2014 IEEE Symposium on Security and Privacy

Dancing with Giants: Wimpy Kernels for On-demand Isolated I/O

Zongwei Zhou Miao Yu Virgil D. Gligor
ECE Department and CyLab, Carnegie Mellon University
{zongweiz, miaoy1, virgil}@andrew.cmu.edu

***Abstract*—To be trustworthy, security-sensitive applications must be formally verified and hence small and simple; i.e., wimpy. Thus, they cannot include a variety of basic services available only in large and untrustworthy commodity systems; i.e., in giants. Hence, wimps must securely compose with giants to survive on commodity systems; i.e., rely on giants' services but only after efficiently verifying their results. This paper presents a security architecture based on a *wimpy kernel* that provides *on-demand isolated I/O channels* for wimp applications, without bloating the underlying trusted computing base. The size and complexity of the wimpy kernel are minimized by safely outsourcing I/O subsystem functions to an untrusted commodity operating system and exporting driver and I/O subsystem code to wimp applications. Using the USB subsystem as a case study, this paper illustrates the dramatic reduction of wimpy-kernel size and complexity; e.g., over 99% of the USB code base is removed. Performance measurements indicate that the wimpy-kernel architecture exhibits the desired execution efficiency.**

I. Introduction

Modern architectures can isolate security-sensitive application code from the untrusted code of commodity platforms, enabling their safe co-existence [13, 14, 16, 31, 44, 47, 48, 50, 56, 60–62, 66]. This is necessary because large untrustworthy software components will certainly continue to exist in future commodity platforms. Competitive markets with low cost of entry, little regulation, and no liability will always produce innovative, attractively priced, *large* software systems comprising diverse-origin components with uncertain security properties. As Lampson metaphorically put it a decade ago, *among software components, only the giants survive* [41]. Thus, the best one can hope for is that some trustworthy software components can be protected from attacks launched by adversary-controlled *giants.* To be trustworthy, software components must be verified, and to be verified they must be comparatively small, simple, and limited in function. In contrast to the giants, these software components are *wimps.*

Problem. Unfortunately, isolating security-sensitive wimps from untrusted giants does not guarantee wimps' survival on commodity platforms. To avoid re-creating giants inside their isolated execution environments, wimps often give up a variety of basic services for application development, which greatly undermines their usefulness and viability. For example, wimps typically lack persistent memory [52], file system and network services [13, 14, 16, 31, 44], flexible trusted paths to users [69], and I/O services needed for many applications; e.g., in industrial control, finance, health care, and defense.

Past multi-year efforts to restructure giants (e.g., commercial OSes) and provide trustworthy services for applications led to successful research [37, 58] but failed to deliver trustworthy OSes that met product compatibility and timeliness demands of competitive markets [25, 45]. The alternative of including basic services in the trusted computing bases (TCBs)[1] that guarantee safe giant-wimp co-existence has been equally unattractive. TCBs would lose assurance since they would become bloated, unstable, and unverifiable; i.e., they would use large and complex code bases of diverse, uncertain origin (e.g., device drivers) needed for different applications, and require frequent updates because of function additions, upgrades, and patches. Thus, the only remaining option is to place basic application services in the giants. To survive, wimps would have to rely on giant-provided services but only after efficiently verifying their results. In turn, wimps could make their own isolated services available to giants for protection against persistent threats. Continuing with the wimp-giant metaphor, trustworthy wimps must engage in a carefully choreographed dance (i.e., secure composition) with untrustworthy giants.

Among the basic services needed by wimps are on-demand isolated I/O channels to peripheral devices. Past attempts to provide such services with high assurance on commodity systems have been unsuccessful; viz., related work in Section VIII. Some provide isolated I/O channels within system TCBs [53, 69] but only for a few selected devices. Even limited support for few devices invariably increases the size and complexity of trusted code and undermines assurance. For example, including *only* the Linux USB bus subsystem in the XMHF micro-hypervisor[66] would more than double its code-base size and increase its complexity significantly; e.g., it would introduce concurrency in serial micro-hypervisor code since it would require I/O interrupt handling. Other attempts statically allocate selected peripheral devices to isolated system partitions [28, 38, 55, 64, 65] at the cost of losing on-demand (e.g., plug-and-play) capabilities of commodity systems. In contrast, other systems provide on-demand I/O capabilities by virtualizing devices or passing them through to isolated guest OSes, but sacrifice I/O channel isolation from the untrusted OSes [22, 46, 53, 60]. Further attempts to isolate I/O channels rely on special hardware devices equipped with data encryption capabilities [40] to establish cryptographic channels to applications [30, 49, 67]. This approach excludes commodity devices, which lack encryption capabilities, and adds TCB complexity by requiring secure key management for the special devices.

Solution. In this paper, we present a security architecture for *on-demand* isolated I/O channels, which enables security-

[1]TCBs include security [10, 57], micro [39, 59], and exo kernels [20], virtual machine monitors [11, 13, 14, 17, 22, 31], micro-hypervisors [47, 60–62, 65, 66, 69], and separation/isolation kernels [28, 53, 55, 64].

© 2014, Zongwei Zhou. Under license to IEEE.
DOI 10.1109/SP.2014.27

IEEE computer society

sensitive applications (i.e., wimps) to dynamically connect to diverse peripheral devices of unmodified commodity OSes. Central to on-demand isolation of I/O channels is the notion of the *wimpy kernel*, which constructs these channels without affecting the underlying TCB; i.e., without modifying its security properties and increasing the verification effort [39, 66]. The wimpy kernel is an add-on trustworthy service that is isolated from the untrusted OS by the underlying TCB. It executes at the OS's privilege level, mediates all accesses of wimp applications (*wimp apps*) to I/O devices, and prevents the untrusted OS from interfering with wimp apps' execution and I/O transfers, and vice-versa. The wimpy kernel removes a wimp app's direct interfaces to the underlying TCB. Thus, future I/O function innovation that enhances the untrusted OS or wimp apps would only affect the wimpy kernel, leaving the underlying TCB unchanged. Note that TCB used in this paper is a slightly modified version of XMHF [66] – a non-virtualizing micro-hypervisor whose memory integrity has been formally verified. We stress, however, that *any type of TCB* with similar isolation properties as XMHF could be used to support the wimpy kernel.

We minimize the size and complexity of the wimpy kernel to facilitate its formal verification, using two classic security engineering methods. First, we outsource I/O subsystem functions to the untrusted OS, but *only if* the wimpy kernel can verify that the execution of that code is correct. For example, the initialization and configuration of the entire USB controller-hub-device hierarchy is handed over to the wimpy kernel by the untrusted OS on demand. The wimpy kernel verifies the hierarchy without enumerating each device. Second, we further minimize the wimpy kernel by de-privileging and exporting drive and driver-subsystem code to wimp applications, and implementing wimpy-kernel checks that verify applications' use of the exported code. Exporting code requires identification and removal of all driver-code dependencies on the untrusted OS services (e.g., memory management, synchronization, kernel utility libraries), either because they become redundant in the new on-demand mode of operation or because they can be satisfied by the wimp apps or wimpy kernel. For example, synchronization functions that multiplex a device among different applications become redundant, since we already guarantee the isolation and exclusive ownership of devices to a wimp app during its execution.

Contributions. In short, we make the following contributions.

- We introduce the notion of on-demand isolated I/O channels for security-sensitive applications (i.e., for wimps) on unmodified commodity platforms (i.e., on giants).
- We present a security architecture based on a minimal wimpy kernel, which implements on-demand I/O isolation without affecting the underlying TCB.
- We illustrate how the classic outsource-and-verify and export-and-mediate methods are used to minimize the wimpy kernel, and report on the minimization results in detail; e.g., we remove over 99% of the Linux USB code from the wimpy kernel.
- We implement the wimpy kernel for the USB subsystem of Linux and evaluate its performance. Experimental results indicate that our on-demand I/O isolation system incurs acceptable performance overhead.

II. Problem Definition

This section outlines the advantages of the on-demand I/O channel isolation on commodity platforms in the wimp-giant model, describes the adversary model, presents the inherent challenges posed by on-demand channel isolation, and summarizes the security properties to be achieved.

A. On-demand Isolated I/O

In the on-demand I/O isolation model, the untrusted OS manages all commodity hardware resources and devices on the platform most of the time. However, when a security-sensitive application demands exclusive use of a device, the I/O isolation system takes control of necessary hardware communication resources from the untrusted commodity OS, verifies their OS configurations, and allocates them to the application. When the application is done with a channel, the system returns all resources used to the untrusted OS.

The on-demand I/O isolation model has four significant advantages. First, it enables wimp applications to obtain isolated I/O channels to any subset of a system's commodity devices needed during a session, not just to a few devices statically selected at system and application configuration [69]. Cryptographically enabled channels, device virtualization, or pass-through of hardware devices become unnecessary.

Second, it enables trusted audit and control of physical devices without stopping and restarting applications, since all devices can be time-shared between trusted and untrusted applications. This makes it possible to maintain control of physical devices in long-running applications on untrusted commodity OSes; e.g., industrial process control, air-traffic control, and defense.

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

Fourth, it offers a significant opportunity for the reduction of the trusted I/O kernel size and complexity, and hence for enhanced verifiability. That is, the kernel can outsource many of its I/O functions to an untrusted OS and use them whenever it can verify the results of the outsourced functions correctly and efficiently. This opportunity is unavailable in either the static device allocation or virtualization models. In the former the OS cannot configure devices in wimp partitions, and in the latter it does not have direct access to hardware devices.

B. Adversary Model

We adopt the typical adversary model of systems that support giant-wimp isolation. Thus, an adversary could compromise the untrusted commodity OS (i.e., the giant) and can control some of its hardware resources (e.g., physical memory, device I/O ports). The compromised OS can directly

attack wimp apps or intentionally control or mis-configure any device (e.g., modify a USB device's address), including the I/O devices that it hands over to wimp apps, on demand. Controlled or mis-configured devices may unwittingly perform arbitrary operations to breach a wimp app's I/O isolation, such as claiming USB transfers, and issuing Direct Memory Access (DMA) requests. In addition, a malicious or rogue wimp application may attempt to escalate its privilege by manipulating the interfaces with the I/O isolation system or configuring the wimp app's devices. It could also try to break application isolation (e.g., process isolation, file system controls), or even compromise OS execution and corrupt its data. We assume that all the chip-set hardware and peripheral devices do not contain Trojan-Horse hardware circuits, microcode or malicious firmware, which could violate the desired security properties. Side-channel and denial-of-service attacks against isolated I/O channels are also out of scope.

C. Security Challenges

In the giant-wimp isolation model, on-demand I/O channels offer ample opportunity for a giant to interfere with a wimp's I/O operation and compromise its secrecy and integrity. One faces three key challenges in providing such channels.

I/O Channel Interference. Given the fact that hardware resources and devices are dynamically shared by the giant (i.e., untrusted OS) and wimp applications on a time-multiplexed basis, the giant can mis-configure a device, or a transfer path to it, and compromise the secrecy and/or integrity of a wimp's I/O. For example, most devices are interconnected by diverse bus subsystems (e.g., PCI, USB, Bluetooth, HDMI) in modern I/O architectures [36], which now become exposed to subtle isolation attacks; viz., the USB address overlap attack and the remote wake-up attack of Section IV-B1. Hence, I/O channel isolation must now control the multiplexing of complex bus subsystems for different devices.

Mediation of Shared Access to Devices. Further opportunities for interference arise from on-demand I/O; e.g., a rogue wimp/giant may refuse to release the use of I/O resources shared with the giant/wimp (e.g., shared interrupts) after I/O completion. Although both wimps and giants must have time-bounded, exclusive access to shared I/O resources and devices, they must be unable to retain unilateral control over shared I/O resources beyond time bounds specified by mediation policies for device access.

Verifiable I/O Codebase. The opportunity for minimizing I/O kernel size and complexity created by the on-demand I/O isolation model (viz., Section II-A) poses a significant design question. That is, if outsourcing of I/O kernel functions to the untrusted OS is possible *only if* the results of the outsourced functions can be verified correctly and efficiently by the kernel, which functions can be outsourced? Answering this question is important, since the trusted code minimization can be dramatic, as illustrated in Section VII below.

Minimization of I/O kernel code base for verifiability reasons goes beyond the outsource-and-verify method. For example, device driver and bus subsystem code could be decomposed into modules that can be exported to applications, whenever the trusted I/O kernel can mediate the exported modules' access to I/O kernel functions and objects.

Finally, the composition of an trusted I/O kernel with the rest of the TCB must not diminish the existing assurance; i.e., must not invalidate the TCB's security properties and their proofs.

D. Security Properties

The security challenges described above indicate that the typical giant-wimp isolation model must be augmented with additional security properties. These specify how the trusted I/O kernel interacts with wimp applications, giant, and the underlying TCB to provide on-demand isolated I/O channels to peripheral devices. These properties are presented below.[2]

P1. I/O Channel Isolation. This property implies that both the giant and wimp applications cannot compromise the authenticity and secrecy of their I/O transfers, and wimps cannot compromise other wimps' transfers.

P2. Complete Mediation. This property implies that all time-multiplexed accesses of wimp applications to devices via shared I/O hardware resources and bus subsystem software must be mediated.

P3. Minimization of the Trusted Codebase. This property implies that the size and complexity of (1) the code base of a trusted I/O kernel must be minimized to facilitate formal verification; and (2) the underlying TCB must be unaffected by the addition of a trusted I/O kernel.

III. System Overview

To fulfill all three security properties of on-demand isolated I/O systems, we define an add-on security architecture based on a *wimpy kernel* (WK), which composes with the underlying TCB, the untrusted OS, and wimp applications. This section illustrates this architecture, and highlights the code base minimization methodology of the wimpy kernel.

A. Wimpy Kernel: An Add-on Trustworthy Component

As shown in Figure 1, the micro-hypervisor (*mHV*) – the underlying trusted code base of the I/O isolation system – runs at the highest privilege level, protects itself and provides typical isolated execution environment for wimp apps to defend against the untrusted OS and other applications (i.e., the giant). The micro-hypervisor is a trustworthy component "added on" to existing commodity OSes - not a native foundation built at OS inception [25]. The *mHV*, which is a slightly modified version of XMHF [66], implements the giant-wimp isolation model in the sense that it controls only the few hardware resources needed for its isolation, whereas the giant directly controls the remaining system chip-set hardware and peripheral devices.

[2]The similarity of these security properties to those of a traditional reference monitor [8] is *not* entirely accidental. However, achieving these properties for on-demand isolated I/O channels on a commodity OS is a vastly more challenging exercise than building a reference monitor for non-I/O objects from scratch.

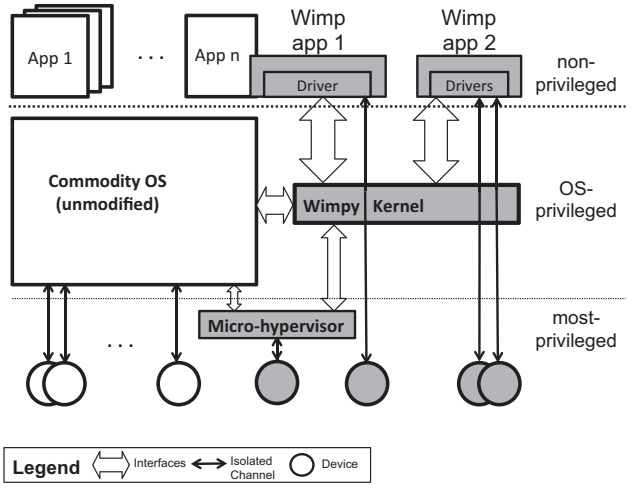


Figure 1. Overview of the I/O isolation architecture. The grey area represents the trusted code base of wimp applications.

The *wimpy kernel* is also an add-on trustworthy component, which is isolated from untrusted OS by *mHV*. It executes at the OS's privilege level, dynamically controls hardware resources necessary to establish isolated I/O channels between wimp apps and I/O devices, and prevents the untrusted OS from interfering with these channels and vice-versa; viz., Property 1 of Section II-D. *mHV* maps the wimpy kernel into the address space of each supported wimp app to facilitate efficient communication between wimp apps and the wimpy kernel. The wimpy kernel leverages typical system techniques, such as CPU rings and guest page table permissions, to protect itself from the non-privileged wimp apps. The wimp apps incorporate modified, unprivileged device drivers to communicate with the isolated I/O devices, under the mediation of the wimpy kernel. The *mHV*, wimpy kernel, and wimp app interactions for channel isolation are described in Section V.

Figure 1 shows that the wimpy kernel must compose with three other system components. First, it must compose with the underlying micro-hypervisor, *mHV*. The key goal of this composition is to retain the stable and formally verified properties of *mHV*, as required by Property P3 (part 2); e.g., memory integrity and address space separation [66]. Second, it must compose with the untrusted OS (giant) since the wimpy kernel outsources its most complex functions to the untrusted OS, whenever it can efficiently verify their results, if its code base is to be small and simple; viz., Property 3 (part 1). Third, it must compose with wimp apps. This is because the minimization of its code base suggests that it should de-privilege and export some of its code (e.g., drivers) to wimp applications whenever it can mediate all accesses of the exported code to I/O devices and channels under its control; viz., Property 2.

B. Composition with mHV

The composition of the wimpy kernel with *mHV* has three important goals: it preserves *mHV*'s wimp-giant isolation model; it avoids addition of new abstractions to *mHV*; and it retains the verifiability of *mHV* and its security proofs. First, the wimpy kernel does not add any security primitives or services to the underlying *mHV* beyond those already required by the typical wimp-giant isolation model, which include physical memory access control [7, 33], device Direct Memory Access (DMA) control [6, 32], and sealed storage and attestation root-of-trust [29].

Second, the wimpy kernel does not require any new abstractions beyond wimp registration/un-registration, which are already offered to the untrusted OS for wimp-giant isolation. These services rely on separation of wimp and untrusted OS address spaces and physical memory, and preserve the memory isolation semantics of *mHV*.

Third, the wimpy kernel does not invalidate *mHV*'s security properties and their proofs. For example, it does not add services and primitives that support I/O channels or virtualization. I/O channels include memory mapping operations that directly affect address-space separation and memory protection proofs, and interrupt processing that greatly complicates those proofs due to added concurrency. Hence, interrupt processing must completely bypass *mHV* and dynamically select handling procedures located in either the untrusted OS or WK, depending on which system component controls the device at the time.[3]

Satisfying these composition goals preserves the simplicity and stability of the underlying micro-hypervisor. With XMHF as its foundation, *mHV* continues to remain much simpler than all past virtualizing hypervisors/VMMs and recent micro-hypervisor designs [15, 47, 60–62].

C. Composition with the Untrusted OS and Wimp Apps

To assure the I/O channel isolation, wimpy kernel needs to control all I/O hardware that is shared by wimp devices with devices of the untrusted OS or another wimp. For example, the USB device of a wimp app could share the USB host controller and hubs with untrusted-OS-controlled devices using this controller. However, to include all OS code that ordinarily controls shared I/O hardware in the wimpy kernel would bloat its code base and substantially increase its verification effort.

To minimize the code base size and complexity of the wimpy kernel, we apply two classic methods of trustworthy system engineering, namely outsource-and-verify functions (whose various *cryptographic* versions have been used since the late '70s [13, 14, 16, 19, 24, 26, 27, 31, 44]) and export-and-mediate code [35, 37, 58]. However, neither method has been used for high-assurance, on-demand I/O isolation kernels for commodity platforms before. I/O isolation was either in security kernels for a few simple devices and not on demand, or was outside security kernels and not minimized for high assurance; viz., related work in Section VIII-A. We achieve significant code base reduction results using these two methods; i.e., we manage to cut down over 99% of Linux USB code from the wimpy kernel, as shown in Section VII-A.

[3]Wimpy kernel uses similar mechanisms to those of references [22, 69] to isolate the interrupts of OS- and wimp-app-controlled devices and bypass *mHV*.

Outsource-and-Verify. We decompose the bus subsystem functions, outsource them to the untrusted OS, and then efficiently verify the results of those functions; viz., Figure 2. For example, the untrusted OS initializes the USB hierarchy, which includes the USB host controller, hubs and devices, and configures the I/O channels for a specific wimp device, whereas the wimpy kernel verifies their correct configuration and initialization. Without verification, the untrusted OS could intentionally mis-configure the shared USB host controller and hubs, and violate I/O channel isolation in an undetectable manner; viz., the *USB address overlap* and *remote wake-up* attacks in Section IV-B1. The verification code is much smaller and simpler than the bus subsystem code and various device drivers left in the untrusted OS, and relies only on generic host controller and hub operations, instead of the device-specific ones. In short, the outsource-and-verify approach enables us to substantially decrease the code base of the wimpy kernel and, at the same time, avoid reliance on the untrusted OS.

Export-and-Mediate. The wimpy kernel code base is further minimized by exporting device drivers and bus subsystem code to isolated wimp applications,[4] which would otherwise have to be supported in the wimpy kernel itself; e.g., the Bus Subsystem Stub of Figure 2 denotes bus subsystem code exported by the wimpy kernel to a wimp app. In Section IV-B2, we illustrate how to export bus subsystem code using USB as an example. In particular, we show how different transfer descriptors for USB transactions are created for wimp apps, and how the wimpy kernel mediates the wimp-app's use of these descriptors by checking the validity of a few isolation-relevant descriptor fields.

To export device driver and bus subsystem code to wimp apps, the wimpy kernel must identify and remove all code dependencies on the untrusted OS. To do this, the wimpy kernel de-privileges the driver support code (e.g., memory management, kernel utility libraries) and mediates the wimp apps' use of it, whenever necessary; viz. Figure 2. Some code dependencies, such as those of synchronization functions for device multiplexing, disappear in the on-demand I/O model and, while they no longer require de-privileging before export, they still require mediation after export. We illustrate how wimpy kernel performs driver support code exporting in Section IV-C.

Efficient Wimp-OS Communication. The wimpy kernel implements low-level communication primitives between wimp applications and the untrusted OS, which are compatible with the unmodified OS; i.e., the OS is neither redesigned nor recompiled. At run time, wimp apps can invoke untrusted OS services, such as file-system and networking services, whose results they can verify efficiently; e.g., using typical cryptographic functions. These primitives are highly efficient because they use Interprocessor Interrupts [7, 33] and shared memory and avoid heavy-weight context switches with *mHV*. We describe the design of the wimp-OS communication in Section IV-D, and illustrate its performance in Section VII-B.

[4]Wimp apps can also outsource-and-verify driver functions (e.g., device initialization, power management) to the OS, and reduce their size and complexity.

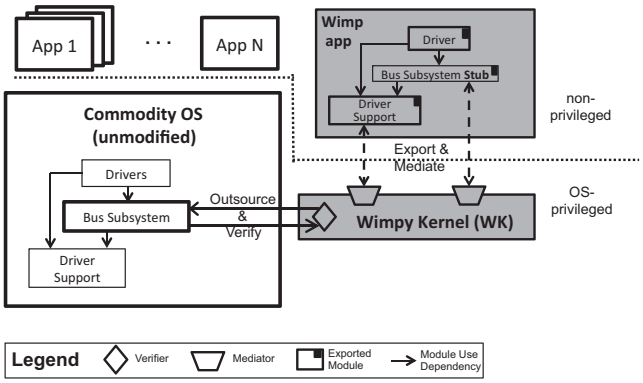


Figure 2. Outsourced functions and exported code of the Wimpy Kernel.

IV. Wimpy-Kernel Design

In this section, we define the scope of the wimpy kernel (WK) design (Section IV-A), present the design methods (Sections IV-B – IV-C), and describe the wimp-OS communication service provided by the wimpy kernel (Section IV-D).

A. Scope

Why Character-Oriented Devices? The design of the wimpy kernel focuses on character-oriented I/O devices for two reasons. First, these devices are pervasive (e.g., their drivers constitute about 52% of all Linux drivers [36]) and the isolation of their channels is more complex than for storage and network I/O devices. Second, the wimpy kernel need not support any storage or network I/O device functions. The reason is that the wimp apps can safely outsource these functions to the untrusted OS and retain their wimpy size and complexity. Wimp apps can do this very efficiently using cryptographic outsource-and-verify techniques [19, 26, 27], whereby they use either authenticated-encryption or MAC modes[5] to checksum and protect the integrity, and when necessary confidentiality, of the objects outsourced to untrusted OS services; e.g., files, databases, emails and other messages.

Why USB Subsystem? We chose the USB subsystem to illustrate the wimpy kernel code minimization method for three reasons. First, the USB bus is very popular in terms of device connectivity. For example, in Linux, 35% of device drivers use USB and 36% PCI; 10% of higher-level protocol drivers use either [36]. Second, we've already used similar code minimization methods for the PCI, and already reported them in the context of trusted path isolation [69]. We also illustrated the export-and-mediate method for simple character-oriented devices such as the PS/2 and VGA, which directly access low-level I/O resources; e.g., the I/O ports and MMIO memory. Third, channel isolation for the USB subsystem is the most complex since it mixes control and data channels, and uses (untrusted) software to maintain the device hierarchy and initialize device addresses (in versions earlier than USB 3.0). In contrast, channel isolation for all other subsystems (e.g.,

[5]An isolated subsystem for key distribution is presented by Zhou *et al.* [70].

PCI) is much simpler. For example, they already have separate control channels: some (e.g., PCI, Firewire) store hierarchy information in hardware, and others (e.g., Bluetooth and HDMI) have hardware-assigned device addresses. In neither case can untrusted OS code modify these channel control components.

Is the Design General? The minimization of wimpy kernel code requires modular decomposition and our design relies on traditional decomposition methods for I/O kernel code; viz., Figure 2. The outsource-and-verify method, which we illustrate with the USB subsystem (Section IV-B), applies to all other bus subsystems with similar code size and complexity minimization results. This is the case because device initialization and configuration functions, which we outsource to the untrusted OS, comprise about 51% of driver code on average [36]. Verification algorithms for the outsourced results are much simpler for all other subsystems (e.g., PCI, Firewire) than for the USB. For example, the verification algorithm for PCI bus is able to collect hierarchy information directly from the hardware registers of PCI bridges without having to derive it. For the Firewire bus, all bus bridges store routing information on how to reach a specific device, which can be directly accessed by the verification algorithm. In addition, for power management code (7.4% of driver code on average), verifying the power state of bus controller and hubs/bridges are general to any bus subsystem, because they comply with the widely accepted ACPI standard.

Our export-and-mediate method follows classic trustworthy-system engineering principles (mentioned above). Although the security-sensitive operations may differ for different bus subsystems and devices, their identification is well understood. In the on-demand I/O isolation model, we identify all operations which, if misused by malicious or compromised wimp apps, could violate the isolation I/O channels belonging to other wimp apps or to the OS. The mediation code of the wimpy kernel verifies that wimp-app operations do not cross the isolation boundary of low-level I/O resources allocated to wimp app devices and is used by all devices and bus subsystems. For example, the wimpy kernel performs simple range checks to ensure that a wimp app's operations only touch its own I/O ports, MMIO memory, and DMA memory. Mediation code also validates interrupt settings by comparing the interrupt vector, which is set by wimp apps, with others set by the untrusted OS. The wimpy kernel need *not* mediate wimp app operations that affect functional properties or availability of the isolated devices, which are more likely to have complex semantics of specific devices or buses. In addition, the method used to export driver-support code (e.g., low-level I/O, memory management, synchronization) to wimp apps (Section IV-C) applies to all devices and buses. However, drivers for different types of devices and buses may have different dependencies on support code.

B. Decomposing the USB Bus Subsystem

The Linux USB bus subsystem implements a variety of I/O functions such as bus enumeration, power management, device-information bookkeeping and the virtual file system (VFS) presentation to user-level application, device hot-plug, and request handling. We apply the outsource-and-verify and the export-and-mediate approaches to decompose this subsystem and include only necessary code in the wimpy kernel. The results are summarized in Table I.

Table I
DECOMPOSITION OF THE LINUX USB BUS SUBSYSTEM.

Code Modules	Design Decisions
Bus enumeration	Outsourced to OS
Power Management	Outsourced to OS
Information & VFS	Removed
Device hot-plug	Removed
Request handling	Exported to wimp apps

Specifically, we outsource the USB bus enumeration function to the OS, and design a simple and efficient verification algorithm in the wimpy kernel to verify the OS's configuration of the USB bus hierarchy (Section IV-B1). We also outsource power management functions of the USB host controller and hubs to the OS, since the wimpy kernel can efficiently verify the power status and prevent the OS from selectively disabling the bus hierarchy and compromising I/O data integrity of wimps. In contrast, device information bookkeeping and virtual file system services become unnecessary, because the wimpy kernel manages only a few devices for wimp apps on-demand. Instead, user-level wimp apps include the device drivers and directly access their devices, without any file-system representation. Also, the device hot-plug is excluded from the wimpy kernel, because it is not applicable to the on-demand I/O isolation model. Finally, the wimpy kernel exports the USB request handling code, which sets up USB transfer descriptors according to the requests from USB device drivers, to the wimp apps. However, the wimpy kernel verifies a few fields in the wimp-app-generated descriptors to ensure that the wimp apps' use of device I/O resources does not violate I/O channel isolation (Section IV-B2).

1) Verifying the Outsourced USB Bus Enumeration: To motivate the need to verify the USB device hierarchy whose management is outsourced to an untrusted OS, we briefly illustrate two attacks in which the compromised OS can breach the I/O data secrecy and integrity of wimp apps. Then we present the algorithm to defend against these attacks and verify the OS-initialized USB hierarchy. The concrete implementation of this algorithm is described in Section VI-B1.

Address Overlap Attack. A compromised OS can intentionally create duplicate addresses for various devices or hubs in the USB hierarchy, as is shown in Figure 3. The ultimate purpose of this type of device mis-configuration is to surreptitiously compromise the wimp I/O data, as illustrated below.

A device with a duplicate USB address can hide from the WK during hierarchy verification, if it responds to control transfers from the WK (e.g., reading device descriptors) slower than the wimp device whose address it duplicates. However, the hidden device ("*hidden dev*") may still intercept or respond to other types of USB data transfers faster. Thus the hidden device can be directed to compromise both I/O data secrecy

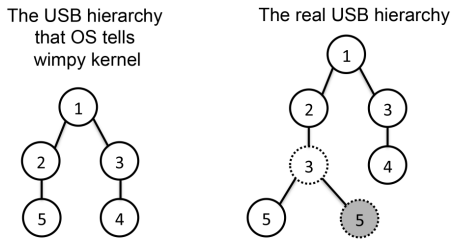


Figure 3. USB address overlap and remote wake-up attacks. *Legend:* The root of the USB bus denotes the USB host controller, the leaves the USB devices, and the intermediate nodes the USB hubs. The number of each tree node denotes the USB device address. The dotted nodes represent the USB devices whose addresses are duplicated in an attack. The grey node denotes the USB device that is suspended by the untrusted OS and can be remotely woken up using external signals (e.g., a special packet sent to a USB Ethernet card).

and integrity of a wimp device with the same address. We present a proof-of-concept experiment to showcase this attack in Section A.

Remote Wake-up Attack. A subtle attack can be launched by USB devices in suspended state which can still respond to external wake-up signals (e.g., a special packet sent to a USB Ethernet card) and resume their active state. Taking advantage of this *remote wake-up* feature, a compromised OS can configure a *hidden dev*, suspend it to evade verification, and later resume it to launch a "USB address overlap attack". However, we note that the remote waking up of a device needs to be coordinated by an upstream, non-suspended USB hub [4]. In a more potent attack, the OS could configure the hub upstream of the suspended device as a *hidden dev* (e.g., the dotted node No.3 in Figure 3), which would hide the remote wake-up event from the wimpy kernel. Thus, to defend against this subtle attack, the wimpy kernel verifies (1) that only the hubs that connect the wimp device to the host controllers are in non-suspended state during wimp execution, (2) that there is no hidden hub in the hierarchy, and (3) the status of all non-suspended hubs to detect any remote wake-up signals.

Hierarchy Verification Algorithm. The purpose of the verification algorithm is to check that only the *USB paths* of the wimp devices are in active state under a USB host controller. Here a USB path denotes a chain of USB devices from the the host controller, via the on-path hubs, and to a specific wimp device.

To design this algorithm, we need to overcome several challenges posed by the two attacks and the complexity of USB bus (illustrated in Section IV-A). For instance, the USB hierarchy information about USB address and hub-device connectivity is maintained only in the bus subsystem software of the untrusted OS. There is no hardware-stored hierarchy information that can be directly used by the WK. When discovering the hierarchy information, the WK must communicate with the USB devices using common operations instead of device-specific ones (to minimize code size and complexity). In addition, the WK must not interfere with the normal functions of the I/O hardware being verified; e.g., it must not make un-recoverable configuration changes.

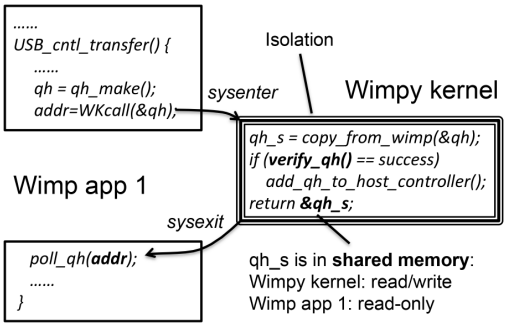


Figure 4. USB Transfer Descriptor Verification by the Wimpy Kernel.

In the on-demand isolation model, the untrusted OS prepares a set of USB paths for all wimp devices, and provides them as inputs to the WK verification algorithm. Specifically, the OS backs up the state of all non-USB-path devices, suspends them, and passes the USB path information to the WK. The USB path information includes the addresses of all devices and on-path hubs, and the ports of their upstream hubs that they connect to. The WK protects the host controller so that the untrusted OS can no longer issue any USB command via this host controller. The WK then executes the following algorithm to verify the OS-prepared USB paths:

(1) WK periodically monitors the port status of all on-path hubs to detect remote wake-up events. If any is detected, the verification fails.
(2) WK examines all hub ports that do not have any down-stream wimp device. These ports should either be disabled or suspended. Otherwise, the WK suspends those ports.
(3) WK scans all the device addresses (e.g., 127 addresses possible for USB 2.0). If it detects any that are active non-USB-path devices, the verification fails.
(4) For each device in USB path, WK suspends it, and then communicates using its address. If there is any reply, a hidden dev or hub is detected, and verification fails.

In Appendix B, we present an informal analysis of the algorithm and argue that it prevents both the USB address overlap and remote wake-up attacks.

2) Mediating the Exported USB Request Handling: In our system, most of the USB device operation module is deprivileged and pushed to the wimp apps. WK only verifies the behavior of the wimp apps that may affect wimp app isolation from the OS. For example, as shown in Figure 4, if a wimp app intends to perform certain operations to its device, it generates a set of transfer descriptors *qh*s. However, it cannot directly add descriptors to controller hardware, which is controlled by WK. Instead, the wimp app invokes the WK using a system call like interface (*WKcall*) with the descriptors *qh*s as input. The WK copies the descriptors to its kernel space, verifies them, and submits the valid descriptors to the host controller hardware. The copied descriptors are placed in a shared memory area to allow efficient descriptor status polling by the wimp app. This cannot compromise security, because the shared memory is read-only for the wimp app.

In this outsourcing model, the wimp apps bookkeep their

Table II
MINIMIZING DRIVER SUPPORT CODE IN WIMPY KERNEL.

Driver Support Code		Minimization Decisions
Memory Management	Virt & phys memory	Exported to wimp apps
	Page permissions	**Mediated by WK**
Synchronization	Locks	Exported to wimp apps
	Threads	Exported to wimp apps
	Signals	Exported to wimp apps
Kernel Library	Utility functions	Exported to wimp apps
	Timer	Exported to wimp apps
Device Library	Class functions	Exported to wimp apps
	I/O ports & mem	Exported to wimp apps
	Config space & Interrupts	**Mediated by WK**
Kernel Services	File system	Outsourced to OS
	CPU scheduling	**Mediated by WK**

USB transfer information, and fill a large amount of other descriptor fields. The WK only needs to verify a few security-critical descriptor fields to verify that wimp apps filled them correctly. The principle of verification is that those fields in the descriptors do not affect the isolation of the wimp apps' devices and other devices controlled by the wimpy kernel and the untrusted OS. The wimpy kernel does not verify descriptor fields that only affect the availability of the wimp apps' devices. In addition, the verification algorithm of the security-sensitive fields are general and simple, without complicated bus-specific semantics. For example, the wimpy kernel performs simple range checking on the Buffer Pointer fields in the descriptors, and makes sure that these fields point to the wimp apps' DMA memory region. Similar checking also applies to other bus subsystems. Section VI-B2 presents the details of USB transfer descriptor verification.

C. Exporting Driver Support Code

Aside from communicating with bus subsystems, device drivers also use a variety of services of commodity OS subsystems; e.g., kernel library, memory management, synchronization, device library and other kernel services [36]. Table II shows examples of such interfaces in each category and how we export them to minimize the code base of WK, according to the on-demand I/O isolation model.

(1) Memory management interfaces are further divided into three types: virtual memory pages, physical pages, page permissions. Virtual and physical page management is done in wimp apps, because during wimp registration, memory (including the code, data and I/O memory) of wimp apps is provisioned by the OS, and isolated by the micro-hypervisor and wimpy kernel. The wimpy kernel verifies that the OS provisions contiguous memory in both virtual and physical address spaces to the wimp apps, so that the wimp apps can easily perform page mapping translation. However, the WK sets page permissions for wimp apps to prevent buggy or compromised wimp code from subverting the WK's virtual memory isolation.

(2) Synchronization functions (e.g., locks, threads, and signals) are either unnecessary in the on-demand isolation model, or can be deprivileged to wimp apps. First, locks (e.g., mutex, semaphore, conditional variable) that are used for multiplexing devices among different applications are unnecessary, because wimp apps exclusively own their devices during execution. Locks for other usage can be easily implemented in user-level. Second, wimp apps implement their own thread management and scheduling functions using user space thread libraries [2] and timer interrupts delivered by WK. Third, if multi-process is needed[6], wimp apps manage the signals between their processes, using user-space signal implementation.

(3) Kernel library for utilities, timers, debugging and bookkeeping are unprivileged and can be replaced by user-level libraries in wimp apps. For example, wimp apps manage their own timers, because WK delivers timer interrupts to wimp apps.

(4) Device library include routines supporting a class of device and other low-level I/O related functions. Device-class functions are now placed in wimp apps, similar to device drivers. Low-level I/O resources such as I/O ports, MMIO and DMA memory are already isolated by the wimpy kernel, thus the wimp apps directly manage them without any runtime mediation by WK. However, configuration space access code (e.g., changing MMIO base address registers, modifying Message Signaled Interrupt Capability) and interrupt management functions (e.g., acknowledging End of Interrupts register, enable/disable interrupts) exported to wimp apps should be mediated by WK, because this code could be exploited by malicious or compromised wimp apps to breach I/O channel isolation.

(5) Kernel services include code for driver interaction with other OS subsystems, such as file systems and CPU scheduling. File system functions are outsourced to the OS by wimp apps, using the wimp-OS communication channels of WK (discussed below). Multi-process CPU scheduling, if needed, is implemented in wimp apps. However, the wimpy kernel needs to sanitize the new process page tables created by wimp apps during forking processes, and mediates page table switches.

D. Wimp-OS Communication

The wimp-OS channels enable bidirectional communication between the untrusted OS and the wimpy kernel or wimp apps. For example, a wimp app can request extra memory from the OS, when it runs out of the memory provisioned. The WK contacts the relevant OS services, and verifies that the dynamically assigned memory regions returned by the OS services are valid (e.g., they do not overlap with the memory regions of other wimp apps).

Conversely, the untrusted OS can use these wimp-OS channels to protect itself from potential buggy wimp behavior or defend against privilege escalation attacks from malicious wimps. When the OS invokes the wimp apps, it places upper bounds on the wimp apps' resources. If a wimp app exceeds these bounds, the OS requests the WK to take appropriate action. WK verifies these requests using the resource accounting information it keeps during wimp app execution. For example, if the OS detects a potentially deadlocked wimp app (e.g., which holds a CPU in excess of an established time bound), it notifies WK with the total running time as an input message.

[6]Multi-thread is usually sufficient for wimp apps that exclusively own their CPUs during execution.

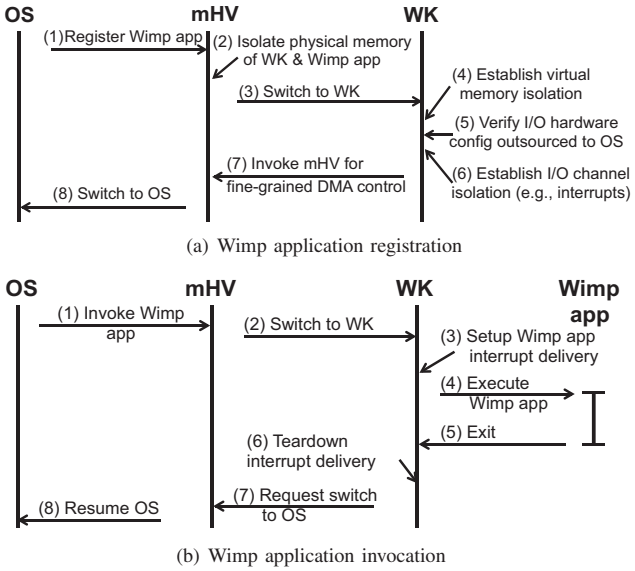
WK verifies this request by calculating the elapsed time of the wimp app, using the CPU time stamp it records during wimp app invocation and the current time stamp. If the total running time is correct, WK then notifies the wimp app to prepare for a descheduling. If the wimp app acts normally in descheduling, it can still be invoked by OS later. However, if the wimp app fails to deschedule for a certain amount of time, the untrusted OS can request the WK to terminate the wimp app. Similarly, an OS helper (e.g., a loadable kernel module) can constantly monitor shared interrupts of OS' devices. If it discovers that a shared interrupt with a wimp app is blocked for a long time, it could also complain to WK using wimp-OS channels.

We designed efficient asynchronous primitives for wimp-OS communication, which are compatible with standard commodity OS implementations. For example, when a wimp app requests OS services, it invokes WK-provided interfaces, instead of directly triggering high-weight context switches coordinated by the underlying micro-hypervisor. This yields substantially better performance for fine-granularity protection than that offered by security/separation kernels [10, 28, 55, 57, 64], recent micro-hypervisors [38, 60, 65], and traditional hypervisor designs [13, 14, 16, 31]. We demonstrated its efficiency in Section VII-B. Specifically, the wimp app provides an OS service number, inputs, and a completion call-back function to WK. The WK signals the OS running on other CPUs using Interprocessor Interrupts (IPIs) [7, 33], which is a standard facility of the Local Advanced Programmable Interrupt Controller (LAPIC) in main-stream multi-processor CPUs. It is frequently used to coordinate multi-processor bootstrap, but we use this capability to send an interrupt to other processors where the OS executes, as a signal of service requests. Before sending the IPIs, WK places the wimp app-provided inputs in a dedicated memory region shared with the OS, which is established by the micro-hypervisor during wimp app registration. After IPIs are sent, WK transfers control back to the requesting wimp app, and the wimp app continues to perform other operations. Later, the OS sends an IPI to WK to signal the service completion, and returns service results using the shared memory region. The WK verifies the service results and passes them to wimp app.

V. System Life-cycle

We illustrate the life cycle of isolated I/O channels and the interactions between the micro-hypervisor, the wimpy kernel and the wimp apps, as shown in Figure 5.

Registration. The untrusted OS or untrusted application provisions the memory (e.g., stack and heap) and wimp device I/O resources (e.g., MMIO memory, DMA memory) required by a wimp app, and explicitly registers the wimp app through an OS-hypervisor interface. During registration, the *mHV* isolates the wimp app's memory and I/O resources, maps the WK to the virtual address space of the wimp app, and transfers control to WK. The WK creates the virtual address page table of the wimp app and itself, verifies the configurations of the wimp devices and necessary hardware, and establishes the isolated I/O channels for the wimp devices (except for the interrupt delivery). Until unregistration, the untrusted OS can no longer tamper with the memory regions and I/O resources of the registered wimp apps.



(a) Wimp application registration

(b) Wimp application invocation

Figure 5. The Life-cycle of Wimp Applications.

Invocation. The OS implicitly invokes the wimp app by executing one of the wimp app's entry points. The *mHV* detects this execution and switches the context to the WK. The WK establishes the wimp-OS channels for the wimp app, sets up the wimp device interrupt delivery, and then begins executing the requested entry points at the wimp app's privilege level. Upon finishing execution, the wimp app suspends its devices and transfers control to WK. The WK disables the wimp-OS channels and wimp device interrupt delivery, and then the *mHV* takes control and performs a context switch to the OS. Between invocations, the OS can run other applications, but cannot use the wimp devices or tamper with the wimp app. Note that the wimp app could be invoked for arbitrary times after registration, and the invocation is efficient, because most I/O configuration overhead has already been offloaded to registration.

Unregistration. The OS explicitly requests wimp app unregistration via the wimp-OS channel, which is faster than via an OS-hypervisor interface. The WK resets the wimp devices to a clean state, tears down the isolated I/O resources of the wimp app with the help of the *mHV*, restores the configurations of the shared I/O hardware, and returns the CPU, memory regions and I/O resources of the wimp app to the OS.

VI. Implementation

A. Micro-hypervisor

The micro-hypervisor implementation is based on XMHF open source package v0.2.2 [1]. We extend XMHF with two main functions. First, we implement a fine-grained DMA protection function of the IOMMU, which allows the wimpy kernel to enable/disable DMA access of a device to a certain memory region, because the original XMHF only supports

coarse-grained DMA protection, which simply disables DMA access of any device to a specific memory region. Our fine-grained DMA protection is based on Xen-4.3.0 source code. Second, we implement the wimp app registration and unregistration interfaces, using the XMHF's memory isolation primitive and the DMA protection primitive we have implemented. The registration interface is the only interface provided to the untrusted OS, and the unregistration interface to the WK. The code base break-down of the micro-hypervisor is shown in Section VII-A.

B. Wimpy Kernel

Due to the simplicity of the reduced USB code in the wimpy kernel, we implement it from scratch based on the source code of Coreboot/Seabios [3] and the Enhanced Host Controller Interface (EHCI) host controller driver in Linux, adding the USB hierarchy verification and transfer descriptor (TD) verification algorithm. As for wimpy kernel interfaces, we implement the WKcall for wimp apps based on x86 fast system call instructions, and the wimp-OS communication channel based on IPIs and shared memory. Note that our prototype is implemented on x86 platforms, and we have not fully implemented the interrupt delivery and isolation mechanisms. The experimental results in Section VII show the minimality and efficiency of the WK.

1) USB Hierarchy Verification: The hierarchy verification algorithm only requires a few standard operations, including PCI configuration space operations to access EHCI host controller registers [5], and basic USB control and interrupt transfer operations to access registers of USB hubs, via the host controller [4]. The control and interrupt transfers are much easier to configure than the other two USB transfers (i.e., bulk and isochronous) and require smaller TCB.

In Step 1 of the algorithm, WK monitors remote wake-up events by setting periodic interrupt transfers to the port status endpoints of all on-path hubs. The endpoint data contains a bit to indicate that the hubs have coordinated a wake-up event. This type of event is always be detected by the periodic checking.

In Step 2, WK scans through all device addresses by sending standard SET_Configuration commands to each address. By specification, every USB device supports at least a default configuration No.1, thus an active device should always respond to a SET_Configuration=1 command. We choose this command, because its USB transaction does not have a data stage and introduces less latency overhead. A non-malicious USB device should always acknowledge this command within 50ms. If a scanned device address does not exist, the command will return an error immediately.

In Step 3, WK suspends an on-path hub or wimp device by sending a SET_Feature command to the upstream hub port that the hub/device connects to. If the upstream hub is the root-hub, WK directly accesses the port status registers of the host controller using PCI read command. After a device is suspended, WK finds out hidden devices by sending a SET_Configuration command to the same address device.

2) USB Transfer Descriptor Verification: There are four different types of descriptors specified in USB 2.0, namely Queue Head (QH), Isochronous Transfer Descriptor (iTD), Split Transaction Isochronous Transfer Descriptor (siTD) and Frame Span Traversal Node (FSTN) [4]. QH contains zero or more Queue Element Transfer Descriptors (qTD).

The WK exposes seven interfaces to wimp apps, in two categories: `attach_QH`, `attach_iTD`, `attach_siTD` and `attach_FSTN` for submitting descriptors; `reactivate_qTD`, `reactivate_iTD` and `reactivate_siTD` for reactivating the executed descriptors. FSTN descriptors need not be reactivated [5].

For the first four interfaces, WK verifies the following fields of the descriptors: the Device Address fields in QH, iTD, and siTD, to assure that the addresses refer to the correct wimp device; the Buffer Pointer fields in qTD, iTD, and siTD, to make sure that the addresses point to the wimp app's own DMA memory region; a few other fields that lead to undefined operations if configured incorrectly, such as the Maximum Packet Length field in QH and iTD, the Total Bytes to Transfer field in siTD, and the Typ field in FSTN.

3) Wimpy-Kernel Interfaces: We implement the WKcall interface using the standard x86 Fast System Call instruction [7, 33] (SYSENTER for requesting wimpy kernel services, and SYSEXIT for the wimpy kernel to switch to the wimp app, both after serving syscalls and when invoking the wimp app). Parameters (e.g., service ID, pointers to input/output data structures) are passed by registers. Alternatives like SYSCALL/SYSRET and "int 0x80" work, but SYSENTER/SYSEXIT is widely available on x86 platform and is more efficient.

For wimp-OS channels, WK triggers an IPI by programming the interrupt command register (ICR) of LAPIC to specify the IPI vector number and delivery destination. The delivery status bit of ICR indicates whether the IPI is sent. On the receiving CPUs, the IPIs are delivered as normal edge-triggered interrupts. The IPIs are used as notifiers of wimp-OS communication. The real data, including wimp-OS service ID and input/output parameters, is passed by shared memory buffer, which is established during wimp app registration, by *mHV*.

C. Device Driver Study

We perform device driver study on Linux Ubuntu 12.04 with kernel 3.2.0. We develop automatic scripts to extract the external interfaces that character device drivers use, by analyzing the symbol tables in drivers' binary headers and looking for undefined symbols. We filter out the undefined symbols that point to the functions implemented in other drivers, and leave only the symbols of Linux kernel services. To verify correctness and completeness, we compare the results of these scripts with those of the OCaml/CIL source code analysis tools used in [36], and the CodeSurfer software analyzer [9][7].

[7]We are able to compile driver sub-directories using the academic version of CodeSurfer, though building the entire Linux kernel fails.

Table III
MINIMIZING DRIVER SUPPORT CODE IN THE WIMPY KERNEL. "IN LINUX" COLUMNS SHOW THE UNIQUE OS INTERFACES USED IN ALL DRIVERS AND CHARACTER-ORIENTED DEVICE DRIVERS, RESPECTIVELY.

Driver Dependency	In Linux		In Wimpy Kernel
	All	Char Dev	
Memory Management	113	67	set_page_permission
Synchronization	189	95	None
Kernel Library	863	349	None
Device Library	612	212	en/disable_irq, config_write
Kernel Services	442	254	None

We manually study those driver dependencies, and present the result of deprivileging the relevant OS support code in Table III, following the interface categories defined in [36]. According to the analysis presented in Section IV-C, we identify only a few interfaces that should be implemented in the WK, and the support code of other interfaces can be all deprivileged to wimp apps. The wimpy kernel should verify the `set_page_permission` requests to prevent the wimp app from subverting the wimpy kernel's virtual memory isolation. The wimp apps use `enable_irq` and `disable_irq` WKcalls to enable/disable the interrupts of their devices, and use `config_write` to modify configuration space registers, under the WK's mediation.

Drivers also invoke privileged instructions, such as `wrmsr/rdmsr`, `wbinvd`, `lgdt/lidt`, and `ltr`, which are available at the user-level. However, by scanning through all character device drivers in Linux, we found only one instruction (`wbinvd` for invalidating cache entries) used in one video driver during device initialization. When migrating this driver, wimp apps can outsource the initialization functions to the untrusted OS. Otherwise, the WK simply provides a WKcall interface for this instruction.

VII. EVALUATION

We implement and evaluate the system on an off-the-shelf HP Elitebook 8540p with a Dual-Core Intel Core i5 M540 CPU running at 2.53 GHz, 4GB memory; a Hitachi GST Travelstar 7200 rpm 500GB SATA-II disk; an Intel 82577LM Gigabit network card; and an Infineon v1.2 TPM. The machine is also equipped with two USB 2.0 host controllers and two immediate downstream rate matching hubs for transforming high-speed USB transactions to low-speed ones. The machine runs a 32-bit Ubuntu 12.04 OS with Linux kernel 3.2.0-36.56. The wimp application tested in our experiments is a prototype that includes a USB keyboard device driver. In all network experiments, the machines are connected via 1Gbps Ethernet links.

A. Code Base Size Evaluation

We use SLOCCount[8] to calculate the Source Lines of Code (SLoC) of the *mHV* and the WK. As shown in Table 3(a), the micro-hypervisor has 25211 SLoC, adding 660 SLoC to the XMHF [1] code base for wimp app registration and unregistration, and 4925 SLoC to complete the XMHF's DMA protection primitive. The code addition does not invalidate the XMHF's formally-proved memory integrity property [66]. The code base of our micro-hypervisor is smaller than other micro-hypervisors, and much smaller than full functioning VMMs/hypervisors[9].

[8]SLOCCount by David A. Wheeler, http://www.dwheeler.com/sloccount/

Table IV
SYSTEM CODE BASE SIZE. (*) IN MICRO-HYPERVISOR IMPLEMENTATION, WE AUGMENT THE ORIGINAL XMHF WITH FINE-GRAINED DMA PROTECTION CAPABILITY.

(a) Micro-hypervisor

Modules	SLoC
Registration	447
Unregistration	213
XMHF*	24551
Total	25211

(b) Wimpy Kernel

Modules	SLoC
USB Subsystem	2144
WKcall	249
Wimp-OS Channel	106
Others	1038
Total	3537

Table V
COMPARISON OF CODE SIZE IN USB SOFTWARE STACK BETWEEN WK AND IN LINUX. (*) WE CALCULATE ONLY THE USB DRIVERS INCLUDED IN THE LINUX KERNEL TREE.

Wimpy Kernel				Linux		
Verification		Others	Total	USB Subsystem	USB Drivers	Total
Hierarchy	TD					
93	107	1944	2144	19820	>206376*	>226196*

Table 3(b) shows the code base break-down of the current WK prototype. The WK code size is about 3.6K SLoC, 60% of which is USB bus subsystem relevant code. This code base is sufficient to support all types of USB 2.0, 1.1, and 1.0 devices, and all types of USB transfer mode, such as control, interrupt, bulk and isochronous transfers [4].

Table V compares the WK USB software stack to the commodity Linux one (Both only support USB EHCI host controller). We manage to introduce only 2144 SLoC of USB code to the wimpy kernel, which represents more than 99% reduction compared with the over 22K SLoC of Linux USB code base. Note that the reduction result in practice is even better, because when calculating the Linux USB code, we do not include a significant number of third party USB drivers out of the Linux kernel tree and drivers relevant to high-level protocols (e.g. SCSI drivers for USB flash drive). In addition, the USB hierarchy verification algorithm and transfer descriptor verification algorithm only use 93 and 107 SLoC, respectively.

B. Micro-benchmarks

USB Hierarchy Verification. Table VI shows the latency of each step in the USB hierarchy verification algorithm. Among them, device address scanning (step 3) dominates the latency overhead. However, this overhead is acceptable, because this algorithm is only invoked once per wimp application registration, and does not affect the more frequent wimp app invocations.

[9]Fides [62] has 7.2K SLoC, but without DMA protection, multi-core and AMD x86 virtualization support. The new version of TrustVisor based on XMHF [66] has about 24K SLoC without implementing fine-grained DMA protection. Guardian [15] has approximately 25K SLoC. NOVA's code base contains 36K SLoC [61]. BitVisor [60] has 194K SLoC. Most full-function VMM/hypervisors have code-base sizes which are nearly an order of magnitude larger than our micro-hypervisor; e.g., Xen (263K SLoC), VMWare ESXi (200K SLoC), KVM (200K SLoC), and Hyper-V (100K SLoC).

Table VI
LATENCY BREAK-DOWN OF THE USB HIERARCHY VERIFICATION ALGORITHM.

	Step 1	Step 2	Step 3	Step 4	Total
Time (ms)	0.29	0.54	573.03	1.32	575.18

Table VII
LATENCY COMPARISON OF WK-INVOLVED AND HYPERVISOR-INVOLVED CONTEXT SWITCHES.

	WKcall	Wimp-OS Channel	Hypercall	Page Fault
Time (μs)	0.38	0.23	7.56	20.68

Table VIII
LATENCY OF WIMP-APP LIFE-CYCLE OPERATIONS.

	Registration	Invocation	Unregistration
Time (ms)	583.79	0.26	0.97

USB Transfer Descriptor Verification. In our experiments, the latency overhead of TD verification is negligible. For example, verifying a QH and an iTD only takes about 0.28 μs and 0.42μs, respectively. In comparison, a micro-frame, the minimum time unit in USB specification, takes 125μs.

Wimpy-Kernel Interfaces. Table VII illustrates the latency overhead of two main wimpy kernel interfaces; i.e., the WK-calls for communicating with wimp applications, and the IPI-based wimp-OS channels for communicating with the OS. These two interfaces avoid the more heavy-weight micro-hypervisor-involved context switches and greatly improve overall system performance. Hypercalls and hardware page faults are the two most widely used methods of triggering hypervisor-involved context switches. In comparison, our WK-calls are about 20 times faster than hypercalls and 54 times faster than page faults. Our wimp-OS IPI channels are 33 times faster than hypercalls and 90 times faster than page faults. In addition, using the asynchronous wimp-OS channels, the wimp apps and wimpy kernel do not block waiting for the OS services.

System Life-cycle Operations. Table VIII presents the latency overhead of the registration, invocation and unregistration of a wimp application. The latency of wimp application invocation and unregistration are much smaller than those of registration, because the more heavy-weight hardware configuration verification is only invoked during registration.

C. Macro-benchmarks

In this section, we attempt to evaluate the overhead of the micro-hypervisor and the wimpy kernel to the co-existing OS, both in CPU and I/O performance. We use the standard SPECint 2006 as our CPU-bound benchmarks. For I/O workloads, we choose the iozone 3.397 for disk read/write, netperf 2.5.0 (TCP and UDP) and Apache Benchmark (*ab*) for networking. Specifically, in the iozone test suite, we choose evaluation parameters to be (block size: 4KB, file size: 8GB). For netperf, we set the message size to be 16384 bytes and select a 120 seconds duration. For Apache, we run the Apache HTTP Server 2.2.22 on our testbed and run *ab* on another machine to generate 200, 000 transactions using 20 concurrent connections.

We evaluate the performance overhead in two steps. First, we compare the performance overhead of TrustVisor and *mHV* (named "TrustVisor" and "*mHV*" in Figure 6), as both of them are based on XMHF and provide basic isolated execution environments. In the TrustVisor test cases, we run TrustVisor along with the OS, without registering or invoking any isolated software modules. In the *mHV* test cases, we run the *mHV* without registering any wimp application (the WK is not mapped to any wimp app's address space). Second, we further measure the performance overhead introduced by the WK on the same benchmarks (named "*mHV* w/ WK" in Figure 6), by registering the wimp app but not invoking it. The results shown in Figure 6 are all normalized to the benchmark results on the vanilla OS.

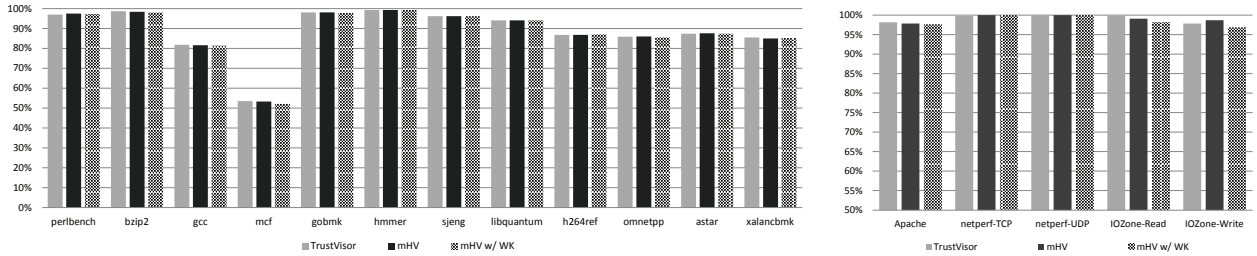
CPU Benchmarks. As shown in Figure 6(a), the *mHV* incurs similar performance overhead as TrustVisor, because they have similar memory foot-print, and rely on the same hardware virtualization support for memory isolation. The performance overhead introduced by the WK memory foot-print is negligible, comparing to that of the TrustVisor and the *mHV*.

I/O Benchmarks. The I/O evaluation results are shown in Figure 6(b). We measure the network transfer rate (KB/s) of Apache web server and netperf benchmark, and the disk read/write throughput (KB/s) of the iozone benchmark. All disk and network I/O test results in our experiment show less than 4% performance downgrade, comparing with the vanilla OS case. The performance of the *mHV* is similar to that of the TrustVisor, and the *mHV* w/ WK cases always have slightly worse performance. This is because the first two cases use coarse-grained DMA protection, which is more light-weight than the fine-grained DMA protection used in the wimpy kernel. We expect that the I/O performance overhead will decrease along with more advanced hardware for DMA protection.

VIII. RELATED WORK

A. I/O Isolation Systems

Limited Device Support. Security kernels [10, 57], isolation kernels [53], and micro-hypervisors [15, 70] support isolated channels for a few selected user-interface devices (e.g., security administrators) within their TCBs. This approach inevitably increases the size and complexity of trusted code and does not apply to the wide variety of devices that need to be supported outside the TCB. Zhou *et al.* [69] illustrate a limited form of user-verifiable trusted paths to application-code modules, protected by a micro-hypervisor [47]. Filyanov *et al.* [21] also proposes using an isolated software module to control user-centric devices (e.g., keyboard and display), but does not protect the I/O data from an OS's intentional mis-configuration. The DriverGuard system [17] only protects the confidentiality of the I/O data between commodity devices and small code modules in device drivers. In contrast with these systems, we address the seemingly conflicting and more challenging requirements of supporting diverse and complex I/O devices while, at the same time, maintaining overall system



(a) CPU Benchmarks (SPECint 2006)

(b) I/O Benchmarks

Figure 6. The CPU and I/O macro-benchmark results.

simplicity. Our system protects I/O data against subtle device mis-configuration attacks (e.g., USB address overlap attacks and remote wake-up attacks), which these systems do not (claim to) counter.

Static Device Allocation. Separation kernels [28, 55, 64] can isolate I/O channels by allocating devices to different system partitions, which are statically defined at system configuration time. They also enforce strict information flow policies among these partitions – a goal that we do not share. NoHype [38, 65] dedicates I/O devices with virtualization support (e.g., SR-IOV [34]) to virtual machines (VMs) through a static pre-allocation process. Their system design is based on the observation that a VM running on a cloud platform only needs a limited number of I/O devices; e.g., network interface cards, storage, and graphic cards. Thus, they cannot protect I/O data from user devices such as a mouse, VGA, or printer. In contrast, our system focuses on providing dynamic, on-demand isolation of a wide variety of peripheral devices.

Device Virtualization and Passthrough. During the past decade, advances in device virtualization have decreased the trusted code base for isolated I/O channels, gradually evolving from the monolithic hypervisors/VMMs to hypervisors with privileged device management domains [11], then to hypervisors with disengaged privileged domains [18], and finally to hypervisors with isolated driver domains [22, 51]. However, applications in their guest domains still communicate with virtualized devices via the untrusted guest OS on which they run, which still implies that a huge code base has to be trusted for on-demand, isolated I/O. Hypervisors with device pass-through support (e.g., Xen, KVM, and [46]) or para-passthrough support (e.g., BitVisor [60]) exclusively assign I/O devices to a specific guest VM. A driver of the pass-through device still has to co-exist with the untrusted guest OS. Worse, a compromised control domain can break the isolation of the pass-through devices. In contrast, our system is specially designed to avoid virtualizing hardware devices of commodity OSes. We control only the necessary hardware for I/O channel isolation, and rely on a small and simple trusted code base.

Special Devices. Some systems take advantage of special hardware devices – equipped with data encryption capability – to establish secure I/O channels with isolated software [30, 40, 49, 67]. Our system avoids the attendant secure key management issues and special devices, and supports protected I/O channels to commodity peripheral devices.

B. Isolated Execution Environments

Recent advances on isolated execution environments (IEEs), using both software [47, 48, 56, 61, 62, 66] and hardware architectures [50], illustrate the safe co-existence of software code modules or applications with an untrusted OS. However, most IEEs lack basic services for application development. Other systems add a few such services to the IEE with minimal TCB support; e.g., persistent memory [52], file system and network services [13, 14, 16, 31, 44], inter-IEE communication [62], and limited user trusted path [69]. They do not include services for on-demand isolated I/O channels to diverse and complex peripheral devices (e.g., USB devices). In contrast, this paper addresses this unmet challenge on commodity platforms. In addition, most of these systems [13, 14, 16, 31, 52] adopt a synchronous, or blocking, service communication model (i.e., the application execution ceases during services) and entail additional overhead for application-OS context switches with low-level hypervisor support. In contrast, we support asynchronous and more efficient wimp-OS communication channels without any low-level micro-hypervisor support and hence avoid extra overhead, on multi-core platforms. More recently, the Drawbridge/Library OS system [54] packaged rich, high-level application services (e.g., rendering engines, language run-time) with IEE software, but left the device drivers and kernel driver subsystem to the host OS. Thus, the trusted code base of their application I/O services is much larger than ours.

C. Device Driver Isolation and Decomposition

Several approaches [12, 23, 42, 43, 51, 63, 68] exist to isolate device drivers from the OS kernel, and/or move them to user-space, primarily for the purpose of improving driver reliability and fault isolation. Swift *et al.* propose using hardware memory protection domains to isolate the drivers of a monolithic kernel [63]. LeVasseur *et al.* [43] and Nikolaev *et al.* [51] propose running unmodified device drivers of guest operating systems in separate virtual machines. SUD [12] moves device drivers to an emulated Linux kernel environment in user-space. Leslie *et al.* [42] implement user-level device drivers on a Linux kernel. I/O channel isolation of all these systems relies on very large and untrusted OS code bases. In contrast, the overriding goal of our system is to decouple the I/O channels from an untrusted OS to obtain much higher isolation assurance. Ganapathy *et al.* [23] propose a microdriver

architecture that splits driver code, leaving the critical path code in the kernel and moving the rest (e.g., initialization, configuration) to a user-level process. They aim to achieve high driver performance and compatibility with commodity OSes. We share similar driver decomposition goals, but we focus primarily on reducing a system's trusted code base by outsourcing I/O management functions to the untrusted OS and verifying their behaviors in the wimpy kernel.

Williams *et al.* [68] develop an architecture that isolates device drivers in user space and a reference validation mechanism (RVM) that mediates their low-level interactions (e.g., MMIO, DMA, interrupts) with I/O devices. RVM relies on safety specifications for individual devices to identify allowed and prohibited interactions. This architecture has different goals than ours as it is based on an extensively re-designed OS. Also, enforcing safety specifications for individual devices is insufficient in the on-demand I/O model, since this model requires the composition of safety specifications for multiple interconnected devices via complex bus subsystems that are shared on a time-multiplexed basis.

Similarly, micro-kernels [39, 59] restructure commodity OSes by leaving essential functions like task scheduling and IPCs in the kernel, and moving the rest of OS functions to user-space; e.g., device drivers and bus subsystems. Though providing high assurance, these systems require extensive OS re-design, which is precisely what we avoid. Instead, we achieve safe co-existence of our trusted code base with unmodified commodity OSes. For our purposes, it would be equally undesirable to use a micro-kernel [39] and its user-level driver subsystems as our wimpy kernel, because we seek to retain the I/O programming model of commodity OSes, and encourage wimp applications to reuse commodity device drivers to the largest possible extent.

IX. Conclusion

Trustworthy applications are unlikely to survive in the marketplace without the ability to use a variety of basic services securely, such as on-demand isolated I/O channels to peripheral devices. This paper presents a security architecture based on a wimpy kernel that provides these services without bloating the underlying trusted computing base. It also presents a concrete implementation of the wimpy kernel for a major I/O subsystem, namely USB subsystem, and a variety of device drivers. Experimental measurements show that the desired minimality and efficiency goals for the trusted base are achieved.

Acknowledgment

We are grateful to the reviewers, and Weidong Cui in particular, for their insightful suggestions. We also want to thank Asim Kadav and Yueqiang Cheng for help with the driver study, and Amit Vasudevan for micro-hypervisor assistance and debugging. This research was supported in part by CMU CyLab under the National Science Foundation grant CCF-0424422 to the Berkeley TRUST STC and a gift from Intel Corporation. The views and conclusions contained in this paper are solely those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

References

[1] eXtensible Modular Hypervisor Framework. http://xmhf.org [Accessed on 21 Feb 2014].

[2] GNU Pth - The GNU Portable Threads. http://www.gnu.org/software/pth/ [Accessed on 21 Feb 2014].

[3] SeaBIOS. http://www.coreboot.org/SeaBIOS [Accessed on 9 Nov 2013].

[4] Universal Serial Bus Specification, Revision 2.0, 2000.

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

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

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

[8] J. P. Anderson. Computer security technology planning study. volume 2. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972.

[9] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Workshop on Inspection in Software Engineering*, 2001.

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

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

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

[13] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, 2007.

[14] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. Architectural Support for Programming Languages and Operating Systems*, 2008.

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

[16] Y. Cheng, X. Ding, and R. Deng. Appshield: Protecting applications against untrusted operating system. Technical Report SMU-SIS-13-101, Singapore Management University, 2013.

[17] Y. Cheng, X. Ding, and R. H. Deng. DriverGuard: Virtualization-based fine-grained protection on i/o flows. *ACM Transaction on Information and System Security*, 16(2):1–30, 2013.

[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 *Proc. ACM Symposium on Operating Systems Principles*, 2011.

[19] D. Denning. Cryptographic checksums for multilevel database security. In *Proc. IEEE Symposium on Security and Privacy*, 1984.

[20] D. R. Engler, M. F. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. 29(5):251–266, 1995.

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

[22] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual

machine monitor. In *Proc. Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[23] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[24] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. of CRYPTO*, 2010.

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

[26] V. D. Gligor and B. G. Lindsay. Object migration and authentication. *IEEE Transactions on Software Engineering*, SE-5(6):607–611, 1979.

[27] R. Graubart. The integrity lock approach to secure database mangement. In *Proc. IEEE Symposium on Security and Privacy*, 1984.

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

[29] T. C. Group. TPM specification version 1.2, 2009.

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

[31] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *Proc. international conference on Architectural support for programming languages and operating systems*, 2013.

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

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

[34] Intel LAN Access Division. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://download.intel.com/design/network/applnots/321211.pdf, 2011.

[35] P. A. Janson. Removing the dynamic linker from the security kernel of a computing utility. Technical Report MIT-LCS-TR-132, 1974, 1974.

[36] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[37] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, SE-17(11):1147–1165, 1991.

[38] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proc. annual International Symposium on Computer Architecture*, 2010.

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. ACM Symposium on Operating Systems Principles*, 2009.

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

[41] B. Lampson. Software components: Only the giants survive. *Computer Systems: Theory, Technology, and Applications*, (9):137–145, 2004.

[42] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[43] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. Symposium on Operating Systems Design and Implementation*, 2004.

[44] Y. Li, A. Perrig, J. McCune, J. Newsome, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. Technical Report CMU-CyLab-14-001, Carnegie Mellon University, 2014.

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

[46] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *Proc. USENIX Annual Technical Conference*, 2006.

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

[48] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. European Conference in Computer Systems*, 2008.

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

[50] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[51] R. Nikolaev and G. Back. Virtuos: an operating system with kernel virtualization. In *Proc. ACM Symposium on Operating Systems Principles*, 2013.

[52] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *Proc. IEEE Symposium on Security and Privacy*, 2011.

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

[54] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

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

[56] R. Sahita, U. Warrier, and P. Dewan. Protecting critical applications on mobile platforms, 2009.

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

[58] M. D. Schroeder, D. D. Clark, and J. H. Saltzer. The Multics kernel design project. In *Proc. ACM Symposium on Operating Systems Principles*, 1977.

[59] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *Proc. ACM symposium on Operating systems principles*, 1999.

[60] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing I/O device security. In *Proc. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2009.

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

[62] R. Strackx and F. Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *Proc. ACM conference on Computer and Communications Security*, 2012.

[63] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM Symposium on Operating Systems Principles*, 2003.

[64] W. R. Systems. Wind river vxworks mils platform.

http://www.windriver.com/products/platforms/vxworks-mils/MILS-3_PO.pdf [Accessed on 7 Nov 2013], 2013.

[65] J. Szefer, E. Keller, R. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proc. ACM Conference on Computer and Communications Security*, 2011.

[66] 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 *Proc. IEEE Symposium on Security and Privacy*, 2013.

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

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

[69] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

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

APPENDIX

A. USB Address Overlap Attack Experiments

We experiment with the USB address overlap attack, and analyze its impact on I/O channel isolation. Note that USB device communication has two directions: IN means data is transferred from device to host controller, while OUT represents the opposite. There are four types of data transfer: control, interrupt, bulk, and isochronous. Each type has different latency and bandwidth guarantees, and is performed by different types of USB devices.

We perform the analysis using two keyboards, one is Dell SK8115, as the wimp device, the other one is Dell L100, as a device controlled by the adversary. We changed the USB address of Dell L100 to overlap that of Dell SK8115. In the experiment, when performing control transfer IN direction communication (e.g., reading device descriptors), Dell SK8115 always replies faster, so we only read its device descriptors from the host controller. Dell L100 is hidden from the control software (e.g., verification software, wimp applications). However, when performing control transfer OUT direction communication (e.g., sending command to light the caps-lock LED on the keyboard), we discovered that the caps-lock LEDs on both keyboards are always lighted together. This means the hidden Dell L100 can silently intercept control OUT data of the isolated-channel device, which breaks the secrecy of the I/O channel. Moreover, if we perform interrupt control IN communication (e.g., reading keyboard input), key-presses on both keyboards are accepted normally, which means that the hidden Dell L100 can inject data into the isolated channel and break its integrity.

In summary, the USB device address overlap attack can break both the secrecy and integrity of isolated I/O channels, without being noticed by any control software.

B. Analysis of the USB Hierarchy Verification Algorithm

We first analyze that Steps 1 to 3 are able to find out all non-USB-path devices that are still in active state. The untrusted OS may attempt to hide a device when the WK scans it in Step 3, and remotely wake it up later. However, the remote wake-up event of a device must be coordinated by a non-suspended hub. This hub is either be a non-USB-path hub, or a hub on a USB path. For the former the WK will always discover it in the linear scan, and for the latter the remote wake-up event will be detected by the WK, as shown in Step 1.

Although Steps 1 to 3 guarantee that all non-suspended devices have correct addresses are on the USB paths, this does not prove that the given USB paths are correct, because hidden devs (or hubs) may still be on USB paths. Step 4 can rule out any hidden dev that is on a different USB-path with the targeted device whose address the hidden dev duplicates, but it cannot detect the hidden dev that is on the same USB-path with the targeted device ("same-path hidden dev").

We now provide a informal correctness argument on a proposition that the untrusted OS cannot configure any "same-path hidden dev" that manages to evade the WK verification and compromise the wimp I/O data isolation later. To be "meaningful", the same-path hidden device must either be able to intercept/fake messages between the host controller and the targeted device, or it must have suspended devices that are hidden downstream and can be remotely woken up later.

Before continuing with the argument, we need to make four observations on USB 2.0 specification. First, a non-malicious device/hub in its Configured state will *not* respond to SET_Address commands, unless it is deconfigured by a SET_Configuration command and transits back to Address state. Second, if a hub is in the Deconfigured state, all its downstream devices lose power and transit back to the Attached state, which is similar to resetting all downstream devices. Third, the remote wake-up capability is disabled by default, and can only be enabled when the device/hub is in its Configured state. Forth, a hidden device downstream to its target device cannot affect the message secrecy and integrity of the target device, because the target device always receives and responds to USB transactions faster than the downstream hidden device.

Our informal correctness argument is as follows: If the untrusted OS intends to configure a hidden device to duplicate the address of its upstream device, the SET_Configuration command to the hidden device is always intercepted by the upstream device, thus the hidden device can never transit to the Configured state, and thus "meaningless". If the untrusted OS sets a hidden device to duplicate the address of its downstream device, the hidden device must first be deconfigured, and thus all downstream devices will lose power and all their configurations. The hidden device itself becomes "meaningless".

In conclusion, we informally argue that the hierarchy verification algorithm can prevent both the USB address overlap and remote wake-up attacks.