are malicious (e.g., contain malware) and can manipulate non-malicious devices to breach I/O separation.

This dissertation showed that such models require new transfer-mediation constructs that are both sound and practical; e.g., they must apply on *any* commodity I/O hardware configuration regardless of its ability, or lack thereof, to mediate I/O transfers. Of particular interest is the concrete application of the model presented herein for I/O kernels that separate channels for untrusted OSes and isolated applications on-demand. This dissertation also showed that existing carefully but informally designed on-demand I/O kernels (i.e., the Wimpy Kernel and the GPU Separation Kernel) still include heretofore unknown design vulnerabilities that violate application isolation. Removal of these vulnerabilities becomes evident in the model-based analysis presented in this thesis, and resulting re-designs can become provably secure in the Wimpy Kernel.

# Appendix A

# Example Dafny Specification of I/O Separation Model

This section briefly describes the formal specifications of the I/O separation model in Dafny. With formal verification tools such as Dafny, we can specify models and properties and then invoke Dafny to automatically verify that the models satisfy the properties. The full specification of the I/O separation model, the concrete model, and the sound design of Wimpy Kernel can be downloaded at https://github.com/superymk/iosep_proof. The functions and files described in this chapter are mainly for the I/O sepration model, and a similar schema applies to the concrete model and the sound design of Wimpy Kernel as well.

**Simplified Specifications of Operations.** Figure A.1 presents a simplified Dafny specification of the Device Write operation in the file MODEL.DFY. The operation takes the current state $k$, the ID of the device issuing the write transfer, and the IDs of objects to be modified, together with the new values to be written. The operation returns the resulting state $k'$, as well as a boolean value $d$ to indicate whether the operation is allowed or denied. Then the specification presents the preconditions and postconditions of the operation. The operation requires the current state $k$ to fulfill all the state invariants. The device must be active, and the write transfers must be defined in TDs. After the operation returns, it ensures that the result state $k'$ fulfills all the state invariants, and the operation fulfills all transition properties. The body of the operation specifies the operation implementation first, then proves the implementation against all specified postconditions of the operation given the preconditions. All other operations are formally specified under the same schema.

**State Transition.** The model calls K_CALCNEWSTATE function in SECURITYPROPERTIES.DFY to apply a single transition; that is, an operation on the state $k$. As shown in Figure A.2, the function takes a state

```
method DevWrite(
    k: State,
    dev_sid: Subject_ID,
        // ID of the device issues the write access
    td_id_val_map: map<TD_ID, TD_Val>,
        // IDs of TDs, and values to be written
    fd_id_val_map: map<FD_ID, FD_Val>,
        // IDs of FDs, and values to be written
    do_id_val_map: map<DO_ID, DO_Val>
        // IDs of DOs, and values to be written
) returns (k': State, d: bool)
        // Return k' as new state, d as allow/deny decision
    requires IsValidState(k) ∧ IsSecureState(k)
        // Requirement: k fulfills all SIs
    requires Dev_ID(dev_sid) in k.subjects.devs
    requires SubjPID(k, dev_sid) ≠ NULL
        // Requirement: Device is in state and is active

    requires ∀ td_id2 • td_id2 in td_id_val_map
        ⟹ DevWrite_WriteTDWithValMustBeInATransfer(
                k, dev_sid, td_id2, td_id_val_map[td_id2])
    requires ∀ fd_id2 • fd_id2 in fd_id_val_map
        ⟹ DevWrite_WriteFDWithValMustBeInATransfer(
                k, dev_sid, fd_id2, fd_id_val_map[fd_id2])
    requires ∀ do_id2 • do_id2 in do_id_val_map
        ⟹ DevWrite_WriteDOWithValMustBeInATransfer(
                k, dev_sid, do_id2, do_id_val_map[do_id2])
        // Requirement: Issued transfers must be defined in
        // TDs first

    ensures IsValidState(k') ∧ IsSecureState(k')
        // Property: k' fulfills all SIs
    ensures IsSecureOps(k, k')
        // Property: DevWrite fulfills all TCs defined in
        // IsSecureOps

    ensures (∀ td_id • td_id in td_id_val_map
        ⟹ td_id in k.objects.tds) ∧
            (∀ fd_id • fd_id in fd_id_val_map
        ⟹ fd_id in k.objects.fds) ∧
            (∀ do_id • do_id in do_id_val_map
        ⟹ do_id in k.objects.dos)
        // Property: Written objects are in the I/O state
    ensures SubjWrite_ObjsToWriteMustHaveSamePIDWithSubj(
                k, dev_sid, td_id_val_map, fd_id_val_map,
                do_id_val_map)
        // Property: All written objects must be in the
        // same partition with the device

    ....
    // Additional proved properties, e.g., the operation
    // always returns true
{
    // Operation implementation
    var tds' := WriteTDsVals(k.objects.tds, td_id_val_map);
    var fds' := WriteFDsVals(k.objects.fds, fd_id_val_map);
    var dos' := WriteDOsVals(k.objects.dos, do_id_val_map);

    var k'_subjects := k.subjects;
    var k'_objects := Objects(tds', fds', dos');

    k' := State(k'_subjects, k'_objects, k.pids);
    d := true;

    // Proof of operation properties
    ...
}
```

Figure A.1: Simplified specification of Device Write operation in Dafny

$k$, and the operation name $op$ corresponding to the transition taking place. After the transition is done, it returns the result state and the boolean decision value $d$. The function requires state $k$ to be secure, i.e., fulfilling all state invariants, because all operations on a secure state transition only to another secure state. The function further requires operations' preconditions to always imply operations' postconditions, and that the preconditions hold for the given operation $op$. The first statement is always true according to the specifications of operations, and the second statement is also always true, because operations can only take place after their preconditions are met.

The body of the K_CALCNEWSTATE function returns an arbitrary state fulfilling the postconditions of operation $op$. Note that this function does not compute the result state $k'$ by applying the implementations of operations. The operation correctness properties are not used in proving theorems 1 and 2 and Corollary 1, as long as the result state $k'$ fulfills the postconditions of the operations. The K_CALCNEWSTATE function is defined as a Dafny function to enable its use in the proof of the theorems and the corollary.

**High Level Properties and Theorems.** The security properties SP1, SP2 and Theorems 1, 2 are formally specified and verified at the beginning of the file SECURITYPROPERTIES.DFY.

```
function K_CalcNewState(k: State, op:Op) : (result: (State, bool))
    requires IsValidState(k) ∧ IsSecureState(k)
        // Requirement: k fulfills all SIs
    requires P_OpsProperties(k, op)
        // Requirement: For operation <op>, its
        // preconditions always imply its postconditions
    requires P_OpsFulfillPreConditions(k, op)
        // Requirement: If operation <op> takes place,
        // then it must fulfill all its preconditions on
        // the current state k

    ensures IsValidState(result.0) ∧
        IsSecureState(result.0)
        // Property: The result state fulfills all SIs
        // result.0 is the result state
    ensures IsSecureOps(k, result.0)
        // Property: The operation <op> fulfills all TCs
        // defined in IsSecureOps
{
    if(op.DrvReadOp?) then
        var k', d :| DrvRead_PostConditions(k, op.drv_sid,
            op.read_objs, op.tds_dst_src, op.fds_dst_src,
            op.dos_dst_src, k', d); (k', d)
    else if (op.DevReadOp?) then
        var k', d :| DevRead_PostConditions(k, op.dev_sid,
            op.read_objs, op.tds_dst_src, op.fds_dst_src,
            op.dos_dst_src, k', d); (k', d)
    else if (op.DevWriteOp?) then
        var k', d :| DevWrite_PostConditions(k,
            op.dev_sid, op.td_id_val_map, op.fd_id_val_map,
            op.do_id_val_map, k', d); (k', d)
    else if (op.EmptyPartitionCreateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.EmptyPartitionDestroyOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.DrvActivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.DrvDeactivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.DevActivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.DevDeactivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.ExternalObjsActivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else if (op.ExternalObjsDeactivateOp?) then
        var k', d :| Common_PostConditions(k, k', d);
        (k', d)
    else
        var k', d :| DrvWrite_PostConditions(k, op.drv_sid,
            op.td_id_val_map, op.fd_id_val_map,
            op.do_id_val_map, k', d); (k', d)
}
```

Figure A.2: K_CalcNewState for applying individual state transition

# Bibliography

[1] Cairo-perf-trace. http://www.cairographics.org/.

[2] Intel graphics driver. https://01.org/linuxgraphics/.

[3] Lima graphics driver for ARM Mali GPUs. http://limadriver.org/.

[4] Nouveau graphics driver. http://nouveau.freedesktop.org/.

[5] Phoronix Test Suite. http://www.phoronix-test-suite.com/.

[6] Radeon graphics driver. http://www.x.org/wiki/radeon/.

[7] The Linux Documentation Project. http://www.tldp.org/HOWTO/Plug-and-Play-HOWTO-7.html [Accessed on Jun. 20, 2019], 2007.

[8] D. 5200.28-STD. *Trusted Computer System Evaluation Criteria*. Dod Computer Security Center, December 1985.

[9] AMD. AMD Radeon documentation. http://www.x.org/wiki/RadeonFeature/#index10h2.

[10] AMD. AMD I/O virtualization technology (IOMMU) specification. AMD Pub. no. 34434 rev. 1.26, 2009.

[11] AMD. AMD 64 Architecture Programmer's Manual: Volume 2: System Programming. Pub. no. 24593 rev. 3.23, 2013.

[12] AMD. Accelerating nvm express storage with eideticom's noload, amd epyc cpus, and p2pdma. https://www.amd.com/system/files/documents/accelerating_nvm_express_storage_epyc_eideticom.pdf, 2019.

[13] Android. Trusty TEE, 2019.

[14] S. Anthony. Massive, undetectable security flaw in USB: It's time to get your PS/2 keyboard out of the cupboard. *Extreme Tech*, (July 31), 2014.

[15] J. Applebaum, J. Horchert, and C. Stocker. In *Catalog Reveals NSA Has Back Doors for Numerous Devices*, volume Dec. 29. Springer Online, 2013.

[16] ARM. Open source Mali-200/300/400/450 GPU kernel device drivers. `http://malideveloper.arm.com/develop-for-mali/drivers/open-source-mali-gpus-linux-kernel-device-drivers`.

[17] ARM. ARM AMBA 5 AHB Protocol Specification, 2015.

[18] BAE Systems Information Technology LLC. Security Target, Version 1.11 for XTS-400, Version 6, 2004.

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. ACM Symposium on Operating Systems Principles*, 2003.

[20] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation, 1976.

[21] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *Proc. USENIX Annual Technical Conference*, 2010.

[22] Y. Cheng and X. Ding. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 201–218, Berlin, Heidelberg, 2012. Springer-Verlag.

[23] Y. Cheng and X. Ding. Guardian: Hypervisor as security foothold for personal computers. In *Proc. International Conference on Trust and Trustworthy Computing*. 2013.

[24] Y. Cheng, X. Ding, and R. H. Deng. Driverguard: A fine-grained protection on i/o flows. In V. Atluri and C. Diaz, editors, *Computer Security – ESORICS 2011*, pages 227–244, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[26] D. D. Clark. The end-to-end argument and application design: the role of trust. *Fed. Comm. LJ*, 63:357, 2010.

[27] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, 2011.

[28] L. Constantin. What is a “Supply Chain Attack?". In *Motherboard*, Sept. 2017.

[29] M. Corporation. Cve details: Linux kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 [Accessed on 10 Jun. 2019].

[30] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[31] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[32] Data61/CSIRO. Frequently asked questions on sel4. https://docs.sel4.systems/FrequentlyAskedQuestions.html, 2019.

[33] R. Di Pietro, F. Lombardi, and A. Villani. CUDA leaks: information leakage in GPU architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[34] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009.

[35] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM Symposium on Operating Systems Principles*, 1995.

[36] J. Epstein, C. Inc, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Danner, C. R. Martin, M. Branstad, G. Benson, and D. Rothnie. A high assurance window system prototype. *Journal of Computer Security*, 2(2):159–190, 1993.

[37] J. Epstein, J. McHugh, R. Pascale, H. Orman, G. Benson, C. Martin, A. Marmor-Squires, B. Danner, and M. Branstad. A prototype b3 trusted x window system. In *Computer Security Applications Conference, 1991. Proceedings., Seventh Annual*, pages 44–55, Dec 1991.

[38] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. G. Jr., E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. *CoRR*, abs/1809.04774, 2018.

[39] EuroCPS. Quark system on chip platform. https://www.eurocps.org/eurocps-platforms/quark-intel/, [Fetched on 2019-09-18].

[40] K. Fatahalian and M. Houston. A closer look at GPUs. *Commun. ACM*, 51(10):50–57, Oct. 2008.

[41] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305, 2017.

[42] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *Proc. Annual Computer Security Applications Conference*, 2005.

[43] A. Filyanov, J. M. McCune, A.-R. Sadeghi, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *Proc. IEEE/IFIP Conference on Dependable Systems and Networks*, 2011.

[44] M. Gasser. Building a secure computer system. In *Van Nostrand Reinhold, New York*, 1988.

[45] V. Gligor and M. Woo. Establishing software root of trust unconditionally. In *Proc. NDSS*, 2019.

[46] V. D. Gligor. Security limitations of virtualization and how to overcome them. In *Proc. International Workshop on Security Protocols, Cambridge University*, 2010.

[47] V. D. Gligor, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W.-D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan. Design and implementation of secure Xenix. *IEEE Transactions on Software Engineering*, 13(2):208–221, 1986.

[48] I. GreenHills Software. Integrity-178b separation kernel security target. http://www.niap-ccevs.org/st/st_vid10362-st.pdf [Accessed on 7 Nov 2013], 2010.

[49] B. Hoanca and K. J. Mock. Screen oriented technique for reducing the incidence of shoulder surfing. In *Security and Management*, pages 334–340, 2005.

[50] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[51] Huawei Technologies CO., LTD. EMUI 8.0 Security Technical White Paper, 2017.

[52] B. D. S. Inc. Open hub: Linux kernel. https://www.openhub.net/p/linux [Accessed on 10 Jun. 2019].

[53] Intel. Deeper levels of security with intel(r) identity protection technology. http://ipt.intel.com/Libraries/Documents/Deeper_Levels_of_Security_with_Intel%C2%AE_Identity_Protection_Technology.pdf.

[54] Intel. Intel processor graphics programmer's reference manual. https://01.org/linuxgraphics/documentation/driver-documentation-prms.

[55] Intel. Enhanced Host Controller Interface Specification for Universal Serial Bus, 2002.

[56] Intel. Intel virtualization technology for directed I/O architecture specification. Intel Pub. no. D51397-006 rev. 2.2, 2013.

[57] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual: Volume 3: System programming guide. Pub. no. 253668-048US, 2013.

[58] Y. J. Jang. *Building trust in the user I/O in computer systems*. PhD thesis, Georgia Tech, 2017.

[59] S. Kato. Implementing open-source CUDA runtime. In *Proc. of the 54the Programming Symposium*, 2013.

[60] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proc. of the 22nd ACM SOSP*, pages 207–220, 2009.

[61] N. Knupffer. Intel Insider – What Is It? (IS it DRM? And yes it delivers top quality movies to your PC). http://blogs.intel.com/technology/2011/01/intel_insider_-_what_is_it_no/[Accessed on 30 Oct 2013].

[62] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, 2014.

[63] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462. IEEE Computer Society, 2010.

[64] I. T. Lab. Qubes OS. https://qubes-os.org/.

[65] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 19–33. IEEE Computer Society, 2014.

[66] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[67] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-based attestation for peripherals. In *Proc. of the 3rd Int. Conf. on Trust and Trustworthy Computing*, volume 6101 of *LNCS*, pages 16–29. Springer, 2010.

[68] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the Integrity of PERipherals' firmware. In *Proc. of the 18th ACM CCS, Chicago, IL*, pages 3–16, 2011.

[69] Linaro Limited. OP-TEE, 2019.

[70] S. Lipner, T. Jaeger, and M. E. Zurko. Lessons from VAX/SVS for high assurance VM systems. *IEEE Security and Privacy*, 10(6):26–35, 2012.

[71] D. C. P. LLC. High-bandwidth digital content protection system. `www.digital-cp.com/files/static_page_files/8006F925-129D-4C12-C87899B5A76EF5C3/HDCP_Specification%20Rev1_3.pdf`.

[72] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, May 2008.

[73] T. Markettos, C. Rothwell, B. Gutstein, A. Pearce, P. Neumann, S. Moore, and R. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *Proceedings of the Network and Distributed Systems Symposium (NDSS)*, 2019.

[74] A. Markuze, A. Morrison, and D. Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 249–262, 2016.

[75] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[76] C. Maurice, C. Neumann, O. Heen, and A. Francillon. Confidentiality issues on a GPU in a virtualized environment. In N. Christin and R. Safavi-Naini, editors, *Financial Cryptography and Data*

*Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, volume 8437 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2014.

[77] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proc. IEEE Symposium on Security and Privacy*, 2010.

[78] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proc. Network and Distributed Systems Security Symposium*, 2009.

[79] J. Mclean. Reasoning about security models. In *1987 IEEE Symposium on Security and Privacy*, pages 123–123, April 1987.

[80] L. Mearian. There's no way of knowing if the NSA's spyware is on your hard drive. *Computerworld*, 2, 2015.

[81] MIT Technology Review. Triton is the world's most murderous malware, and it's spreading. https://www.technologyreview.com/s/613054/cybersecurity-critical-infrastructure-triton-malware/ [Accessed on 7 Jun. 2019], 2017.

[82] M. Naor and A. Shamir. Visual cryptography. In *Advances in Cryptology–EUROCRYPT'94*, pages 1–12. Springer, 1995.

[83] Nvidia. Virtual GPU technology. http://www.nvidia.com/object/virtual-gpus.html.

[84] Nvidia. Trusted Little Kernel (TLK) for Tegra: FOSS Edition, 2015.

[85] NXP Semiconductors. I2C-bus Specification and User Manual, https://www.nxpcom/docs/en/user-guide/UM10204.pdf, April 2014.

[86] P. Oikonomakos, J. Fournier, and S. Moore. Implementing cryptography on TFT technology for secure display applications. In *Smart Card Research and Advanced Applications*, pages 32–47. Springer, 2006.

[87] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[88] PCI-SIG. Multicast, https://pcisig.com/specifications, May 2008.

[89] M. Peinado, Y. Chen, P. Engl, and J. Manferdelli. NGSCB: A Trusted Open System. In *Proc. Australasian Conference on Information Security and Privacy*, 2004.

[90] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir. Utilizing the IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 549–562, 2015.

[91] Qualcomm Technologies, Inc. Snapdragon mobile platform - Snapdragon Security, 2019.

[92] Reuters. Malicious virus shuttered power plant: Dhs. https://www.reuters.com/article/us-cybersecurity-powerplants-idUKBRE90F1F720130116 [Accessed on 7 Jun. 2019], 2013.

[93] Reuters. Ukraine's power outage was a cyber attack: Ukrenergo. https://www.reuters.com/article/us-ukraine-cyber-attack-energy-idUSKBN1521BA [Accessed on 7 Jun. 2019], 2017.

[94] C. L. Rothwell. Exploitation from malicious PCI Express peripherals, PhD Thesis, University of Cambridge, Computer Laboratory, UCAM-CL-TR-934, Feb 2019.

[95] X. Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkely, CA, USA, 1st edition, 2014.

[96] J. M. Rushby. Design and verification of secure systems. 15(5):12–21, 1981.

[97] J. M. Rushby. Separation and integration in MILS (The MILS Constitution). February 2008.

[98] F. L. Sang, É. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an I/OMMU vulnerability. In *5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, October 19-20, 2010*, pages 7–14. IEEE, 2010.

[99] F. L. Sang, V. Nicomette, and Y. Deswarte. I/O Attacks in Intel PC-based Architectures and Countermeasures. In *SysSec Workshop (SysSec), 2011 First*, pages 19–26, July 2011.

[100] A. A. Sani, L. Zhong, and D. S. Wallach. Glider: A GPU library driver for improved system security. *CoRR*, abs/1411.3777, 2014.

[101] R. Schell, T. Tao, and M. Heckman. Designing the GEMSOS security kernel for security and performance. In *Proc. National Computer Security Conference*, 1985.

[102] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The multics kernel design project. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 43–56, 1977.

[103] T. Shanley and D. Anderson. *PCI System Architecture*. Addison-Wesley Professional, 4th edition, 1999.

[104] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[105] C. Smowton. Secure 3D graphics for virtual machines. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 36–43, New York, NY, USA, 2009. ACM.

[106] U. Steinberg and B. Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *Proc. European Conference on Computer Systems*, 2010.

[107] Stephen Bates. p2pmem-test. https://github.com/sbates130272/p2pmem-test/blob/master/README.md, 2017.

[108] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why not virtualizing GPUs at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[109] W. R. Systems. Wind river vxworks mils platform. http://www.windriver.com/products/platforms/vxworks-mils/MILS-3_PO.pdf [Accessed on 7 Nov 2013], 2013.

[110] The System Management Interface Forum (SMIF), Inc. System Management Bus (SMBus) Specification Version 2.0, 2000.

[111] K. Tian, Y. Dong, and D. Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[112] Trustonic. Mobile device security is hard - Trustonic makes it easy, 2019.

[113] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 430–444. IEEE, 2013.

[114] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. überspark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 87–104, Austin, TX, 2016. USENIX Association.

[115] VMWare. Graphics acceleration in view virtual desktops. http://www.vmware.com/files/pdf/techpaper/vmware-horizon-view-graphics-acceleration-deployment.pdf.

[116] T. Weigold, T. Kramp, R. Hermann, F. Höring, P. Buhler, and M. Baentsch. The zurich trusted information channel — an efficient defence against man-in-the-middle and malicious software attacks. In *Proc. International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, 2008.

[117] S. Weiser and M. Werner. SGXIO: Generic trusted i/o path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, 2017.

[118] Wikipedia. Screen tearing. `http://en.wikipedia.org/wiki/Screen_tearing`.

[119] Wikipedia. Stuxnet. `https://en.wikipedia.org/wiki/Stuxnet` [Accessed on 7 Jun. 2019].

[120] Wikipedia. CAN bus, 2019.

[121] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. USENIX Conference on Operating Systems Design and Implementation*, 2008.

[122] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[123] Wired. Exclusive: Computer virus hits u.s. drone fleet. `https://www.wired.com/2011/10/virus-hits-drone-fleet/` [Accessed on 7 Jun. 2019], 2011.

[124] Xen. Xen VGA passthrough fetched on 2015-01-19). `http://wiki.xen.org/wiki/Xen_VGA_Passthrough`.

[125] L. Xia, J. Lange, P. Dinda, and C. Bae. Investigating virtual passthrough i/o on commodity devices. *SIGOPS Oper. Syst. Rev.*, 43(3):83–94, July 2009.

[126] Xilinx Inc. Pcie peer-to-peer support. `https://xilinx.github.io/XRT/2018.3/html/p2p.html`, 2018.

[127] H. Yamamoto, Y. Hayasaki, and N. Nishida. Secure information display with limited viewing zone by use of multi-color visual cryptography. *Optics express*, 12(7):1258–1270, 2004.

[128] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 989–1003, New York, NY, USA, 2015. ACM.

[129] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan. VGRIS: Virtualized gpu resource isolation and scheduling in cloud gaming. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 203–214, New York, NY, USA, 2013. ACM.

[130] J. Zaddach, A. Kurmus, D. Balzarotti, E. Blass, A. Francillon, T. Goodspped, M. Gupta, and I. Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proc. of the 29th ACSAC*. ACM, 2013.

[131] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[132] 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*, 2012.

[133] Z. Zhou, J. Han, Y.-H. Lin, A. Perrig, and V. Gligor. Kiss: Key it simple and secure corporate key management. In *Proc. International Conference on Trust and Trustworthy Computing*, 2013.

[134] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 308–323, Washington, DC, USA, 2014. IEEE Computer Society.