An I/O Separation Model for Formal Verification of Kernel Implementations

Miao Yu Virgil Gligor Limin Jia
ECE Department and CyLab, Carnegie Mellon University

***Abstract*—Commodity I/O hardware often fails to separate I/O transfers of isolated OS and applications code. Even when using the best I/O hardware, commodity systems sometimes trade off separation assurance for increased performance. Remarkably, device firmware need *not* be malicious. Instead, any *malicious driver*, even if isolated in its own execution domain, can manipulate *its device* to breach I/O separation. To prevent such vulnerabilities with high assurance, a formal I/O separation model and its use in automatic generation of secure I/O kernel code is necessary.**

This paper presents a formal I/O separation model, which defines a separation policy based on authorization of I/O transfers and is *hardware agnostic*. The model, its refinement, and instantiation in the Wimpy kernel design, are formally specified and verified in Dafny. We then specify the kernel implementation and automatically generate verified-correct assembly code that enforces the I/O separation policies. Our formal modeling enables the discovery of heretofore unknown design and implementation vulnerabilities of the original Wimpy kernel. Finally, we outline how the model can be applied to other I/O kernels and conclude with the key lessons learned.

***Index Terms*—I/O separation; access control and authorization; trustworthy computing; security architectures;**

I. Introduction

An important goal of security architectures is to separate I/O transfers of isolated applications and retain application protection from compromised OSes and other applications, with high assurance. To accomplish this without enlarging the underlying trusted code base (e.g., without enlarging micro-kernels, micro-hypervisors, separation kernels), existing designs rely on dedicated I/O kernels [1], [2], [3], [4]. They deprivilege device drivers, export them to isolated applications to separate them from each other, and authorize them to access only their own devices. This also helps eliminate applications' exposure to unneeded drivers, and is a major advantage since driver code continues to comprise a very large portion of modern OS kernels and accounts for many security flaws.

To ensure that an isolated but malicious driver cannot compromise another isolated application by manipulating its own device, I/O kernels rely on the underlying I/O hardware (e.g., I/O controllers and IOMMUs) to enforce the association of an I/O device with an object of an isolated application/driver and authorize each I/O transfer. Unfortunately, hardware vendors have produced commodity hardware that focuses primarily on improved performance [5], [6], increased connectivity [7], [8], and lower cost [9], at the expense of fine-grained I/O device associations with isolated-application/driver objects and transfer authorization. For example, early PCI buses and more recent CAN buses allow unauthorized peer-to-peer device transfers, which can be leveraged by a malicious driver to access device registers of another isolated application. Other designs can only associate buses with isolated-application objects and enforce *read-write* permissions for buses but not individual devices. For example, IOMMUs [9], [10] authorize accesses at the granularity of PCI *bus controllers* via PCIe-to-PCI bridges instead of individual PCI devices, again, allowing malicious drivers to breach isolation. In addition, insecure performance optimizations, such as deferred IOTLB clearing [11], designed to counter the significant performance degradation caused by frequent switches between authorized transfers, can also lead to breaches of application isolation; see Section II.

This shows that the security guarantees of I/O kernels are intimately connected to the choice of underlying I/O hardware: a poor choice often leads to security vulnerabilities. However, neither a formal model nor a high-assurance design and implementation of I/O separation exists to date. As a result, current I/O kernels cannot match the high assurance of their underlying trusted code base; e.g., micro-kernels, micro-hypervisors [12], [13], [14], [15] and separation kernels [16], [17]. This imbalance can lead to isolated-application vulnerabilities: a malicious application can exploit flawed I/O transfer authorization to breach the isolation of other applications[1].

Our goal is to formalize I/O separation and develop an abstract model, which can be used as the blueprint for high-assurance I/O kernel design and implementations, and make explicit the assumptions about underlying hardware-authorization properties. Our model *does not preclude* hardware designs with inadequate authorization, like PCI, PCIe-to-PCI bridges or USB host controllers, as they occupy a large fraction of the marketplace. Instead, it makes explicit the kernel design and implementation requirements for high assurance, if such hardware were to be used.

We define an abstract I/O separation kernel model in Dafny [20] (Section IV), after outlining the need for its four-layer refinement for real system use (Section III). We define key components and operations of I/O devices and drivers, specify transfer authorizations, and formalize two desired security properties: *no transfer across an I/O separation boundary* and *no object reuse* in on-demand I/O. We prove the abstract I/O separation model satisfies these two properties. We then define a concrete I/O model that includes more detailed notions of separation and I/O transfers (Section V).

[1]Similar vulnerabilities have already been witnessed when isolated VM applications rely on isolated drivers [18] to ensure I/O separation. A single driver could still exploit I/O hardware to bypass device-transfer authorization [19].

We construct a mapping to the abstract model and prove the concrete model sound via refinement. Applying the models to real I/O kernels, we instantiate the concrete model to the Wimpy kernel *design* [21], and we prove its soundness via refinement to the concrete model (Section VI). Finally, we specify and verify the Wimpy kernel *implementation* in Vale/Dafny [22] and automatically generate verified-correct assembly code that enforces the I/O separation policies—an unavailable feature of any OS kernel to date (Section VII). This formal process leads to discovery of vulnerabilities in the original design (Section VI-B) and code (Section VII-B).

Due to space constraints, the paper focuses on explaining the high-level concepts. All Dafny and Vale specifications, from model to kernel code, and proofs can be downloaded from *https://github.com/superymk/iosep_proof/raw/master/proof.zip*.

II. Common I/O Vulnerabilities and Threats

We review the different ways I/O hardware authorizes accesses, summarize vulnerabilities caused by inadequate I/O hardware authorization, and present the threats countered.

A. I/O Transfer Authorization and Separation

An I/O *transfer* is informally viewed as an ordered association of one or more devices to one or more I/O objects of an isolated application/driver. In the simplest case, the association is one-to-one; e.g., a USB device can exclusively transfer data to a buffer of an isolated driver in an application. A single device can also be associated with objects of several isolated drivers; e.g., a single GPU device can display output of several isolated application drivers concurrently. Several devices can be associated on demand with a single I/O buffer of an isolated application sequentially shared by several drivers; e.g., several USB devices. In all cases, the device-object order of the association indicates whether the I/O object is read or written.

An I/O transfer is *authorized* if a driver cannot 1) bypass or modify its device's association with the isolated-driver object, and 2) perform the transfer without the permissions (i.e., *write*, *read*) required by the association order. Failure to enforce 1) or 2) by inadequate hardware can enable isolated but malicious drivers to breach I/O separation of isolated applications. Conversely, high-assurance authorization requires formal analysis of both 1) and 2).

Note that merely enforcing memory *address-space separation* for drivers and applications, while useful [23], is insufficient for transfer authorization. For example, on-demand activation of a device/driver from one isolated application to another can cause *use-after-free* violations without breaching address-space separation. Similarly, a *peer-to-peer* device transfer crossing different applications can violate their isolation without breaching address space separation. Also, a malicious driver executing a single instruction that *broadcasts data* to the registers of multiple devices can violate application isolation without breaching address-space separation.

B. Inadequacy of Existing Hardware

Existing hardware that authorizes I/O transfers at different levels of granularity is summarized in Figure 1.

	No authorization	***Non-selective authorization***	***Selective authorization***
Intel AMD	• PCI, no ACS • SMBus	• PCIe-to-PCI bridge with IOMMU	• PCIe with IOMMU & ACS
ARM	• AHB, no ACS • early: ASB	• AXI-to-AHB bridge with SMMU • TZ within normal or within secure world	• AXI with SMMU • TZ normal vs. secure world
IBM	• PCI, no ACS	• PCIe-to-PCI bridge with IOMMU	• PCIe with IOMMU/CAPI & ACS
External buses	• CAN • I2C	• Firewire (with OHCI) • USB (with IOMMU)	

Fig. 1. **Examples of authorization levels of I/O hardware.**

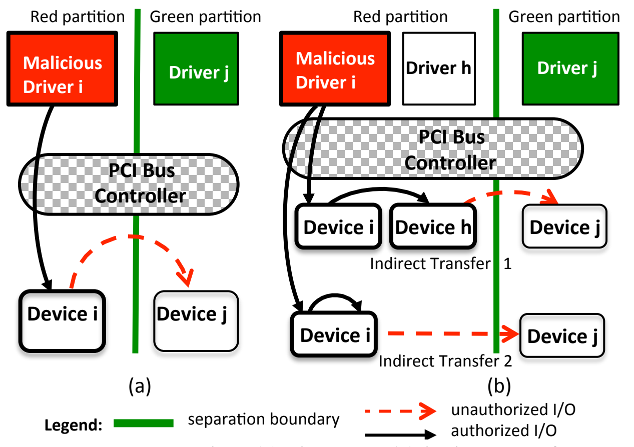


Fig. 2. **Unauthorized (a) direct and (b) indirect transfers.**

1. No authorization. The first column identifies several buses that completely fail to authorize transfers; e.g., neither selectively associate individual devices with isolated-application objects nor enforce *read-write* permissions for I/O transfers. A device can access another device's data registers 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' authentication [24], [25], [26], [27], and yet allow device peer-to-peer (P2P) transfers. Sender devices always have the same privilege as the bus controllers in accessing configuration and data registers of recipient devices. This enables unauthorized transfers to these objects; i.e., a *write* over the recipient device data registers or a *read* from the recipient device. When devices are connected to these buses, isolated but malicious drivers can manipulate them to perform unauthorized *direct* and *indirect* P2P transfers to other devices, as illustrated below.

Example 1. Unauthorized direct transfers. As shown in Figure 2(a), a malicious driver i can configure its device i to perform a device P2P transfer and access device j of another isolated application without any authorization, thereby breaking I/O separation.

Example 2. Unauthorized indirect transfers. Figure 2(b) shows Indirect Transfer 1 whereby a PCI device i configures

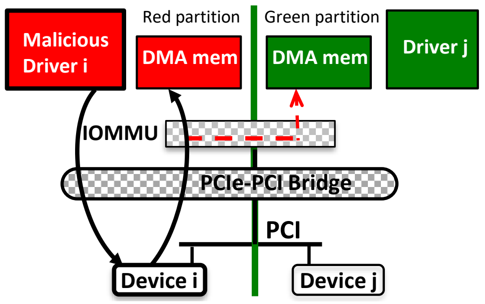


Fig. 3. **Non-selective authorization of I/O transfers.**

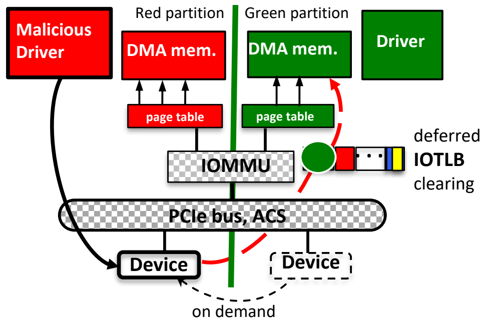


Fig. 4. **Unauthorized transfer caused by deferred IOTLB clearing.**

another PCI device h via a P2P transfer and enables it 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, direct transfers of device i to device h do not break I/O separation. Instead, transfers of device h to device j do. In Indirect Transfer 2 of Figure 2(b), device i writes over one of its own transfer descriptors, which enables device i to issue an unauthorized cross-boundary transfer to device j.

2. Non-selective authorization. The second column of Figure 1 shows I/O buses that fail to support *individual* device association with isolated-driver objects and enforce their *read-write* permissions; i.e., access authorization is non-selective. Instead, these commodity bus controllers can only associate buses with these objects and enforce *read-write* permissions for buses. For example, some IOMMUs [9], [10] authorize accesses at the granularity of PCI *bus controllers* via PCIe-to-PCI bridges instead of individual devices. Similarly, when issuing requests on behalf of their individual DMA devices, USB host controllers [8] use their own identities instead those of individual devices. In both cases, the IOMMU regards all DMA device transfers as originating from the I/O bus controller. Hence, it cannot authorize transfers selectively per individual device.

Example 3. Non-selective authorization of transfers. As shown in Figure 3, device i is manipulated by malicious driver i to *read* or *write* a DMA memory region of another device j across an isolation boundary, even though the IOMMU is correctly configured. For selective authorization of transfers, devices of different isolated applications must be connected to different PCIe-to-PCI bridges yielding restricted hardware configurations [4]. ARM TrustZone authorizes I/O transfers selectively between the normal and secure world, but it cannot authorize transfers selectively within a single world.

3. Selective-authorization failure from optimization. The last column of Figure 1 illustrates the best hardware for selective (per device) authorization. Unfortunately, commodity OS kernels using this hardware often have to trade transfer-authorization assurance for added performance. For example, to decrease the significant cost of selective authorization via an IOMMU [28], commodity OS kernels perform *all* transfers into one shared kernel buffer-pool, and then authorize kernel transfers to/from isolated applications [11]. This works well for low-assurance kernels (e.g., Linux), but adds substantially more complex code to high-assurance micro-hypervisors [13], [14] or micro-kernels [12], [15] and causes significant performance degradation due to frequent switches to/from them.

Example 4. Unauthorized transfer enabled by a performance–I/O isolation trade-off. Figure 4 illustrates the vulnerability of this performance-isolation assurance trade-off via an attack that breaches DMA memory isolation by exploiting the deferred clearing of IOMMU's IOTLB entries up to 10 ms [28], [11]. During this delay, the DMA device is allocated to a malicious (red) driver, which is isolated in another application, on demand. The malicious (red) driver instructs the DMA device to issue a transfer whose target virtual address is translated using the undeleted IOTLB entry (green circle), breaching red-green separation.

Similar vulnerabilities can be caused by green applications, which could breach the isolation of other green applications.

C. Threats

The examples above show how an adversary can breach *confidentiality* and/or *integrity* of sensitive I/O data across isolated applications. The emerging attack pattern is simple: the adversary either compromises an OS driver or provides an isolated application containing a deliberately compromised driver. Then the compromised driver can mis-configure its device to setup unauthorized I/O transfers either directly or indirectly. These attacks appear in all commodity OSes—even when they benefit from formally verified micro-hypervisors and micro-kernels that isolate device drivers.

We assume that attackers control drivers of their devices. Note that an attacker need *not* corrupt device firmware to launch the I/O separation attacks; e.g., surreptitiously modify the device controller's firmware by re-flashing [29], [30], [31] or by supply-chain compromise [32], [33], [34], [35]. Techniques to *verify* the correct device firmware and register contents after re-flashing are known [36] and hence are not addressed here. Also, we assume that legitimate backdoors are disabled before system operation; e.g., hardware debugging interfaces for privileged access to I/O devices [37]. As customary, we also assume that device hardware is non-malicious. Denial-of-service and covert-channel attacks in I/O transfers are also irrelevant to the I/O separation model.

III. Model Motivation and Layered Approach

The challenges of building an I/O separation model are to provide security guarantees for (1) different I/O hardware designs, different types of I/O accesses (e.g., P2P, broadcast), direct and indirect transfers; and (2) different I/O kernels, ranging from high-assurance, such as Wimpy kernel (WK) [21] and the GPU separation kernel (GSK) [38], to low-assurance OS kernels (Linux) [28], [11]. The model must apply to *all* trusted execution environments and hardware-supported enclaves. We outline why these challenges are important and how to meet them via a layered modeling approach.

A. Motivation

Application to Different I/O Hardware Designs. It must be possible to instantiate the formal model on *any* commodity I/O configurations regardless of the hardware ability, or lack thereof, to authorize I/O transfers. The reality is that a large variety of commodity processors which support application isolation (e.g., via trusted execution environments [39], partitions [16], [17], pieces of application logic [40], [13], [14], and enclaves [41], [15]) will continue to be interconnected to I/O hardware that fails to adequately authorize separate device transfers. This is both for high-performance and low-cost; e.g., in cyber-physical systems (CPS) [42] and vehicular computing [42], [26]. Our formalism suggests specific ways to handle inadequate I/O hardware.

1. *No authorization*. In this case, an I/O kernel defines separate device-driver associations and transfer permissions. Driver code is either (a) formally verified not to violate the defined authorizations or (b) de-privileged, exported to isolated applications, and have all its device accesses authorized by the I/O kernel at run time, or a mixture of both (a) and (b). In case (a), verified drivers are authenticated at boot time (via secure and trusted boot) and configured within the I/O kernel. This is practical only for systems with few devices; e.g., CPS [42] and vehicular computing [42], [26]. In case (b), unverified driver accesses to devices could incur substantial overhead, and hence could only be used for drivers that access devices infrequently. In both cases, the model shows how authorization is formally specified and verified; using transitive closures of transfer descriptors; see Section IV-B.

2. *Non-selective authorization*. In this case, the I/O kernel has two non-exclusive options, and is more scalable than the no-authorization remedy above. First, only the driver code of devices connected to the *same* PCI bus needs to be formally verified to satisfy individual device-transfer authorization, and only these drivers need to be authenticated and configured within the I/O kernel during boot. Second, whenever practical, I/O configurations are restricted to a single device per PCI bus and this is enforced at boot time. The drivers of these devices are untrusted and can be exported to isolated applications.

3. *Selective-authorization failure*. Here, a formal model applied to micro-hypervisors and I/O kernel verification shows that all separated but malicious drivers are de-privileged and safely exported to isolated applications along with their local buffers. This solves the I/O separation breaches shown in *Example 4* of Section II-B, and substantially simplifies both formal micro-hypervisor and I/O kernel verification and naturally avoids performance penalties (Section VIII-B). To enable this, the I/O kernel relies on the I/O hardware to enforce transfer authorization *late*, and the benefits of late authorization are discussed in Section IV-D.

Application to Different I/O Kernels. Different I/O kernels isolate drivers in different ways: some isolate them within the I/O kernels themselves, while others isolate them within applications; e.g., in partitions of separation kernels, isolated pieces of application logic supported by micro-hypervisors, trusted execution environments, or hardware-isolated enclaves. Some I/O kernels support device activation on demand whereas others support only static activation during system boot. Despite their differences, all I/O kernels support a notion of *separation* to encapsulate drivers and their associated I/O objects (e.g., data buffers and configuration registers) and devices of isolated applications. Hence, any model must support this notion, and we do this via *I/O partitions*; i.e., at any given time, each device, driver, and object belongs to *one and only one* partition, and they can move from one partition to another. Then two security policy properties that an I/O separation model should naturally enforce are, at a high level: (1) no cross-partition transfers, and (2) no object data reuse in a new partition in on-demand I/O. We will make these more concrete in Section IV. The formal application of the model to an I/O kernel is discussed in Section V whereas the informal application to other I/O kernels is in Section VIII-A.

B. Layered Modeling

Figure 5 illustrates our layered modeling approach that yields a verified assembly implementation of Wimpy kernel. We use Dafny and start from an *abstract I/O separation model* and use a hierarchy of verified refinements. The abstract I/O separation model specifies key device and driver components and operations, formalizes the notion of I/O partitions, and specifies I/O authorization and properties.

The second layer comprises *Concrete I/O models*, which are obtained from (verified) refinement of the abstract model, have more details, including specific types of I/O hardware authorization capabilities and I/O separation policy, as discussed above. The soundness of a Concrete I/O model is proven by leveraging the simulation relation between the abstract I/O separation model and the Concrete I/O model.

The third layer is *I/O kernel designs* whose formal specifications represent the (verified) refinement of the Concrete I/O model in different device classes, whereby some or all mechanisms for device identification, initialization, transfer authorization may differ among these classes. This allows the instantiation of different I/O kernels for typical OS device classes: a Wimpy kernel, a GPU separation kernel (see Figure 5), a NIC kernel, etc. Then soundness of an I/O kernel design specification is proven by the simulation relation between the Concrete I/O model and the I/O kernel design specifications.

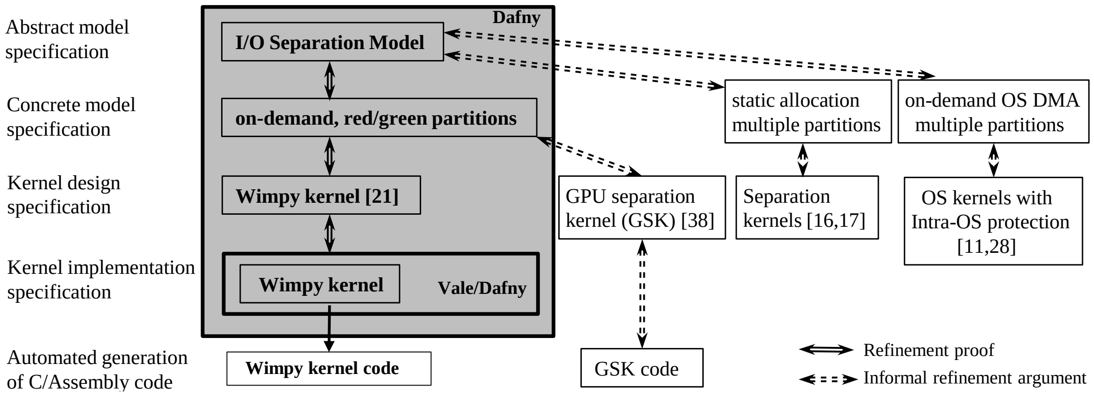

Abstract model specification
Concrete model specification
Kernel design specification
Kernel implementation specification
Automated generation of C/Assembly code
Dafny
I/O Separation Model
on-demand, red/green partitions
Wimpy kernel [21]
Wimpy kernel
Vale/Dafny
Wimpy kernel code
static allocation multiple partitions
on-demand OS DMA multiple partitions
GPU separation kernel (GSK) [38]
Separation kernels [16,17]
OS kernels with Intra-OS protection [11,28]
GSK code
Refinement proof
Informal refinement argument

Fig. 5. **Model Refinements and Automated Code Generation**

Using an intermediate concrete-model specification reduces the proof effort for different I/O kernels refined from the same concrete model. Further, kernel-design specifications are necessary for three practical reasons. First, design specifications and (failure of) their soundness proofs enable early detection of security vulnerabilities before any effort for implementation specifications, code generation, and their testing is expanded; e.g., vulnerabilities of the original Wimpy-kernel design [21] in Section VI-B. Second, the different I/O kernel designs (e.g., Wimpy kernel, GPU separation kernel, and NIC separation kernel) obtained by device class partitioning can be provably composed within a secure I/O subsystem. This can further lead to the sound composition of implementation specifications; e.g., composition is facilitated by the sound correspondence between individual I/O kernel designs and their implementation specifications. Thus, the much more complex provable implementation- and code-composition effort can be avoided. Third, design changes for a device class often retain the same I/O kernel design specifications and avoid constructing new soundness proofs; e.g., USB 3.0 on-demand time-multiplexing mechanism does not change USB 2.0 I/O kernel specifications.

The last layer of specifications is *I/O kernel implementations* in Vale/Dafny [15], so x86 assembly code can be automatically generated and the implementations can be proven correct in Dafny. This refinement proof shows that the Wimpy kernel implementation correctly refines its design specification.

C. Code sizes, level of effort, and model reusability

All the models and refinement proofs shown in the gray box of Figure 5 are formally specified and verified.

Code sizes. The I/O separation model takes 28,518 lines of Dafny code (LOC) the concrete model takes 47,120 LOC, the WK design takes 55,426 LOC, and the WK implementation takes 136,815 of Dafny and Vale LOC. The WK implementation are more than double the size of the design specifications because it requires formal refinement of subjects, objects, and operations for x86 platforms. The automatically generated assembly code takes 12,031 instructions.

Level of effort. The overall level of effort for the model design, specifications of the verified refinements, proofs, and WK implementation was 4.5 person-years, of which 2.5 person-years were required for the sound WK design and implementation.

Model reusability. Reuse of the model components can save substantial level of effort for future I/O separation designs. For example, the abstract model can be profitably reused for other secure systems, since its separation properties are general; see Section VIII-A. This would save over a person-year worth of effort. Both the abstract and concrete model specifications can be reused for the GSK design (see Figure 5) saving about 2 person-years or about half of the overall effort. Finally, improved proof structure could also lead to decreased level of effort; e.g., caching and reuse of proof results, reuse of the many lemmas that are common to different proofs.

IV. I/O Separation Model

We describe our labeled-transition-based abstract I/O separation model and define the security properties of I/O separation. Detailed definitions are in Appendix A.

A. Abstract Model State

We list key I/O components then describe abstract state.

Overview of Devices, Drivers, and I/O Objects. Device *drivers* are arbitrary programs that run on CPUs and read-/write devices. *Devices* include 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) or Port I/O (PIO) devices, *read* and *write* memory via Direct Memory Access (DMA), and request CPUs' attention via interrupts.

Device drivers 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. 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. *Function descriptors* (FDs) define configurations of device functions other than I/O 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. A TD contains a list of entries, each comprising a pointer to an I/O object, requested access modes, and new values to write. These TDs encode separation policies and thus require special attention.

Each *internal* I/O object is owned by a device or a driver, while the *external* I/O objects are not owned by any devices or drivers. 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 (e.g., interface data structures for protected USB devices) from drivers to authorize driver access to these objects.

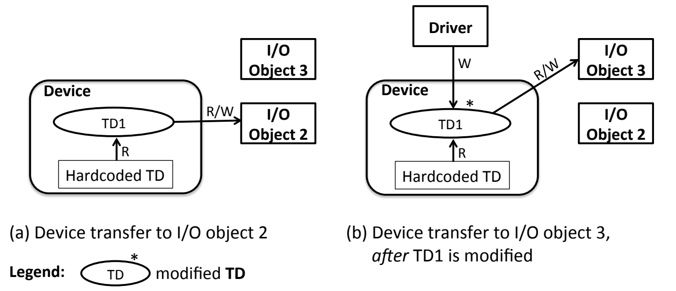
Each device owns a *hardcoded TD*, which defines local transfers issued to the device's own *internal objects*, based on the device's hardware/firmware implementation. For example, a GPU's hardcoded TD defines *read* transfers to TDs of frame-buffer base registers to enable the GPU to poll their configurations. 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.

Devices, drivers, and objects can be *inactive* or *active*. Inactive devices and drivers cannot perform any I/O transfer. Active devices and drivers can issue I/O transfers. Active objects are the ones accessible by active devices and drivers.

State Definition. The abstract system state encompasses I/O device and driver states that are relevant to I/O separation. To facilitate specification of access-control (authorization) policies, the model consists of subjects and objects and a set of partitions containing devices, drivers, and I/O objects. The set of *subjects* comprises *Drivers* and *Devices*. All subjects have unique (e.g., non-reusable) names. Each driver is associated with a partition ID (i.e., the partition holding the driver) and a set of objects IDs, owned by the driver. Each device is associated with an ID of the hardcoded TD it owns. *Objects* comprise I/O objects TDs, FDs, and DOs, as introduced above. Objects are also associated with partition IDs. FDs and DOs have string values. Each TD's value is a list of tuples of a pointer to an object, requested access modes, and new values to write. Partitions have unique names. A special NULL I/O partition includes all inactive subjects and objects. Active subjects and objects can only be in non-NULL partitions.

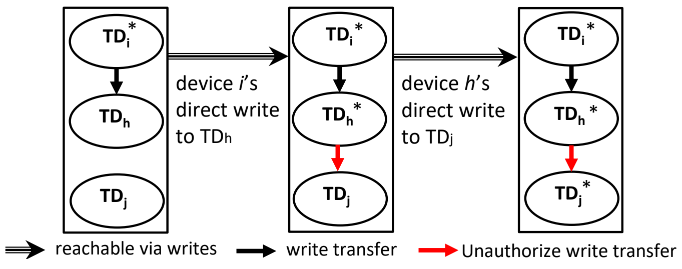
B. State Transitions

A *state transition* occurs when an I/O operation modifies a system state. Operations in I/O systems are categorized into four groups: I/O transfers, partition creations and destructions, subjects/objects activations, and subjects/objects deactivations. Each operation is defined as a Dafny method and includes authorization decisions of the abstract I/O separation model.



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

Fig. 6. **Device transfers authorized by TDs.**



TD$_i$*
TD$_h$
TD$_j$
device *i*'s direct write to TD$_h$
TD$_h$*
device *h*'s direct write to TD$_j$
TD$_j$*
reachable via writes
write transfer
Unauthorize write transfer

Fig. 7. **Computing transitive closure of TD state**

1) I/O transfers and their authorization: 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 can *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 6(a) can issue a read/write 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; see Figure 6(b).

For these I/O transfers, the kernel needs to check that the subjects and objects of the transfers are in the same partition. However, this check alone is insufficient to prevent cross-partition indirect transfers; see Figure 2 (b). To do so, *writes* to TDs require a more complex check, which we detail next.

Example Transitive Closure of Active TDs. Writes to active TDs enable *indirect* transfers by devices. For example, in the Indirect Transfer 1 of Figure 2(b), malicious driver i writes to TD$_i$ of device i, and hence configures device i to *write* the TD$_h$ of device h, which causes 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. To accommodate hardware that cannot mediate I/O transfer between h and i (e.g., device P2P transfers), the kernel needs to prevent driver i from writing TD$_i$ with values that may enable unauthorized future indirect *write* transfers.

The solution is to first compute a transitive closure of the state of all active TDs when TD$_i$ is written by driver i, and then check that the transitive closure does not include cross-partition accesses. Figure 7 builds on the example of Indirect Transfer 1 of Figure 2(b) to illustate how the transitive closure computation reveals the unauthorized *write* indirect transfer to the TD$_j$ of device j. Each box represents a state of all active TDs (denoted as *TD_state*), the leftmost one being the current state containing all the active TDs when i has written to TD$_i$. TDs with an * are updated and different from the initial state. The computation starts from the leftmost state and discovers that driver i's write to TD$_i$ has created a write transfer from device i to TD$_h$, indicated by the arrow from TD$_i$ to TD$_h$. The algorithm then generates a new TD state, as illustrated by the second box, after allowing i to write to TD$_h$. This write creates a write transfer from h to TD$_j$, which is cross-partition and marked in red. At this point, our algorithm has not determined this transfer is cross-partition. In the next step, the algorithm analyzes this new TD state and discovers that it enables device h to write TD$_j$, and generates the third TD state by allowing h to write TD$_j$. Now, the algorithm examines the write issued and determines that this is an unauthorized indirect transfer. At this point, the kernel knows that i's write to TD$_i$ shouldn't be allowed. The transitive closure computation aims to include all reachable TD states from the initial state, until a fixed point is reached or an unauthorized access is found. In this example, the transitive closure computation includes the three TD states shown. More details are in Appendix A-D.

2) Partition creation/destruction: A new I/O partition with ID new_pid can be created as long as new_pid has not been used before. A partition with ID pid can be destroyed if no subject or object exists in that partition, to ensure that no objects have dangling references to non-existent partition IDs.

3) Object activation/deactivation: Inactive subjects and external objects can be activated into an existing partition. To ensure no data leaks or accidental references to data occur in the old partition, the kernel clears the data contained in those objects, except when it is a device's (immutable) hardcoded TD, as we assume the firmware is trusted. If the threat model considers device firmware to be malicious, the verified correct firmware can be locally re-flashed and/or remotely attested.

Active subject and external objects can be deactivated by setting their partition ID to be NULL. The kernel needs to ensure that no active device can issue transfers to the item to be deactivated by computing and checking the transitive closure of the *TD_state* in the current system state. This check is necessary; otherwise, after a deactivate operation, active devices can issue transfers to inactive objects, and further issue cross-partition transfers after these objects are reactivated into new partitions. For example, consider the case where 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. Merely clearing all objects of driver h does not remove the insecure transfer, and hence cannot prevent this attack. That is, the check of deactivate operations ensures that *the lifetime of configuration of transfers to an object must be strictly less than the lifetime of the object in a partition.*

C. Soundness

We define the following two isolation properties, over the execution traces of the abstract I/O separation model.

(SP1) *No I/O transfer crosses a partition*;

(SP2) *Only hardcoded TDs can be reused in a new active 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. Further, when objects are moved into a new partition (i.e., by object deactivation and reactivation), they may be accessed by all subjects of the new partition, but not by the old ones. The subjects in this new partition also cannot *read* any value written by any subjects of the old partition. To do otherwise, violates SP2: either TDs are reused or the new partition is the NULL partition. In the later case, subjects cannot issue 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.

We prove that our abstract I/O separation model is sound with respect to the above mentioned security properties.

Corollary 1 (Soundness of abstract model)**.** *The abstract I/O separation model satisfies SP1 and SP2.*

Next, we explain how to prove the soundness theorem in Dafny. First, we identify *state invariants*, which define the security guarantees of abstract model states (Appendix A-C). These state invariants are preserved by all state transitions. We show one example invariant below.

(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 the TD, 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 is 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 the fact that hardcoded TDs cannot be accessed by drivers, reflects the invariant that hardcoded TDs are local to devices and are immutable between firmware re-flashings.

Then, we define *transition constraints* (aka., transition property), a set of properties for every possible state transition. Each transition constraint specifies properties of the starting state, resulting state, and the operation. Below, we illustrate the constraint used to prove SP2. Appendix A-C contains all the transition constraints.

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

The general properties for a transition from state k to k' that we prove 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 an I/O transfer operation is allowed by the kernel, then the subjects in that operation must be in the same partition as the objects accessed by that operation. Property (3) can be derived from the combination of the state invariant **(SI1)**, which ensures only same-partition transfers from device reads and writes are allowed by the TD states, and the semantics of driver reads and writes, which check the partition IDs of objects being accessed. We then show that the initial state is secure and use the above one-step lemmas and inductively show that both the state invariants and transition constraints hold on all traces.

Theorem 1 (State Invariants)**.** *If state k_n is the resulting state after a sequence of n transitions from 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 case, where k_i transitions to k_{i+1} and $i \in [0, n-1]$, and k_i is assumed to fulfill all state invariants, k_{i+1} must fulfill all state invariants due to 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.

Our proof relies on a set of axioms (formally defined in Dafny specifications) to represent common assumptions about devices and drivers; see Appendix A-C.

D. Discussion: Late versus Early Authorization

Authorization of all I/O transfers based on transitive-closure computation is necessary to accommodate hardware with inadequate I/O authorization; i.e., for the model to be *hardware agnostic*. This also shows how the (un)availability of adequate I/O hardware can impact a system's security and performance.

A typical system with I/O hardware for access authorization supports what we call *late authorization*: the I/O kernel authorizes all direct transfers at the time they happen. Late authorization is not possible with inadequate I/O hardware; e.g., the indirect accesses in Figure 2 (b). Instead, conservative *early authorization* where the kernel computes the transitive-closure and rejects transfers that enables future cross-partition accesses before the access really happens is needed. It 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 authorization 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 I/O transfer authorization 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 2* and Figure 2(b), 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, authorization without adequate I/O hardware support needs to stop this write *early*; otherwise, it invites successful attacks.

In short, late authorization is always sound and complete whereas early authorization, while always sound, can be incomplete.

V. A Concrete I/O Model

We describe the verified refinement of the abstract I/O separation model to obtain a concrete I/O model, which is further refined to obtain the Wimpy kernel design.

A. Isolation and Device Policies

This concrete I/O model makes explicit its policy for separating transfers of isolated (green) applications from those of untrusted (red) OS and applications using I/O hardware authorization and specific ephemeral-device activation policies.

Red-Green I/O Partitions. The red-green I/O separation policy allows untrusted (red) OS drivers to run and access all their devices in an untrusted (red) I/O partition; aka., the "red partition". When an isolated green application requires I/O separation on-demand, the I/O kernel creates a new I/O 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 isolated green partitions to enforce I/O separation for multiple isolated applications, while only one red partition exists. The red partition must exist in this on-demand concrete separation model, because the untrusted (red) OS and applications run first, before loading the I/O kernel to run security-sensitive green applications (often by a micro-hypervisor or micro-kernel).

Ephemeral Devices. Ephemeral devices are created by multiplexing shared physical devices to create the illusion of separate physical devices. For example, an I/O kernel can multiplex a shared bus controller to create separate ephemeral devices [21]. Different I/O kernels implement different policies for ephemeral-device activation. This concrete model enforces the separation policy that a physical device and its mapped ephemeral devices must not be active at the same time. The creation of separate ephemeral devices enables secure multiplexing of physical devices, but does *not* address I/O separation; vulnerabilities shown in Section II still persist.

Ephemeral differ from virtual devices in several ways. For example, unlike virtual devices [43], they have a short lifetime, as short as a few I/O transfers. They often provide minimal functionality required by isolated drivers, whereas virtual devices aim to provide rich functionality to support OS and applications. Unlike virtual devices, whose device activation policy is uniform across different VMs, ephemeral-device activation depends on the types of physical devices they multiplex.

I/O Policies. This concrete model enforces different I/O transfer policies for drivers and devices in different partitions, reflecting differences in the underlying authorization mechanisms. For example, by default, drivers of a green partition (green drivers) can write to external TDs of devices associated with that partition, which allows the (ephemeral) bus controllers configured by green drivers to read these TDs and issue transfers to I/O objects in other green partitions. To prevent bus controllers from issuing direct transfers to objects in a different green partition, an I/O kernel needs to authorize transfers issued by the bus controllers when green drivers write TDs. This authorization requires TD-state transitive closure computation. In practice, an I/O kernel may not compute the transitive closure for performance reasons, and instead may impose a stronger and more conservative policy that forbids green devices from writing to TDs.

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 P2P transfers, I/O multicast and broadcast. The I/O kernel ensures all transfers issued by red drivers and devices are confined to the red partition by leveraging existing hardware security isolation mechanisms; e.g., setting the IOMMU page tables and resetting the IOTLB. The I/O kernel also prevents configurations that allow P2P transfers on PCI buses, which have inadequate access control of I/O transfers.

B. Defining the Concrete Model

Compared to the abstract model, the concrete model includes additional I/O operations using abstractions of specific hardware isolation mechanisms such as IOMMU and IOTLB and extended device and driver (de)activation conditions. New state invariants and transition constraints are applied to the concrete model to prove soundness.

State. In addition to everything in the abstract I/O separation model, a concrete-model state includes a dedicated partition ID for the red partition and an ephemeral device map, mapping each ephemeral or physical device to a set of ephemeral devices and indicating which ones are active. This map helps enforce the separation policy of ephemeral devices and ensure that a physical device is not active at the same time as its mapped ephemeral devices.

In the initial state of the concrete model, the red partition is the only active partition, and devices, drivers, and objects are either active in the red partition or inactive.

State Transitions. Each operation of the concrete model maps to one and only one operation of the I/O separation model, as shown in Table I of Appendix B. For each operation in the abstract model, this concrete model makes a distinction between green and red partitions.

The driver *write* operation in the abstract model is split into a *green driver write* and a *red driver write* operations. The former relies on I/O kernel code for access control. The latter may rely on available I/O hardware mechanisms (e.g., IOMMU and/or PCIe ACS) to block any cross-partition transfers issued by red devices and thus the concrete model imports isolation assumptions based on the hardware mechanisms. This is because green applications require I/O separation of ephemeral devices at finer-granularity than can be achieved by even perfect I/O hardware mechanisms alone. In contrast, I/O kernels only need to prevent red drivers from accessing the physical device mapped to ephemeral devices, which can be achieved by using I/O hardware mechanisms.

Similarly, driver, device, and external object activation and deactivation operations are split into a green and a red version. Red drivers and their external objects have (de)activation operations acquire/release memory in the red partition on-demand and they are never allowed to be moved to a green partition. The green drivers and external objects are directly activated into green partitions on demand, but never in a red partition. Red driver, device, or external object deactivation operations require that no other red device can issue direct or indirect transfers to it. This requirement is fulfilled by using adequate hardware mechanisms that are capable of fine-grained access control such as IOMMU and PCIe ACS. The model imports this requirement as an Axiom.

State Invariants and Transition Constraints. For simplicity of exposition, we show only how a single state invariant, SI1, of the abstract model is refined to obtain invariants in the concrete model. The transition constraints are almost identical to those of the abstract model, so we omit them. Similar to the operations, the state invariants are concretely applied to green and red partitions.

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) in the red partition, and (ii) must not be hardcoded TDs.

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 direct interpretation of SI1 in the abstract model to green devices, as no indirect transfer is allowed at all by SI2c(ii). An I/O kernel could enforce the weaker SI1 for the green partition by computing a transitive closure, but this is less desirable because of the performance penalty incurred by doing so. SI2c is a more conservative policy which can be easily implemented by the I/O kernels because only green drivers can configure transfers by USB host controllers, USB devices, and (ephemeral) interrupt controllers, if any. In Section VI-B, we will illustrate a vulnerability in the original Wimpy kernel design, which violates SI2c.

C. Soundness of the Concrete Model

The I/O separation properties for the concrete model are: (1) subjects can only issue I/O transfers to objects in the same red or green partitions; (2) only hardcoded TDs can be reused in a new green or red partition. These properties are instantiations of SP1 and SP2 to green and red partitions.

To prove that the concrete model enforces I/O separation, we need to prove the refinements of **Theorem 1** and **2**, and **Corollary 1** of the (abstract) I/O separation model. 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.

We leverage the simulation relation between the abstract and concrete model in proving these theorems. We formally define a mapping f from concrete states, operations, and transitions to abstract states, operations, and transitions as illustrated in Table I of Appendix B. (For brevity, we overload f for all of the mappings.) We then prove 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 there exist o and k_2 such that k_1 can make a successful transition to k_2 under o; i.e., such that $f(ck_2) = k_2$ 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 which cannot. For example, SI1c belongs to the first group, while SI2c belongs to the second group. 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. The proofs of (1) – (3) are straightforward since 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 is stronger than SI1. A direct proof of the second group of properties is trivial, because concrete operations enforce these properties in their specifications.

VI. Wimpy Kernel Design

The Wimpy kernel (WK) [21] aims to enforce on-demand I/O channel separation for isolated applications. It shares the same I/O separation goals of the concrete model in Section V-A and is specialized to ensure the proper red/green parition separation in the presence of USB host controllers and USB peripheral devices. We describe its specification, an instantiation of the concrete model from Section V, and finally discuss its vulnerabilities.

A. Instantiating Concrete Model to WK Design

The WK design inherits much of the state, operations, state invariants, and transition constraints from the concrete model from the previous section. One specific operation is the activation of USB host controllers and devices into the red partition, which is specified as *concatenations* of concrete-model operations (Table II in Appendix B). The concatenation ensures the atomicity of this operation: all of the devices are activated at the same time. The WK design refines the ephemeral-device policy from the concrete model to require an ephemeral device for the same physical device to be active in one partition. The soundness proof of WK design relies on a straightforward simulation proof, as the mappings are mostly identity functions.

B. Vulnerabilities of the original WK Design

We compared the *sound* kernel specifications with the original design [21] and discover discrepancies that lead to vulnerabilities. We illustrate the two vulnerabilities found and identify the invariants they violate.

Vulnerability 1: *A violation of red-green separation. Red devices can issue unauthorized transfers to green drivers, devices, and external objects.*

This vulnerability is caused by a violation of the state invariant SI1c of the sound WK design and concrete I/O model, which in turn violates invariant SI1 of the abstract model. Consequently, the separation property SP1 can not be proven in the original design.

This vulnerability happens when a red device is under the same PCI bus with the isolated USB Host Controller (USB HC). Because PCIe-to-PCI bridges authorize device P2P transfers and DMA transfers at the granularity of PCI bus controllers, the vulnerability can be exploited as in Example 1, 2 and 3 in Section II. WK incorrectly assumes that IOMMU and PCIe ACS mediate these transfers.

To remove this vulnerability, the sound WK design restricts hardware configurations to prevent such transfers. These restrictions require green USB host controllers to be attached only to PCIe (not PCI) buses, and then the IOMMU and PCIe ACS block unauthorized red-device transfers. Note that the WK design could implement the transitive closure computation and checks of the abstract model, but it is inefficient. So the WK design deviates from the abstract model and makes explicit assumptions about the I/O hardware.

Vulnerability 2: *A violation of green-green separation. Green drivers can issue indirect writes to external TDs and cause their devices to 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 WK design, which denies *indirect* transfers in green partitions. Figure 8 illustrates how to exploit this vulnerability. Nexus RVM [2] also has this vulnerability.

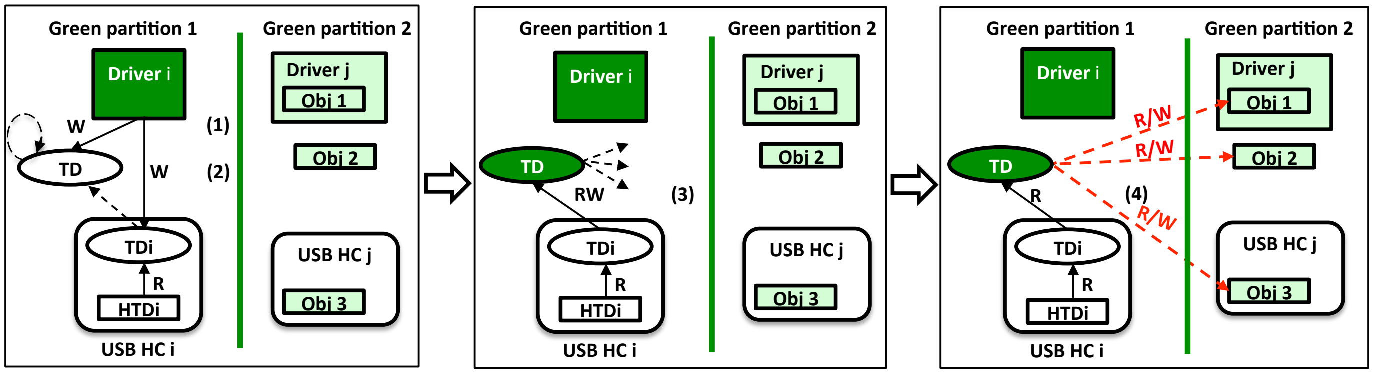


Fig. 8. **Wimpy kernel design vulnerability 2.**

At time (1), malicious driver i *writes* to an external TD, with a value defining 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* the external TD. WK 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), the USB HC device i *reads* the value of the external TD. Because driver i modified the TD and defined a write transfer to the TD itself at time (1), device i can then *write* new configuration values to this TD, under the control of malicious driver i. Next, device i *overwrites* the external TD thereby enabling transfers to driver j and device USB HC j. The WK 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 WK does *not* ensure that the values written by the device prevent indirect transfers.

At time (4), the USB HC i issues an *indirect* transfer to device USB HC j and/or driver j of partition 2 via the external TD, which violates the green-green partition separation. This transfer is similar to the Indirect Transfer 2 of Figure 2(b).

To remove this vulnerability, the sound WK design specs prevent all direct TD *write* transfers by devices in their local green partition. Again, computing the TD state transitive closure would have removed this vulnerability. For performance concerns, the sound WK enforces a stronger and more conservative policy: WK design prevents all direct TD *write* transfers by devices in their local green partition. This type of check is not included in the original WK design [21].

VII. Generation of Formally Verified WK Code

We use Vale/Dafny [22] to specify and verify a new, sound Wimpy kernel (WK) implementation, following the general approach used in prior work [15]. Comparing the new implementation with the original, we identify additional implementation errors in the original code.

A. WK API Specifications and Verification

All the WK APIs are specified in Vale, using the specifications of 15 CPU instructions. To generate assembly code, we implement an assembly printer to print the abstract-syntax-tree (AST) of APIs generated by Vale into GNU assembly format. Then, a new WK implementation is achieved by replacing the APIs with the newly generated ones in the original WK code.

To verify the Wimpy kernel APIs, Vale specifications are converted into Dafny (by Vale); and we discharge proof obligations in Dafny, by a simulation (refinement) proof that connects the implementation to the design.

System State. The WK implementation defines the detailed machine state, including CPU register state and stack state, and refines the design specifications as follows:

Identifiers. The implementation must use device, driver, and object identifiers established on x86 platforms instead of abstract identifiers of the design specifications. For example, USB peripheral device identifiers use USB hierarchy addresses (i.e., USB bus ID, USB address, connected hub's USB address, and hub port) and I/O objects in memory use physical memory addresses; and PIO registers use PIO addresses.

Ephemeral USB Host Controller Objects. The implementation must use designated registers of ephemeral USB HCs to instantiate the abstract objects in the design.

USB Interface Data Structures (IDSes). The implementation must use the designated fields of USB IDSes, which map to abstract objects in the design.

We define mappings between the concrete implementation on the x86 platform and their abstract definitions in the design specifications.

Operations. WK implementation code defines 21 APIs and 23 direct I/O accesses, each of which maps to exactly one operation of the design specifications (Table III of Appendix B).

We specify direct *I/O accesses* of drivers and devices to physical memory and device registers of x86 platforms in Dafny to provide a complete list of read and write accesses that hardware can perform (Section IV-A). If we missed some access patterns, for instance, memory accesses by green drivers, we would have been able to prove an insecure kernel secure. WK code also includes auxiliary *foreign function interfaces* (FFIs) such as APIs provided by the underlying micro-hypervisor (Appendix C).

Axioms. WK APIs and direct I/O accesses rely on 107 axioms, 105 of which are independent of I/O hardware authorization and are used to reduce proof effort. A further break down is in Appendix C. Only the following two axioms define I/O hardware authorizations.

Ax1I If OS devices can only reference I/O objects outside the untrusted OS partition by memory addresses, then IOMMUs prevent these devices from accessing *any* objects outside that partition.

Ax2I Only USB IDS queue headers may define transfers to USB peripheral devices.

Ax1I holds because IOMMU authorizes all memory accesses by devices. Sound WK implementation of moving a device between red and green partitions invokes the mHV to flush the IOTLB, just as any OS kernel would context switch an IOMMU. Other cached entries of the mHV hardware authorization mechanisms are also invalidated to prevent stale access permissions from being used to breach isolation. Ax2I holds because queue-element transfer descriptors are always linked by a queue header [8].

State Invariants. The implementation maintains 88 state invariants, 36 of which are refinements of the state invariants of WK design. Among the rest of the 52 new invariants introduced in the WK implementation, 34 invariants refer to the CPU registers, stack, and global variables used by the WK APIs. These invariants define valid values that can be set in these registers and memory; e.g., the ESP register always points to WK's stack; global variables containing identifiers of isolated drivers' partitions cannot contain the identifier of the untrusted OS partition. 13 invariants define valid mappings between identifiers used in code and abstract identifiers used by design specifications. Another invariant ensures that the I/O objects of OS drivers and devices have valid memory regions. Only 4 invariants are related to the I/O hardware used by WK.

SI1I The physical address spaces of OS drivers and devices, isolated drivers, physical USB HCs, and WK code, stack, global variables must not overlap.

SI2I USB IDSes in WK code, I/O objects of isolated drivers, physical and ephemeral USB HCs, and USB peripheral devices must not have PIO addresses; and I/O objects of USB peripheral devices must not have memory addresses.

SI3I All ephemeral USB host controllers' interrupts must be disabled.

SI4I USB IDSes referenced by ephemeral USB HCs must be either a queue header (QH) or a queue-element transfer descriptor (QTD) of those IDSes.

Transition Constraints. In addition to the refinements of the WK-design transition constraints, the implementation enforces the following constraint.

TC1I All mappings between identifiers used in WK code and identifiers used in design specifications are not modified by APIs or direct I/O accesses.

The Vale/Dafny specifications of the WK implementation support I/O separation since their correspondence to the sound WK design specifications is already proven.

B. Vulnerabilities of the Original WK code

We re-examined the original WK code [21] and found that it fails to satisfy the Vale/Dafny specifications of the verified WK implementation in four areas. This shows that careful but informal implementation of the original kernel code can still lead to exploitable vulnerabilities without formal verification. The first three vulnerabilities arise from violations of implementation-specific state invariants and transition constraints, whereas the fourth arises from incomplete kernel mediation of IDS modifications by drivers. These are low-level implementation errors, unrelated to the design vulnerabilities of Section VI-B.

Vulnerability 1: Isolated application drivers can modify their devices' USB addresses. In the original WK implementation, a rogue isolated driver could configure its device to overlap its address with another isolated driver's USB device. Hence, the rogue driver could access the latter device and break I/O separation between isolated application drivers. This vulnerability violates TC1I.

Vulnerability 2: Physical and ephemeral USB HCs can have PIO addresses. This enables untrusted OS applications to access both the ephemeral USB HCs owned by isolated application drivers and their mapped physical USB HCs, despite selective authorization of all memory accesses. This vulnerability violates state invariant SI2I.

Vulnerability 3: WK fails to clear a physical USB HC's I/O objects when releasing it to untrusted OS/applications. The original kernel code fails to prevent unauthorized object reuse. Note that isolated drivers cannot clear these objects since they do not have direct I/O access to them, as these USB HCs are shared. Hence, I/O separation between untrusted OS/Apps and isolated drivers does not hold. This vulnerability violates the refinement of the concrete model's TC1c in the implementation.

Vulnerability 4: Isolated drivers can modify USB IDSes after verification by WK. This enables time-of-check-to-time-of-use attacks and violates I/O separation between isolated drivers, and between untrusted OS/applications and isolated drivers.

C. Current Limitations

This WK implementation is unoptimized: only one USB IDS can be used in one API invocation by green applications, whereas the original implementation takes multiple USB IDSes. For now, this implementation does not support USB device interrupts and assumes that WK disables them.

VIII. Discussion

A. Future Use for Other I/O Kernel Designs

Our I/O model can be instantiated to other I/O kernels. We informally discuss those shown by dashed lines in Figure 5.

The GPU separation kernel (GSK) [38] is a special I/O kernel that separates ephemeral GPUs to create separate screen areas for trusted and untrusted applications to coexist on the same display. This allows both applications to perform concurrent I/O operations securely on these GPUs. Informally, GSK design can be generated from the same concrete model used for Wimpy kernel (WK). In contrast to WK, GSK's ephemeral

GPUs can be active in green and red partitions at the same time, even though they map to the same physical GPU. GSK ensures the separation of ephemeral GPUs. More concretely, the state of GSK design needs one additional variable: the ID of the ephemeral GPU used by the untrusted OS. Ephemeral GPUs in isolated applications can be active at the same time, and thus concurrent I/O operations are allowed on the same physical GPUs. To satisfy the ephemeral device policy of the concrete model, GSK needs to maintain additional state invariants and transition constraints for ephemeral GPUs; e.g., ephemeral GPUs must be inactive when its mapped physical GPU is active. The informal analysis shows that the original GSK design fails to enforce I/O separation when the physical GPU shares the same PCI bus with a red device; i.e., it has the vulnerability 1 of the WK design. The solution is to enforce GSK use of only PCIe GPUs. GSK does not have vulnerability 2 of the WK design because isolated applications only need to provide display contents (data object values) and window geometry configurations (function descriptor values) without accessing any TDs.

Traditional separation kernels [17], [16] can also be formalized in our framework. However, they would have a different I/O separation policy from the concrete model for WK and GSK, and thus require a new concrete model. First, 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 authorize transfers issued by drivers. Isolated applications use these channels to communicate with I/O kernels in different partitions. Third, the allocation of devices to these I/O kernels is static and cannot be modified on demand, because devices cannot move between partitions. I/O kernels instantiating such a concrete model can enforce all policies as traditional separation kernels; in addition, they can support DMA accesses that were excluded by traditional separation kernels [44], [17].

Low-assurance OS kernels, such as the Linux kernel, can also fit into our framework. However, another concrete model is necessary to capture the intra-OS I/O separation policies that prevent devices from accessing non-DMA memory. This new model would have two I/O partitions, one comprising all active subjects and objects, and the other the NULL I/O partition for inactive ones. The kernel would ensure that Non-DMA memory areas contain only inactive I/O objects (i.e., unmapped in DMA memory), and thus active devices cannot access them. Unlike previous I/O separation designs [11], the new concrete-model instantiation in OS kernels could support device P2P communications, with early authorization.

B. Lessons Learned

Several practical lessons arise from the process of designing and applying the models to the verification of I/O kernels. Three of these lessons are summarized below.

First, the application of our models to systems with inadequate I/O hardware clearly shows the significant extra verification cost and limited scalability of these systems; see Section III. This suggests that inadequate I/O hardware should be deprecated for use in large commodity systems.

Second, formal design and implementation force rigorous reasoning beyond careful but informal development. Without formal specifications and verification, security assurance is difficult to obtain even for small I/O kernels (e.g., fewer than 4K SLoC) that use the best I/O hardware. Subtle I/O kernel design and implementation vulnerabilities can be easily missed by informal development; viz., Sections VI-B and VII-B. Also, simply using the best I/O authorization hardware to separate address spaces is insufficient; e.g., separation of ephemeral devices (viz., Section V-A), which are multiplexed on the same physical device, must be formally verified.

Third, the performance of formally verified I/O separation need not incur any penalty beyond that of secure IOMMU context switches in existent OS kernels [11]. Just the opposite: use of *non-shared*, de-privileged drivers and buffers in isolated applications, which don't rely on *shared* buffer pools in OS kernels, *and* of small and simple, dedicated I/O kernels naturally offer added performance benefits [21]. In contrast, substantial performance improvements that preserve intra-OS kernel I/O separation by *selective authorization* can still incur up to 25% throughput overhead and 20% increased CPU utilization by using shared buffer pools [11].

IX. Related Work

Low assurance and limited I/O functions.. Device virtualization [3], [45] can support separate I/O transfers to different virtual machines. However, this yields a much larger trusted code base hence lowers assurance significantly. To minimize trusted code bases, some isolation kernels [46], and micro-hypervisors [47], [19], [48] support I/O separation for only a limited set of I/O devices and functions. Other micro-kernels (e.g., Nexus RVM [2]) are incompatible with commodity OSes. They fail to authorize P2P device transfers (viz. the vulnerability 2 in Section VI-B) and support multiple-device broadcasts on a bus controller. Finally, since no commodity I/O devices can encrypt/decrypt traffic with secret keys of crypto enclaves, they cannot transfer to/from encrypted memory; i.e., SGX [41]. Instead, they separate I/O transfers to isolated drivers, which establish crypto channels with enclaves [49].

Verified kernels. The SELinux security kernel [50] enforces MAC policies but assumes that unverified kernel mechanisms support I/O separation. The seL4 micro-kernel can ensure static I/O separation when it implements a separation kernel [51], as described in Section VIII-A. The ExpressOS [52] micro-kernel provides formally verified application security properties without trusting system services. However, it does not enforce I/O separation for isolated applications; e.g., an application can read I/O buffers of other applications and I/O separation for general devices is not enforced.

Acknowledgment

We thank the reviewers for their thoughtful comments and suggestions. The research reported herein was supported in part by the Department of the Navy, Office of Naval Research, under Grant No. N00014-17-1-2892.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *ACM SOSP*, 1995.

[2] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider, "Device driver safety through a reference validation mechanism," in *USENIX OSDI*, 2008.

[3] L. Xia, J. Lange, P. Dinda, and C. Bae, "Investigating virtual passthrough i/o on commodity devices," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 83–94, Jul. 2009.

[4] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *USENIX ATC*, 2010.

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

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

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

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

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

[10] F. L. Sang, É. Lacombe, V. Nicomette, and Y. Deswarte, "Exploiting an I/OMMU vulnerability," in *Proc. Int. Conf. on Malicious and Unwanted Software, MALWARE*, 2010, pp. 7–14.

[11] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *ASPLOS*, 2016, pp. 249–262.

[12] 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 *ACM SOSP*, 2009, pp. 207–220.

[13] 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 *IEEE S&P*, 2013, pp. 430–444.

[14] 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 *USENIX Security*, 2016, pp. 87–104.

[15] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *ACM SOSP*, 2017, pp. 287–305.

[16] I. GreenHills Software, "Integrity-178b separation kernel security target," https://www.commoncriteriaportal.org/files/epfiles/st_vid10362-st.pdf [Accessed on 2 Dec 2020], 2010.

[17] J. M. Rushby, "Separation and integration in MILS (The MILS Constitution)," in *Technical Report SRI-CSL-08-XX*, February 2008.

[18] 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 *ACM SOSP*, 2011, pp. 189–202.

[19] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *IEEE S&P*, 2012.

[20] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR'10*, 2010, pp. 348–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939141.1939161

[21] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated I/O," in *IEEE S&P*, 2014, pp. 308–323.

[22] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *USENIX Security*, 2017, pp. 917–934.

[23] R. Achermann, N. Hossle, L. Humbel, D. Schwyn, D. Cock, and T. Roscoe, "A least-privilege memory protection model for modern hardware," 2019.

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

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

[26] 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 *IEEE S&P*, 2010, pp. 447–462.

[27] ARM, "ARM AMBA 5 AHB Protocol Specification," 2015.

[28] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU scalably," in *USENIX ATC*, 2015, pp. 549–562.

[29] 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 *ACM ACSAC*, 2013.

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

[31] 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 *NDSS*, 2019.

[32] J. Applebaum, J. Horchert, and C. Stocker, "Catalog Reveals NSA Has Back Doors for Numerous Devices," 2013. [Online]. Available: https://www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html

[33] S. Anthony, "Massive, undetectable security flaw in USB: It's time to get your PS/2 keyboard out of the cupboard," *Extreme Tech*, no. July 31, 2014. [Online]. Available: https://www.extremetech.com

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

[35] L. Constantin, "What is a "Supply Chain Attack?"," in *Motherboard*, Sept. 2017. [Online]. Available: https://motherboard.vice.com/en_us/article/d3y48v/what-is-a-supply-chain-attack

[36] V. Gligor and M. Woo, "Establishing software root of trust unconditionally," in *NDSS*, 2019.

[37] Trusted Computing Group, "Hardware Requirements for a Device Identifier Composition Engine," 2018. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf

[38] M. Yu, V. D. Gligor, and Z. Zhou, "Trusted display on untrusted commodity platforms," in *ACM CCS*, 2015, pp. 989–1003.

[39] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems," in *IEEE S&P*, May 2020.

[40] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *IEEE S&P*, 2010.

[41] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016, https://eprint.iacr.org/2016/086.

[42] Wikipedia, "CAN bus," 2019. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus

[43] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SOSP*, 2003.

[44] J. M. Rushby, "Design and verification of secure systems," vol. 15, no. 5, pp. 12–21, 1981.

[45] K. Tian, Y. Dong, and D. Cowperthwaite, "A full GPU virtualization solution with mediated pass-through," in *USENIX ATC*, 2014, pp. 121–132.

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

[47] Y. Cheng and X. Ding, "Guardian: Hypervisor as security foothold for personal computers," in *TRUST*, 2013.

[48] Z. Zhou, J. Han, Y.-H. Lin, A. Perrig, and V. Gligor, "Kiss: Key it simple and secure corporate key management," in *TRUST*, 2013.

[49] S. Weiser and M. Werner, "SGXIO: Generic trusted i/o path for Intel SGX," in *CODASPY*, 2017, pp. 261–268.

[50] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the linux operating system," in *USENIX ATC*, 2001, pp. 29–42.

[51] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "Sel4: From general purpose to a proof of information flow enforcement," in *IEEE S&P*, 2013, pp. 415–429.

[52] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in ExpressOS," in *ASPLOS*, 2013, pp. 293–304.

Appendix A

A. Operations for I/O Separation Model

Each operation in the I/O separation model takes the current state k and the operation 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 completes successfully; when d is false, the operation is denied and $k' = k$. We present 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 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 any active device in k_{new} must be in the same partition as the device and must not include a hardcoded TD.

(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 device's object only, or after appropriate driver *write* operations, which have already been allowed by the I/O kernel; see Section IV-B1.

(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.

(7, 8, 9) Driver/Device/External Objects Activation: $DrvActivate(drv\_id, pid)$, $DevActivate(dev\_id, pid)$, and $ExternalObjsActivate(obj\_ids,\ pid)$. These operations activate the given subject and external objects into the partition with ID pid. The I/O kernel checks that in the current state: (1) The given subject and external objects must be inactive, and (2) the partition pid exists in the set of non-NULL partition IDs. Then, to fulfill the security property SP2, the kernel clears the subject's objects and all external objects to prevent object reuse between partitions, but not modifying any given device's (immutable) hardcoded TD. Finally, the given subject's partition ID, its objects' partition ID and given external objects' partition ID are updated to pid.

(10, 11, 12) Driver/Device/External Objects Deactivation: $DrvDeactivate(drv\_id)$, $DevDeactivate(dev\_id)$, and $ExternalObjsDeactivate(obj\_ids, pid)$. These operations deactivate the given subject and external objects. The I/O kernel first checks that they are active. Then the I/O kernel computes the transitive closure (tc) of the *TD_state* in the current system state. The kernel checks tc to ensure that no active device can issue transfers to any objects of the given subject, nor to the given external objects. On success checks, the kernel updates the partition ID of the given subject and its objects to be NULL.

B. Dafny Specification Example

Simplified Specifications of Operations. Figure 9 presents a simplified Dafny specification of the Device Write operation. The operation takes the current state k, the ID of the device issuing the write transfer, and 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 fulfilling 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 to apply a single transition; that is, an operation on the state k. As shown in Figure 10, the function takes a state 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 secure; i.e., fulfilling all state invariants, because all operations taking a secure state ends up at a secure state *only*. The function further requires operations' preconditions always imply operations' postconditions, and 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.

```
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
    ...
}
```

Fig. 9. **Simplified Device Write Operation in Dafny**

The body of the K_CALCNEWSTATE function returns an arbitrary state fulfilling the post-conditions 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 the theorem 1, 2 and Corollary 1, as long as the result state k' fulfills the postconditions of operations. The K_CALCNEWSTATE function is defined as Dafny function to enable its use in the proof of the theorems and the corollary.

```
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)
}
```

Fig. 10. **K_CALCNEWSTATE: example of a state transition**

C. Axioms, State Invariants and Transition Constraints for I/O Separation Model

Axioms. To prove that the I/O separation model is secure we use 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 IV-B1).
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. The I/O separation model defines the following state invariants for a secure system state:

1. Drivers and devices must have different subject IDs.
2. The set of subjects must not be empty.
3. TDs, FDs and DOs must have different object IDs.
4. The set of objects must not be empty.
5. Each device's TDs must include its hardcoded TD.
6. No two subjects associate (own) the same object.
7. Objects associated with any subjects must exist in the system states' objects.
8. No hardcoded TDs define direct transfers to a TD with both read and write access modes.
9. Hardcoded TDs do not reference any hardcoded TDs.
10. Objects referenced in a device's hardcoded TD must be associated with the device.
11. Arbitrary set of TDs in the system state have finite ranges.
12. Only hardcoded TDs and active objects have values.
13. The partition IDs of the system state do not include NULL.
14. (SI1) For any *TD_state* of a transitive closure in a system state, 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.
15. All objects associated with a subject must be in the same partition with the subject.
16. Active subjects and objects must belong to existing partitions.

Transition Constraints. The I/O separation model defines the following transition constraints:

1. IDs of objects and hardcoded TDs associated with subjects must be immutable in transitions.
2. (TC1) Only hardcoded TDs can be reused in a new partition with non-NULL partition IDs.
3. Hardcoded TDs' values must be immutable in transitions.

D. Computing Transitive closure

Let a *TD_state* contains the values of all active TDs in the current state. The *transitive closure* of a *TD_state* is the set of all reachable TD_states from that state via TD writes. The implementation uses two mutually recursive functions. The first function discovers all direct TD *writes* that can be issued by a device in one TD state, via 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. The second function uses the output of the first function, constructs and traverses

TABLE I
Operation mapping: concrete to I/O separation model

Operations in I/O separataion model	**Operations in concrete model**
DrvWrite	DM_RedDrvWrite
	DM_GreenDrvWrite
DevWrite	DM_RedDevWrite
	DM_GreenDevWrite
DrvRead	DM_RedDrvRead
	DM_GreenDrvRead
DevRead	DM_DevRead
EmptyPartitionCreate	DM_EmptyPartitionCreate
EmptyPartitionDestroy	DM_EmptyPartitionDestroy
DrvDeactivate	DM_GreenDrvDeactivate
DevDeactivate	DM_DevDeactivate
DrvActivate	DM_DrvActivateToGreenPartition
	DM_DrvActivateToRedPartition
DevActivate	DM_DevActivate
ExternalObjsActivate	DM_ExternalObjsActivateToGreenPartition
	DM_ExternalObjsActivateToRedPartition
ExternalObjsDeactivate	DM_GreenExternalObjsDeactivate
	DM_RedExternalObjsDeactivate

TABLE II
WK operations as concatenations of concrete operations.

Operations in concrete model	**Operations in (correct) WK design**
DM_RedDrvRead	WSD_OSDrvRead
DM_GreenDrvRead	WSD_WimpDrvRead
DM_DevRead	WSD_DevRead
DM_RedDrvWrite	WSD_OSDrvWrite
DM_GreenDrvWrite	WSD_WimpDrvWrite
DM_RedDevWrite	WSD_OSDevWrite
DM_GreenDevWrite	WSD_WimpDevWrite
DM_EmptyPartitionCreate	WKD_EmptyPartitionCreate
DM_EmptyPartitionDestroy	WKD_EmptyPartitionDestroy
DM_DevActivate	WSD_DevActivate
DM_DevDeactivate	WSD_DevDeactivate
DM_DrvActivateToGreenPartition	WKD_DrvActivateToGreenPartition
DM_GreenDrvDeactivate	WKD_GreenDrvDeactivate
DM_ExternalObjsActivate ToGreenPartition	WKD_ExternalObjsActivate ToGreenPartition
DM_ExternalObjsActivate ToRedPartition	WKD_ExternalObjsActivate ToRedPartition
DM_GreenExternalObjsDeactivate	WKD_GreenExternalObjsDeactivate
DM_RedExternalObjsDeactivate	WKD_RedExternalObjsDeactivate
DM_DevActivate || DM_DevActivate ...	WKD_MultiDevs_ReturnOS

a graph of *TD_states* with the BFS algorithm, and outputs all potential states that enable I/O transfers.

The transitive-closure computation always terminates since the set of TDs is finite, and each TD has a finite set of values. It terminates even when TD graphs are cyclic. Thus, the number of all possible *TD_states* is finite.

Appendix B

Operation Refinements. Table I shows the operation mapping from the concrete model to the I/O separation model, while Table II illustrates the operations of the correct Wimpy kernel as concatenations of concrete model operations. Table III shows the operation mapping from the sound Wimpy kernel design to its implementation specifications.

Appendix C

FFIs for WK Implementation A breakdown of the FFIs that WK implemented is as follows. 9 of these FFIs invoke APIs provided by the underlying *micro-hypervisor* (mHV)

TABLE III
Operation mapping: WK design to implementation. ("/" denotes operations with the same prefix or suffix. And WSD_DevWrite wraps both WSD_OSDevWrite and WSD_WimpDevWrite, and is for all active devices.)

Operations in (correct) WK design	**WK APIs**
WKD_EmptyPartition Create/Destroy	WK_EmptyPartition Create/Destroy
WKD_DrvActivateToGreenPartition	WimpDrv_Activate
WKD_GreenDrvDeactivate	WimpDrv_Deactivate
WSD_DevActivate/Deactivate	USBPDev_Activate/Deactivate
	EEHCI_Activate/Deactivate
WKD_ExternalObjsActivateToGreenPartition	USBTD_slot_allocate_1slot
WKD_GreenExternalObjsDeactivate	USBTD_slot_deallocate_1slot
WSD_WimpDrvWrite	USBTD_slot_submit_ and_verify_qtd32/qh32
WKD_MultiDevs_ReturnOS	OS_Activate_AllReleasedPEHCIsAndUSBPDevs
WKD_ExternalObjsActivateToRedPartition	OS_Activate_MainMem_ByPAddr
WKD_RedExternalObjsDeactivate	OS_Deactivate_MainMem_ByPAddr
WSD_WimpDrvWrite	WimpDrv_Write_eEHCI_Config/Status/USBTDReg
WSD_WimpDrvRead	WimpDrv_Read_eEHCI_Config/Status/USBTDReg

Operations in (correct) WK design	**Direct I/O Accesses**
WSD_OSDrvRead	WSM_OSDrvRead_ByPAddr/PIO/ObjIDs
WSD_DevRead	WSM_OSDevRead_ByPAddr/PIO
	WSM_OSNonUSBPDevRead_ByObjIDs
WSD_OSDrvWrite	WSM_OSDrvWrite_ByPAddr/PIO/ObjIDs
WSD_OSDevWrite	WSM_OSDevWrite_ByPAddr/PIO
	WSM_OSNonUSBPDevWrite_ByObjIDs
WSD_WimpDrvRead/Write	WSM_WimpDrvRead/Write_ByPAddr
WSD_DevRead/Write*	WSM_USBPDevRead/Write_ByObjID
WSD_DevRead	WSM_EEHCIReadOwnObjs_ByOffset
	WSM_EEHCIReadUSBTD_BySlotID
	WSM_EEHCIReadUSBPDevObj_ByObjID
	WSM_EEHCIReadObjs_ByPAddr
WSD_WimpDevWrite	WSM_EEHCIWriteOwnDO_ByOffset
	WSM_EEHCIWriteUSBPDevObj_ByObjID
	WSM_EEHCIWriteObjs_ByPAddr

and by internal code used by the WK APIs. That is, 4 APIs are provided by mHV to move main memory areas between OS/Apps and isolated drivers on-demand, and 3 APIs are provided by internal WK code that creates, destroys, and separates ephemeral USB HCs on demand. Another API implements the clearing of all mutable objects of USB peripheral devices. The last API enables isolated drivers to clear data objects (DOs) in their memory. Furthermore, WK code includes FFIs for internal functions that support different versions of USB buses; e.g., USB IDS memory moves, data structure parsing.

Axioms for WK Implementation A further breakdown of the 105 axioms is as follows. 37 axioms refer to the correctness of arithmetic and bit-oriented operations. 50 axioms assert that the mappings implemented object values and operations to those of the WK design are valid; e.g., the mapping of I/O access parameters to abstract object identifiers and values of design specifications is formatted properly. These axioms are valid because the mappings from WK implementation specifications to those of WK design have easily checkable formats. 9 axioms refer to trivial hardware properties of OS devices, USB host controllers, peripheral devices and IDSes, and to the memory properties of device drivers; e.g., they constrain valid I/O objects values, and IDSes defining transfers to a USB device always define transfers to all its FDs and DOs. 7 axioms state that the memory layout of WK code, stack, and globals are valid. One axiom asserts the validity of ID mappings of USB TDs. Another axiom asserts that I/O accesses and WK APIs are atomic, and is typical of state-transition models. (The WK implementation uses additional 15 Dafny *assume* statements for trivial utility lemmas for similar proof simplification purpose.)