Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE **On-Demand Isolated I/O for Security-Sensitive Applications on Commodity Platforms**

PRESENTED BY **Zongwei Zhou**

ACCEPTED BY THE DEPARTMENT OF

Electrical and Computer Engineering

__Virgil Gligor_____ _5/14/14_____
ADVISOR, MAJOR PROFESSOR DATE

___Jelena Kovacevic_____ ___5/14/14_____
DEPARTMENT HEAD DATE

APPROVED BY THE COLLEGE COUNCIL

__Vijayakumar Bhagavatula_____ _5/14/14_____
DEAN DATE

On-demand Isolated I/O for Security-sensitive Applications on Commodity Platforms

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Zongwei Zhou
B.E. Automation, Tsinghua University, China
M.E. Computer Science, Tsinghua University, China
M.S. Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2014

Copyright © 2014 Zongwei Zhou

Keywords: On-demand Isolated I/O, Trusted Path, Micro-hypervisor, Isolated Execution Environment, Peripheral Devices, Wimpy Kernel, Composibility, Trusted Computing Base, Outsource and Verify, Export and Mediate, Corporate Key Management

Dedicated to my beloved parents and wife

Abstract

Today large software systems (i.e., *giants*) thrive in commodity markets, but are untrustworthy due to their numerous and inevitable software bugs that can be exploited by the adversary. Thus, the best hope of security is that some small, simple, and trustworthy software components (i.e., *wimps*) can be protected from attacks launched by adversary-controlled giants. However, wimps *in isolation* typically give up a variety of basic services (e.g., file system, networking, device I/O), trading usefulness and viability with security.

Among these basic services, *isolated I/O* channels remained an unmet challenge over the past three decades. Isolated I/O is a critical security primitive for a myriad of applications (e.g., secure user interface, remote device control). With this primitive, isolated wimps can transfer I/O data to commodity peripheral devices and the data secrecy and authenticity are protected from the co-existing giants.

This thesis addresses this challenge by proposing a new security architecture to provide services to isolated wimps. Instead of restructuring the giants or bloating the Trusted Computing Base that underpins wimp-giant isolation (dubbed underlying TCB), this thesis presents a unique *on-demand I/O isolation model* and a trusted add-on component called *wimpy kernel* to instantiate this model. In our model, the wimpy kernel dynamically takes control of the devices managed by a commodity OS, connects them to the isolated wimps, and relinquishes controls to the OS when done. This model creates ample opportunities for the wimpy kernel to *outsource* I/O subsystem functions to the untrusted OS and *verify* their results. The wimpy kernel further *exports* device drivers and I/O subsystem code to wimps and *mediates* the operations of the exported code. These two methodologies help to significantly reduce the size and complexity of the wimpy kernel for high security assurance. Using the popular and complex USB subsystem as a case study, this thesis illustrates the dramatic reduction of the wimpy kernel; i.e., over 99% of the Linux USB code base is removed. In addition, the wimpy kernel also composes with the underlying TCB, by retaining its code size, complexity and security properties.

Acknowledgments

First, I would like to thank my advisor, Prof. Virgil Gligor, for his constant support and guidance throughout my Ph.D. endeavor. His over-forty-year experiences in security research help shape my research tastes and inspire me to think big and aim high. His dedication and enthusiasm urges me to push myself to the limit and always pursue perfection in every bit of details, regardless of the effort to take. I am particularly grateful for every unexpectedly long meeting and extensive discussion we have, and every late night he spend with me to improve papers word-by-word.

I cannot achieve my research results without the great ideas, invaluable feedback, and indispensable collaboration from other members of my thesis committee, Prof. Gregory Ganger, Prof. Adrian Perrig, and Dr. Jesse Walker, and all my co-authors as well: Yueqiang Cheng, Anupam Datta, Robert Deng, Xuhua Ding, Jun Han, Geoff Hasker, Hsu-chun Hsiao, Tiffany Hyun-jin Kim, Yanlin Li, Yue-Hsun Lin, Jonathan McCune, James Newsome, Emmanuel Owusu, Bryan Parno, Ning Qu, Patrick Tague, Amit Vasudevan, Miao Yu, and Xin Zhang. During my internship at Microsoft Research Redmond, my mentors, Himanshu Raj, Stefan Saroiu, and Alec Wolman expanded my horizons and guided me to conduct interesting research on mobile platforms. In addition, I am also indebted to Prof. Jun Li and Yibo Xue, my master advisors at Tsinghua University of China, who opened up my eyes to both academia and industry in the U.S. Without them, I would never end up in Carnegie Mellon.

I also benefited from the interactions with colleges at Carnegie Mellon and other members of security research communities: Davide Balzarotti, Kevin Butler, Luis Brandao, Chen Chen, Weidong Cui, Colin Dixon, Michael Farb, Yoshiharu Imamoto, Limin Jia, Asim Kadav, Min Suk Kang, Gihyuk Ko, Soo Bum Lee, Junda Liu, Aziz Mohaisen, Ian Pratt, Yaxuan Qi, Yong Qiao, Zhiyun Qian, Edward Schwartz, Vyas Sekar, Elaine Shi, Saurabh Shintre, Michael Stroucken, Rui Wang, Tao Wei, Dan Wendlandt, Hui Xue, Osman Yagan, and Jun Zhao.

My life in Pittsburgh (and Seattle at summer 2011) would not be so colorful without a group of brilliant friends: Bin Fan and Shuang Su, Yupeng Fu and Yun

Huang, Tianyu Gu, Boxing He, Mingyang Hu, Yang Hu, Zhaoyin Jia and Jing Xia, Hongwen Kang, Puyan Li, Yanlin Li and Lingjuang Peng, Bo Lin, Li Lv, Long Qin and Yanan Chen, Kai Ren, Yuan Tian, Yuandong Tian, Han Wang, Xiao Wang, Junqing Wei and Shimeng Huang, Xusheng Xiao, Guang Xiang, Liang Xiong, Zhou Ye, Miao Yu and Huilin Hu, Xin Zhang and Jiayao Han, Yin Zhang, Jun Zhao, and Yiqi Zhu.

The devoted love from my parents is always with me, no matter how far I am away from them. Their examples quietly shape my personality and character. I am sure that I could accomplish more if I have exactly followed my Dad's 100% dedication to each of his professions. My mom, who barely made her elementary school, is my earliest and best teacher at all time. Her kind-hearted and thoughtful character inspires me to be a better person overall.

Finally, my wholehearted gratitude goes to my lovely wife, Heshuang. Thanks for putting up with the six-year long-distance life and going through every joy and pain with me. She spreads my wings and makes me soar. I am incredibly lucky to find such an angel that brings the best out of me.

The research in this thesis was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office, and by the National Science Foundation under grants CNS083142, CNS105224, and CCF0424422, and by a gift from Intel Corporation. The views and conclusions contained here are those of the author and should not be interpreted as representing the official policies, either express or implied, of any sponsoring institution, the U.S. government or any other entity.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Wimps and Giants

Modern architectures can isolate security-sensitive application code from the untrusted code of commodity platforms, thereby enabling their safe coexistence [21, 22, 24, 54, 80, 83, 84, 86, 100, 112–114, 127]. 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 [74]. 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*.

Unfortunately, isolating security-sensitive wimps from untrusted giants does not guarantee the wimps' survival on commodity platforms. To avoid re-creating giants inside their isolated execution environments, wimps often give up a variety of basic services, which greatly undermines their usefulness and viability. For example, wimps typically lack persistent memory [91], file system and network services [21, 22, 24, 54, 80], flexible trusted paths to users [23, 30, 37, 137], and isolated I/O services [15, 25, 27, 40, 49, 69, 82, 87, 98, 112, 118, 119] needed for many applications in fields like 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 [67, 105], but failed to deliver trustworthy OSes that met the product compatibility and timeliness demands of competitive markets [45, 81]. For example, the VAX VMM security kernel [67], after eight years of considerable design effort, was never deployed due to the absence of Ethernet support, whose popularity was not anticipated initially. Moreover, including basic services in the trusted computing bases (TCBs) [1] that guarantee safe giant-wimp coexistence has been equally unattractive. With this approach, TCBs would become bloated, unstable, and unverifiable, and thereby lose their assurance; i.e., they would use large and complex code bases of diverse, uncertain origin (e.g., device drivers) needed for different applications and would require frequent updates because of function additions, upgrades, and patches.

In contrast, this thesis presents a unique solution to this problem by placing the basic services in the giants. To survive, wimps must 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.

1.2 Why Isolated I/O?

Among the basic services needed by wimps, *isolated I/O* is a critical security primitive, but has remained an unmet challenge in both academia and industry.

An isolated I/O channel connects an input/output (I/O) device with an isolated software program that is trusted by the user, and protects the secrecy and authenticity of data transfers between them. The user of the trusted path should be able to verify the channel status, i.e., whether the isolated software or the device is the one the user intends to use, and also whether the channel is active.

The most commonly used isolated I/O channels are *user-oriented*, which are also known as

[1] TCBs include security kernels [14, 104], micro-kernels [70, 110], exokernels [33], virtual machine monitors [15, 21, 22, 25, 40, 54], micro-hypervisors [83, 112–114, 119, 127, 137], and separation/isolation kernels [49, 94, 98, 118].

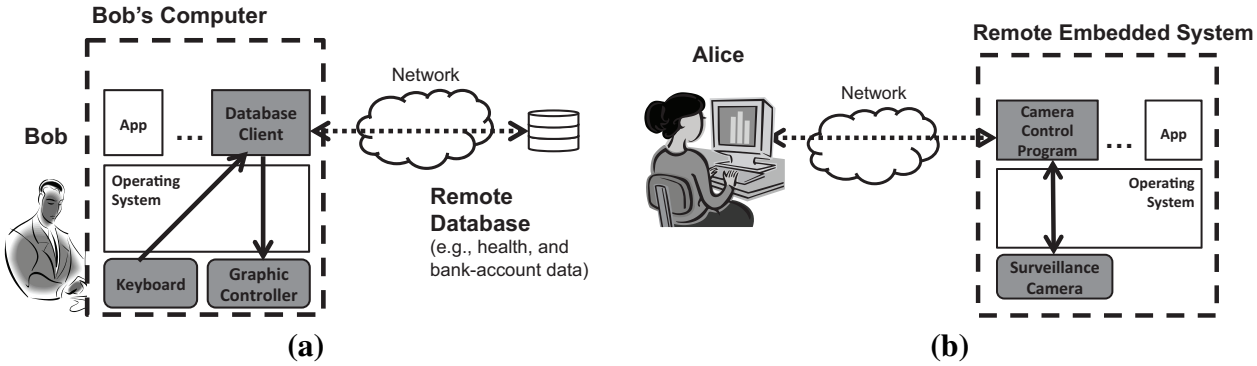


Figure 1.1: **Examples of Isolated I/O Channels.** (a) shows *user-oriented* isolated I/O channels between Bob's keyboard and display and the database client software in Bob's computer. (b) presents an *application-oriented* isolated I/O channel that connects a control program on Alice's remote embedded camera system with the camera itself.

the *trusted paths* in the Orange Book [30]. This channel connects a user's I/O devices (e.g., keyboard, mouse, and display screen) with a program trusted by that user. For example, in Figure 1.1(a), Bob is using his personal computer to access and modify his private data (such as health, investment, and banking data) in a remote database. Of course, Bob wants nobody to reveal or modify his private data input, output or transfer. A user-oriented trusted path is a necessary response to what Clark and Blumenthal call the "ultimate insult" directed at the end-to-end argument in system design [26]; namely, that a protected channel between a user's end-point and a remote end-point provides no assurance without a protected channel between the user himself and his own end-point. Without a trusted path, an adversary (e.g., malware in an OS) could surreptitiously record Bob's key strokes, modify his commands to corrupt the database, or display unauthentic output to trigger Bob's mis-behaviors.

In contrast, *application-oriented* isolated I/O channels are useful in a variety of application scenarios, such as the remote control of embedded real-time systems. As shown in Figure 1.1(b), the application-oriented channel protects the communication between the surveillance camera and the camera control program on the remote embedded system. Without this channel, the control program of the embedded system (and thus the operator Alice) would be unable to determine the true state of the surveillance camera and to effectively control it, in the presence of a malware-compromised embedded operating system. This is a particularly egregious problem in the control of mission critical systems, such as unmanned drones [108], uranium enrichment

centrifuges [35], and networked electricity generators [65].

Despite the incontestable necessity of isolated I/O channels as fundamental security primitives, for the past thirty years, attempts over the last thirty years to provide isolated I/O channels of peripheral devices to security-sensitive applications on commodity systems have been unsuccessful; viz. Sections 3.1 and 8. This thesis focuses on developing innovative architecture and methodology to address this particular challenge and opening up possibilities for implementing the isolated I/O channels in a myriad of high-assurance mission-critical applications in corporate, governmental, military, and consumer areas.

1.3 Thesis Overview

There is a pressing need for *providing isolated I/O channels to security-sensitive applications or software modules on commodity platforms*. With these isolated I/O channels, the security-sensitive code, which is isolated from the untrusted commodity operating system and other applications, is able to transfer data to/from commodity peripheral devices with I/O data secrecy and authenticity remaining protected.

Thesis Statement. *Based on a root-of-trust mechanism for code isolation, small and simple add-on security components can provide isolated I/O channels to security-sensitive applications on commodity platforms.*

Specifically, we presents a *general-purpose, simple, human-verifiable* I/O isolation system that coexists with a *commodity* operating system. The system is *general* in that it allows arbitrary software code to be isolated from arbitrary OSes it runs upon and to be connected with arbitrary commodity peripheral devices. It is *simple* in that the TCB of the system is small and simple enough to achieve complete and accurate definitions of the adversary and security properties of the system, and to facilitate formal verification of those properties. It is *human-verifiable* in that a human using the system can verify that the desired isolated I/O channel is in effect (e.g., that the keyboard is acting as a secure channel to a banking program on that machine). The system is truly useful only when human users can securely communicate with it and when they can be sure that the system is up and running whenever the users need it. This system is *commodity* in that it does not require any intrusive modification or restructuring of the coexisting OSes. The

compatibility with commodity off-the-shelf platforms allows users to enjoy the functional richness and convenience of commodity platforms, and to utilize high-assurance security-sensitive applications on-demand.

Instead of supporting secure I/O services in a restructured or redesigned OS or bloating an underlying TCB that guarantees wimp-giant code isolation with the I/O services, we propose a security architecture for *on-demand* isolated I/O channels, which enables security-sensitive applications (i.e., wimps) to dynamically connect to diverse peripheral devices of unmodified commodity OSes. The new architecture is based on a *wimpy kernel*, a trusted "add-on" that constructs on-demand I/O channels for wimp applications and securely composes with the commodity OS; i.e., it relies on the I/O services in the commodity OS but only after efficiently verifying their results. The wimpy kernel does not increase the size and complexity of the underlying TCB, modify its security properties or increase the formal verification effort. The wimpy kernel removes a wimp app's direct interfaces with 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.

We apply two innovative methods to dramatically reduce the size and complexity of the wimpy kernel for high assurance. 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 configuration of the entire USB controller-hub-device hierarchy is initialized by the untrusted OS and handed over to the wimpy kernel when a wimp app attempt to use a USB device. The wimpy kernel verifies the hierarchy efficiently without enumerating any other device in the hierarchy. Second, we further minimize the wimpy kernel by de-privileging and *exporting* drivers and driver-subsystem code to security-sensitive applications, and implementing wimpy-kernel checks that *mediate* 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 application themselves or wimpy kernel.

In short, this thesis makes 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 and securely composes with the untrusted giants, the underlying TCB, and the wimp apps.
- We illustrate the detailed design of the wimpy kernel, and show how to use outsource-and-verify and export-and-mediate methods to minimize its code base. The reduction of the wimpy kernel is dramatic; i.e., 99% of the Linux USB code is removed.
- We implement the wimpy-kernel-based I/O isolation system for the PCI and the USB subsystems of Linux and evaluate its performance. Experimental results indicate that the system incurs acceptable performance overhead.
- We instantiate two security-sensitive applications, namely secure user interface (a.k.a trusted path) and trustworthy corporate key management system, to show the practicality of the proposed on-demand I/O isolation architecture.

Thesis Organization. The remainder of this thesis includes the following chapters: Chapter 2 defines the security properties achieved in the I/O isolation system and its adversary model. Chapter 3 analyzes the insufficiency of past attempts to provide isolated I/O channels, and highlights the fundamental challenges of achieving this goal. Chapter 4 introduces the idea of on-demand I/O isolation and its advantages and unique challenges, and overviews the wimpy-kernel-based architecture we propose to instantiate this idea. Chapter 5 illustrates the concrete design of multiple aspects of the I/O isolation system. Chapters 6 and 7 present the implementation and evaluation of the secure user interface application (trusted path) and trustworthy corporate key management system, respectively. Chapter 8 summarizes the related work. Chapter 9 discusses various software and hardware extensions of the system, and Chapter 10 concludes the thesis.

Chapter 2

Problem Definition

This chapter defines security properties achieved in I/O isolation systems and the adversary model.

2.1 Desired Properties

A practical and trustworthy I/O isolation system must have the following properties:

Generality. The generality of the I/O isolation system is three-fold: First, the security sensitive applications or software modules can be *any* program that is isolated from the untrusted OS, not just a small number of special programs (e.g., log-in commands and administrative commands) as in early I/O isolation systems [12, 14, 28, 30, 34, 46, 52, 104]. Second, the devices that are connected to the security-sensitive applications can be *any* commodity character device (e.g., USB thumb drives, embedded cameras and accelerometer sensors on mobile phones), not just a small number of user-oriented I/O devices (e.g., a keyboard, mouse, or video display), and not special devices that have data encryption or authentication capabilities. Third, isolated I/O channels are independent of the coexisting OSes; i.e., they can be used with monolithic (e.g.,Windows), kernel-based (e.g., Linux, SELinux), or virtualized (e.g., Xen, Xoar [27]) OSes. This enables isolated I/O channels to function *end-to-end* across different platform configurations; e.g., from an application running on the x86-based Windows OS on a PC to an application running on ARM-based Android OS on a smartphone.

Small and Simple. An I/O isolation system must employ a *small and simple* Trusted Computing Base (TCB). This enables the isolation system to be fully verified by formal methods, and

thus to achieve higher security assurance. For example, many OSes offer isolated I/O channels in the form of secure attention sequences (e.g., Ctrl+Alt+Del) to initiate user communication with the OS. The TCB for the isolated I/O channels is the entire OS, which is large and routinely compromised. These isolated I/O channels, though users may be forced to trust them in practice, are not adequately *trustworthy*. Note that a small and simple code base is a necessary but insufficient condition for formal verification. This arises from the practical constraints of state-of-the-art assurance methods. To date, even the seemingly simple property of address-space separation has only been formally proved for very small code bases; i.e., fewer than 10K lines of code [39].

Human-Verifiable. An I/O isolation system must allow human users to verify the status of the system; i.e., the users must know whether the isolated channels are in effect, and whether the protected devices and program are the ones that the users desire. For example, in a secure user interface application, a user needs to verify that the keyboard is securely connected to the banking application on his/her machine when he/she is typing in sensitive information, such as passwords and transaction amounts. The user also needs to verify that what he/she sees on the graphic display of the machine is exactly what the banking program displays. Without human-verifiability, the user might have no idea what is really happening to the banking application on a malware-infected machine and may lose control of his/her banking account. Similar problems apply to the application-oriented isolated I/O channels.

Commodity. An I/O isolation system must coexist with commodity OSes and platforms such as personal computers, laptops, mobile phones, tablets, and embedded systems. The I/O isolation system must *not* rely on special or extensively modified or restructured OSes, nor does it require dedicated and complicated devices and appliances. For example, some I/O isolation systems [36, 111] only work with dedicated microkernel-based OSes. These systems are difficult to deploy in the mass market and are often limited to a few applications and implementation scenarios.

2.2 Adversary Model

We consider an adversary that has compromised the operating system (OS), which we henceforth refer to as the *compromised OS*. A compromised OS can access any system resources that it controls (e.g., it can access any physical memory address and read/write any device I/O port) and

break any security mechanisms that rely on it (e.g., process isolation, file system access control). The adversary can then leverage the compromised OS to actively reconfigure any device (e.g., it can modify a device's memory mapping, or change the operating mode of a device) and induce it to perform arbitrary operations (e.g., trigger interrupts, issue DMA write requests) using any I/O commands. We say *manipulated device* to reference the devices that the adversary is controlling.

In addition, a *rogue* or *compromised* isolated application may attempt to escalate its privilege by manipulating the interfaces with the I/O isolation system or configuring the isolated devices. It could also try to break the application isolation (e.g., process isolation, file system controls) enforced by the OS, or control its devices to compromise the execution or data of the OS, the I/O isolation system or other isolated applications.

We do not address attacks on device firmware and hardware in this thesis. We assume that all chipset hardware and peripheral devices are not malicious. They do not contain Trojan-Horse circuits, microcode, or malware that would violate the I/O isolation in response to an adversary's surreptitious commands. We assume that the devices operate exactly according to their specifications and will not actively perform unintended operations, such as intercepting bus traffic that is not destinated to them, remaining awake after receiving a "sleep" command, or writing data to a memory address that is not specified in any DMA command.

In addition, we do not consider side-channel attacks that attempt to extract useful information about the isolated I/O channel data (e.g., timing or power analysis). We envision that, if necessary, security-sensitive application would employ countermeasures to defend against such attacks. Denial-of-service attacks are also out of scope; we seek only to guarantee the secrecy and authenticity of the isolated I/O channel.

Chapter 3

Challenges

3.1 Insufficiency of Previous Systems

Past attempts to provide isolated I/O channels of peripheral devices to security-sensitive applications on commodity systems have been unsuccessful. In the following sections and in Table 3.1, we detail the related work and analyze its insufficiency..

Code Isolation Systems. Recent research has demonstrated the removal of the OS from the TCB for security-sensitive applications or code modules [13, 21, 22, 24, 54, 80, 83, 84, 100, 112–114, 127]. These mechanisms enable the safe coexistence of the commodity OS and security-sensitive code. A compromised OS and manipulated devices cannot interfere with the execution of the isolated code, nor can they reveal or tamper with any run-time data of the isolated code.

While this work is necessary for isolating security-sensitive applications and the drivers of their devices, it fails to provide a mechanism to enable isolated modules to communicate with devices with high security guarantees. For storage and networking devices, the isolated modules can simply outsource the I/O services of these devices to the untrusted OS by using data encryption [21, 22, 24, 29, 43, 47, 48, 54, 80], because storage and networking devices transfer data in bulks. Specifically, the wimps can use authenticated encryption techniques to encrypt the data and send the encrypted data to the storage and networking subsystem of the untrusted OS. When the wimps get the data back, they can easily verify the authenticity of the data.

However, providing isolated I/O channels of the rest of the devices, *character devices*, to the isolated module remains a unsolved challenge. Character devices (e.g., video/audio, camera, sen-

Table 3.1: **A Comparison of Existing I/O Isolation Systems.** ✓* represents that only some of the systems in the category has the property.

I/O Isolation System	Generality	Small & Simple TCB	Human-Verifiability	Commodity
W/ limited device support	✗	✓*	✗	✗
W/ static device partition	✗	✓	✓*	✓*
On commodity OS	✓	✗	✗	✓
W/ device virtualization	✓	✗	✗	✓
On restructured OS	✓	✓	✗	✗
W/ special devices	✗	✓	✓*	✗
Our system	✓	✓	✓	✓

sors, user input devices) are pervasive. They account for 57% of device types and 52% of driver code in Linux [66]. They are used to connect software with human and physical world. More importantly, isolation by data encryption does not work for character devices because character devices transfer real-time data in bytes and typically do not have cryptographic capability.

Achieving isolated I/O channels of character devices on commodity platforms is a vastly more challenging exercise than program and device driver isolation. The diversity and complexity of I/O devices far exceeds that of most non-I/O objects (e.g., code and data memory); hence, the complexity of mediating and isolating access to I/O devices also vastly exceed that of non-I/O isolation. Moreover, the I/O hardware of present-day commodity platforms is designed to scale the multiplexing of very diverse peripheral devices and to provide extensive sharing of physical resources, but it includes insufficient support for I/O channel isolation; viz. Section 3.2.

Systems with Limited Device Support. For the past three decades, a few systems have implemented isolated I/O with *limited configurations* on boutique computer systems. These systems comprise a small number of user-oriented I/O devices (such as a keyboard, mouse, video display, or printer) and a small number of programs of a Trusted Computing Base (TCB) [14, 30, 46, 52, 104, 133]. Limited configurations and their use within small and special evaluated TCBs enable significant assurance of I/O isolation. Due to this high level of trustworthiness, these systems do not intend to provide extra mechanisms for human-verifiability to the user.

Some more recent systems provide isolated I/O channels within system TCBs [23, 37, 94,

137] but only for a few selected devices. Even limited support for a 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 [127] would more than double its code-base size and significantly increase its complexity ; e.g., it would introduce concurrency in the serial micro-hypervisor code since it would require I/O interrupt handling.

Systems with Static Device Partition. Other attempts statically allocate selected peripheral devices to isolated system partitions [49, 69, 98, 118, 119, 129] by allocating devices to different system partitions. The partitioning is statically defined at system configuration time and is difficult to change during system run-time, thus it achieves channel isolation at the cost of losing on-demand (e.g., plug-and-play) capabilities of commodity systems. Some of these system only support a limited number of I/O devices in special application scenarios; e.g., network interface cards, storage devices, and graphic cards on a cloud platform [69, 119].

Systems on Commodity OS. Many OSes offer isolated I/O in the form of secure attention sequences–key-combinations (e.g., Ctrl+Alt+Del) to initiate user communication with the OS. The Trusted path on the DirectX system [76] and the Trusted Input Proxy system [16] reserve dedicated areas of the screen to output the identity and status of the current applications. The TCB of these systems include the OSes, which are large and routinely compromised. These isolated I/O channels, though they support a variety of commodity devices, are not adequately trustworthy.

Device Virtualization and Pass-through. 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 [15]. It then continually evolved to hypervisors with disengaged privileged domains [27], then to hypervisors with isolated driver domains [40, 87], and finally to hypervisors with device pass-through support (e.g., Xen, KVM, and [82]) or para-passthrough support (e.g., BitVisor [112]), which assign I/O devices to a specific guest VM. However, applications in the guest VMs still communicate with virtualized or pass-through devices via the untrusted guest OS on which they run, which still implies that a huge code base has to be trusted for isolated I/O. To make things worse, virtualizing hardware devices of commodity OSes also introduces a huge code base (e.g.,

hypervisor, privileged domain, and/or driver domains) ; viz. Section 8.3.3 and Table 8.3.2.

Systems on Restructured OS. Trusted-window managers (such as Nitpicker [36], EROS window manager [111]) mediate and protect user input and output on nicely designed micro-kernel-based OSes. The micro-kernels provide the near-minimum mechanisms needed to implement an operating system (e.g., low-level address space management, thread management, and inter-process communication), while traditional OS functions (e.g., device drivers and file systems) are modularized and removed from the micro-kernels and run in the user space [42, 117]. Thus, ideally, isolated I/O channels on these OSes only rely on the small micro-kernel, the necessary I/O components, and the drivers of the isolated devices, which represents a dedicated but very small TCB. However, the technical and business-related complexities of restructuring a commodity OS to a micro-kernel-based design are immense. In addition, these I/O isolation systems do not claim to provide human-verifiability to their users.

Special Devices. Other attempts to isolate I/O channels rely on special hardware devices equipped with data encryption capabilities [72, 85, 103] to establish cryptographic channels to applications [53, 77, 85, 130]. This approach excludes commodity devices, which lack encryption capabilities, and adds TCB complexity by requiring secure key management for the special devices. It also raises fundamental usability concerns for commodity platforms. For example, how can a user securely set or change the secret key within an isolated program without using some isolated I/O channels of user interface devices to reach that program?

3.2 Complexity of I/O Isolation

As suggested in Section 3.1, isolating the security-sensitive code and device drivers does not sufficiently guarantee I/O isolation because we still need to isolate the devices used by the security-sensitive code from the entire complex hardware architecture. Modern I/O architecture of commodity platforms is designed to multiplex diverse peripheral devices among different applications and to provide extensive sharing of physical I/O resources, but it includes insufficient supports for I/O channel isolation.

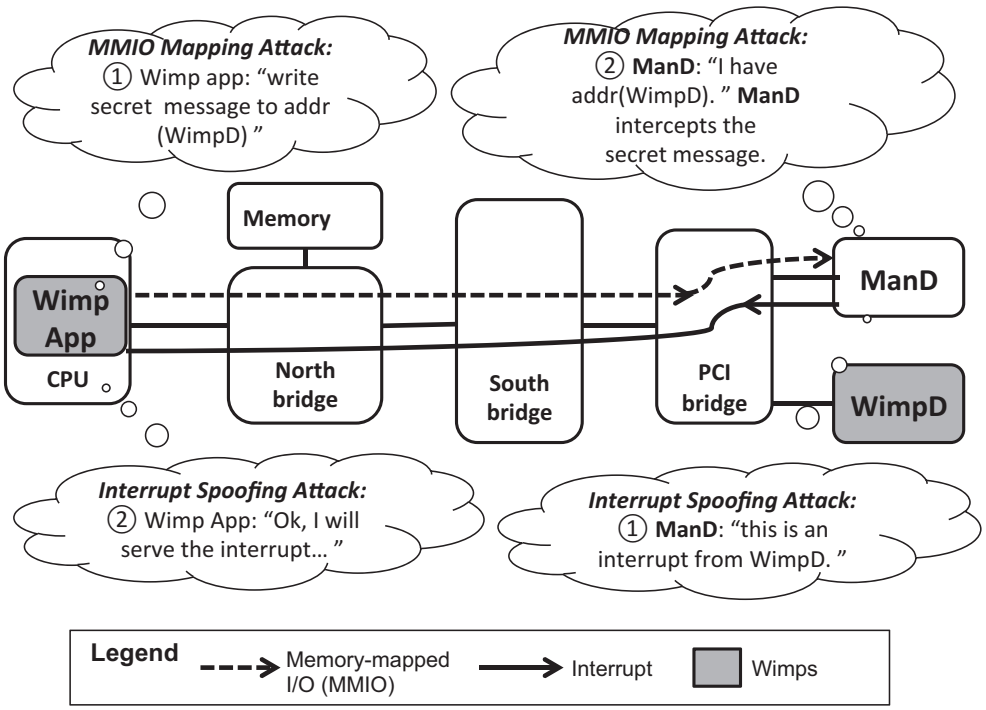


Figure 3.1: **Attacks against Shared Device I/O Resources.** A manipulated device (ManD) launches an *MMIO mapping attack* and an *interrupt spoofing attack* against the path between the isolated code and its device (WimpD).

3.2.1 Shared Device I/O Resources

There are substantial I/O resources (e.g., memory, I/O ports, interrupts) on commodity platforms that are shared by the devices of the isolated code and other devices. To provide I/O isolation, it is necessary to protect the access to the shared hardware resources used by the devices of the isolated code from the untrusted OS. We identify two categories of shared hardware resources on current commodity platforms, and highlight attacks that exploit these shared resources to breach the I/O isolation.

I/O Memory and Ports. The memory-mapped I/O (MMIO) regions of device on commodity platforms share the same physical memory address space and are configurable by system software, such as an OS. A compromised OS (or device driver) may launch an attack, which we call an *MMIO mapping attack*, to breach the isolation of device memory. The compromised OS can intentionally configure the MMIO region of a device (ManD in Figure 3.1) to overlap the MMIO region of the protected device (WimpD in Figure 3.1). During to this mis-configuration, ManD

can then intercept or tamper with the MMIO access of the wimp app to the WimpD. For example, the malicious OS may map the internal transmission buffer of a network interface card over top of the frame buffer of a graphics card, so the display output may be directly sent to a remote adversary via the network. Note that the typical mechanisms for protecting CPU-to-memory access (e.g., by configuring AMD Nested Page Table (NPT) [8]) or DMA (e.g., leveraging Intel VT-D [61]) *cannot* defend against this "MMIO mapping attack". Similar attacks apply to I/O ports. A compromised OS can intentionally configure the I/O port(s) of the protected device to conflict with those of other devices and thus violate the isolation of I/O ports.

Interrupts. Software-configurable interrupts such as Message Signaled Interrupts (MSI) and Inter-processor Interrupts (IPI) share the same interrupt vector space with hardware device interrupts. By modifying the MSI registers of the ManD, a compromised driver of the ManD can spoof the MSI interrupts of the WimpD. As shown in Figure 3.1, the unsuspecting driver in the isolated code may consequently perform incorrect or harmful operations by processing spoofed interrupts from the ManD.

3.2.2 Shared I/O Hardware Configurations

Peripheral devices are interconnected by a variety of I/O chipset hardware such as northbridge, southbridge, and bus controllers, as shown in Figure 3.1. The chipset hardware physically transfers the I/O data and commands (e.g., keyboard scan code, DMA write requests) and interrupts between the protected devices and the CPU and memory used by the isolated software. For the isolation of I/O channels, it is therefore necessary to protect the configurations of the shared chipset hardware and to mediate data transfers to the devices.

For example, USB devices are connected to the computer via a hierarchy of USB host controller and hubs. Instead of directly communicating with USB devices' I/O resources (Section 3.2.1), the program running in CPU must indirectly perform I/O operations to the devices via the USB bus hierarchy. I/O channel isolation for this USB subsystem is the most complex among the diverse bus subsystems (e.g., PCI, USB, Bluetooth, HDMI) in modern I/O architectures [66], due to three complexity aspects: It (1) mixes control and data channels, (2) uses (untrusted) software to maintain the device hierarchy, and (3) uses software-initialized device addresses (in versions earlier than USB 3.0). The configurations of device hierarchy, address and

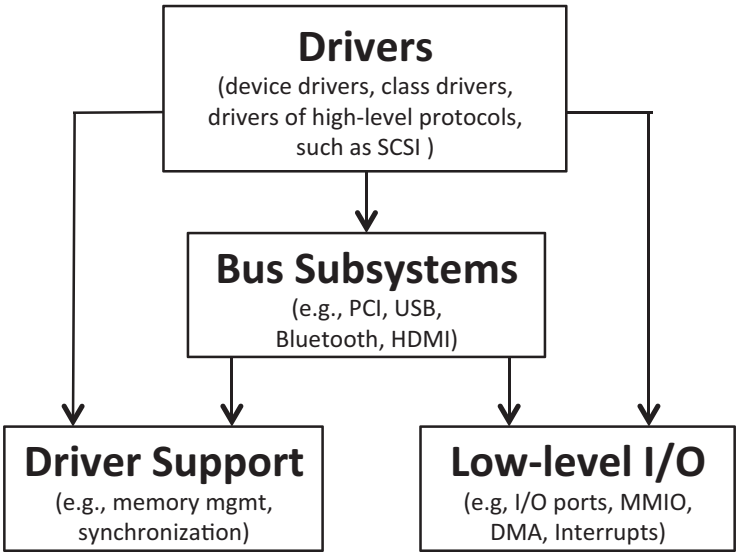


Figure 3.2: **I/O Code Base in Commodity OSes.**

channels in the USB bus subsystem are initialized and managed by the OS, which is untrustworthy. Thus a compromised OS can misconfigure USB host controller, hubs and other devices that shares the same USB bus with the isolated devices, in order to launch subtle attacks against the I/O isolation; viz., the USB address overlap attack and the remote wake-up attack as discussed in Section 5.2.2. Using these attacks, a USB device manipulated by the compromised OS can intercept secret messages transferred between the isolated code and its devices, or inject fake data to tamper with the communication integrity. All in all, I/O channel isolation systems must control the multiplexing of complex bus subsystems for different devices.

3.2.3 Complex I/O Code Base

Due to the diversity and complexity of commodity peripheral devices and chipset hardware, modern device drivers constitute a huge body of code in the OS [66]. The drivers are also tightly integrated with the OS kernel and depends on a variety of utilities from the OS, such as memory management, synchronization and scheduling, bus subsystem management, and interactions with low-level I/O resources, as shown in Figure 3.2.

However, it is insecure and incorrect to include the large and complex I/O code base in the I/O isolation system's TCB. It is insecure because the added I/O code base will greatly swell the size of the TCB and significantly increase its complexity; e.g., interrupt handling code would

introduce concurrency into the TCB. This dramatically increases the complexity of the formal proof of TCB security properties, and thus greatly undermines the security assurance of the I/O isolation system. It is incorrect in that the I/O code base used by the isolated security-sensitive code will conflict with the one in the commodity OS when they both attempt to manage the same set of chipset hardware.

The challenge, therefore. is to find standard and scalable methods to reduce and simplify the I/O code base that is introduced in the I/O isolation system, and to securely and correctly cope with the I/O subsystem in the coexisting commodity OS.

Chapter 4

On-demand Isolated I/O

To address the challenges described in Chapter 3, this thesis proposes a unique *on-demand I/O isolation* idea and a *wimpy-kernel-based* architecture to instantiate this idea. This chapter first introduces the on-demand I/O isolation and its advantages against other I/O isolation models (Section 4.1). We then describe the security challenges and properties associated with the on-demand model in Sections 4.2 and 4.3, respectively. Section 4.4 provides an overview of the wimpy-kernel-based architecture.

4.1 Why 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 [137]. Cryptographically enabled channels, device virtualization, and pass-through of hardware devices become unnecessary.

Second, it enables trusted audit and control of physical devices without stopping and restart-

ing 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 to 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 recompilation.

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 the static device allocation and virtualization models. In the first model the OS cannot configure devices in wimp partitions, and in the second it does not have direct access to hardware devices.

4.2 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 misconfigure 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 [66], which become exposed to subtle isolation attacks; viz., the USB address overlap attack and the remote wake-up attack of Section 5.2.2. Hence, I/O channel isolation must 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 Code Base. The opportunity for minimizing I/O kernel size and complexity created by the on-demand I/O isolation model (viz., Section 4.1) poses a significant design question: 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 6.2.3 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 a trusted I/O kernel with the rest of the TCB must not diminish the existing assurance; i.e., it must not invalidate the TCB's security properties and their proofs.

4.3 Security Properties

The security challenges described above indicate that the typical giant-wimp isolation model must be augmented with additional security properties. These properties specify how the trusted I/O kernel interacts with wimp applications, giants, and the underlying TCB to provide on-demand isolated I/O channels to peripheral devices. These properties are presented below.[1]

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.

[1] The similarity of these security properties to those of a traditional reference monitor [10] 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.

P3. Minimization of the Trusted Codebase. This property implies that (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.

4.4 System Overview

To fulfill all three security properties of on-demand isolated I/O systems, we propose an add-on security architecture based on a *wimpy kernel* (WK), which composes with the underlying TCB, the untrusted OS, and wimp applications [139]. This section illustrates this architecture and highlights the code base minimization methodology of the wimpy kernel. Note that the underlying TCB used in this thesis is a slightly modified version of XMHF [127] – 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.

4.4.1 Wimpy Kernel: An Add-on Trustworthy Component

As shown in Figure 4.1, 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 environments 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 [45]. The *mHV*, which is a slightly modified version of XMHF [127], 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.

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 the 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 4.3. *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

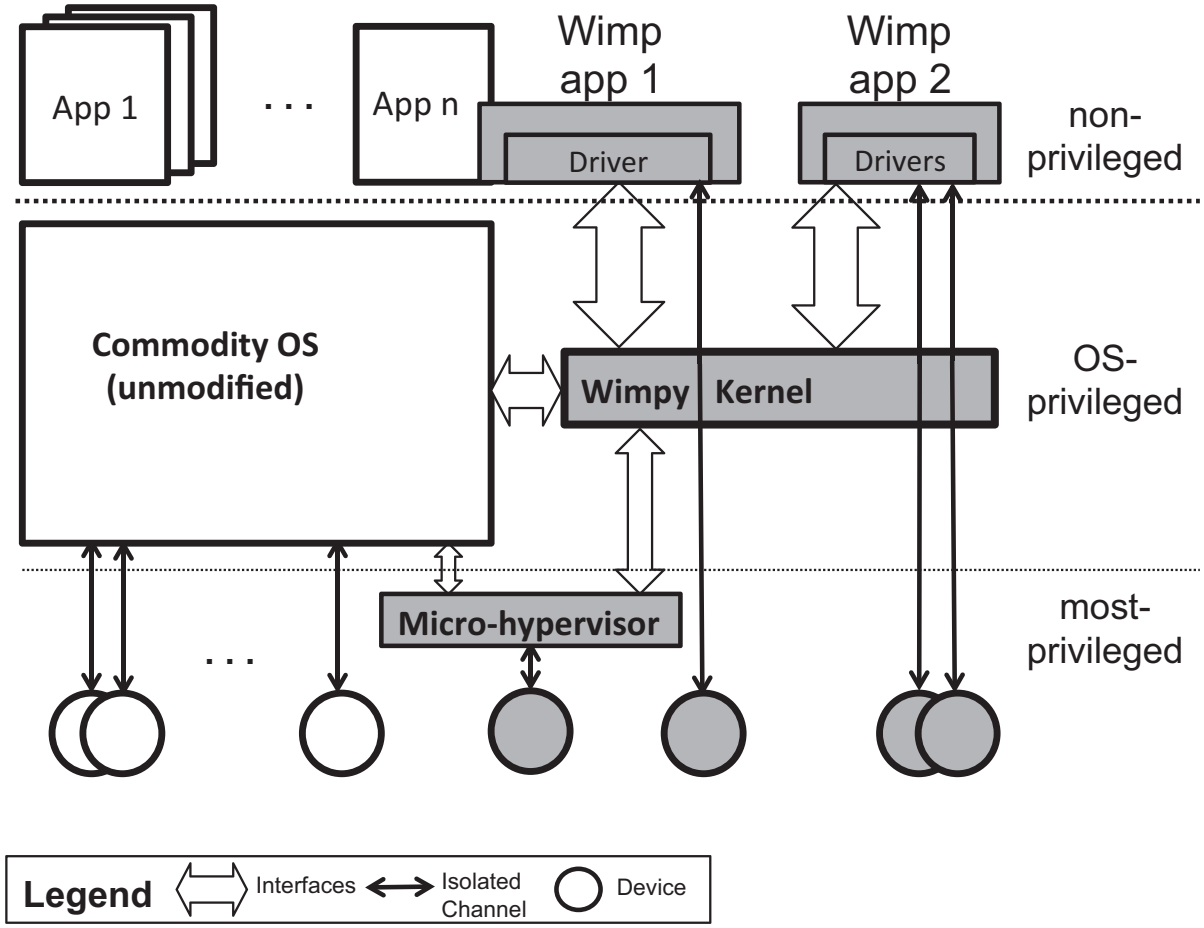


Figure 4.1: **Overview of the I/O Isolation Architecture.** The grey area represents the trusted code base of wimp applications.

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 5.5.

Figure 4.1 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 [127]. Second, it must compose with the untrusted OS (giant) to remain small and simple; viz. Property 3 (part 1). Specifically, the wimpy kernel outsources its most complex functions to the untrusted OS whenever it can efficiently verify their results. 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.

4.4.2 Composition with the Underlying TCB

The composition of the wimpy kernel with *mHV* has three important characteristics: it preserves *mHV*'s wimp-giant isolation model; it avoids the 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 [8, 62], device Direct Memory Access (DMA) control [7, 61], and sealed storage and attestation root-of-trust [50].

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

Third, the wimpy kernel does not invalidate *mHV*'s security properties or 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.[2]

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 as well as more recent micro-hypervisor designs [23, 83, 112–114]. Considering an alternative underlying TCB, a micro-kernel such as seL4 [70], the complexity of the micro-hypervisor is demonstrably lower than that of the formally verified seL4 microkernel [71]. Specifically, seL4 implements more complex abstractions with richer functionality than our hypervisor. For example, seL4 supports full-fledged threads and interprocess communication primitives (as opposed to simple locks), memory allocation (as opposed to mere memory partitioning), and capability-based object addressing (as opposed to merely address space separation via paging).

4.4.3 Composition with the Untrusted OS and Wimp Apps

To assure the I/O channel isolation, the wimpy kernel needs to control all I/O hardware that is shared by devices of the untrusted OS and wimp apps. 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, including 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 (see Figure 4.2), namely outsource-and-verify functions (whose various *cryptographic* versions have been used since the late 1970s [21, 22, 24, 29, 43, 47, 48, 54, 80]) and export-and-mediate code [64, 67, 105]. However, neither method has ever been used for high-assurance, on-demand I/O isolation kernels for commodity platforms. I/O isolation was either done in security kernels for a few simple devices and not on demand, or was performed outside security kernels and not minimized for high assurance; viz., related

[2]Wimpy kernel uses similar mechanisms to those of references [40, 137] to isolate the interrupts of OS- and wimp-app-controlled devices and bypass *mHV*.

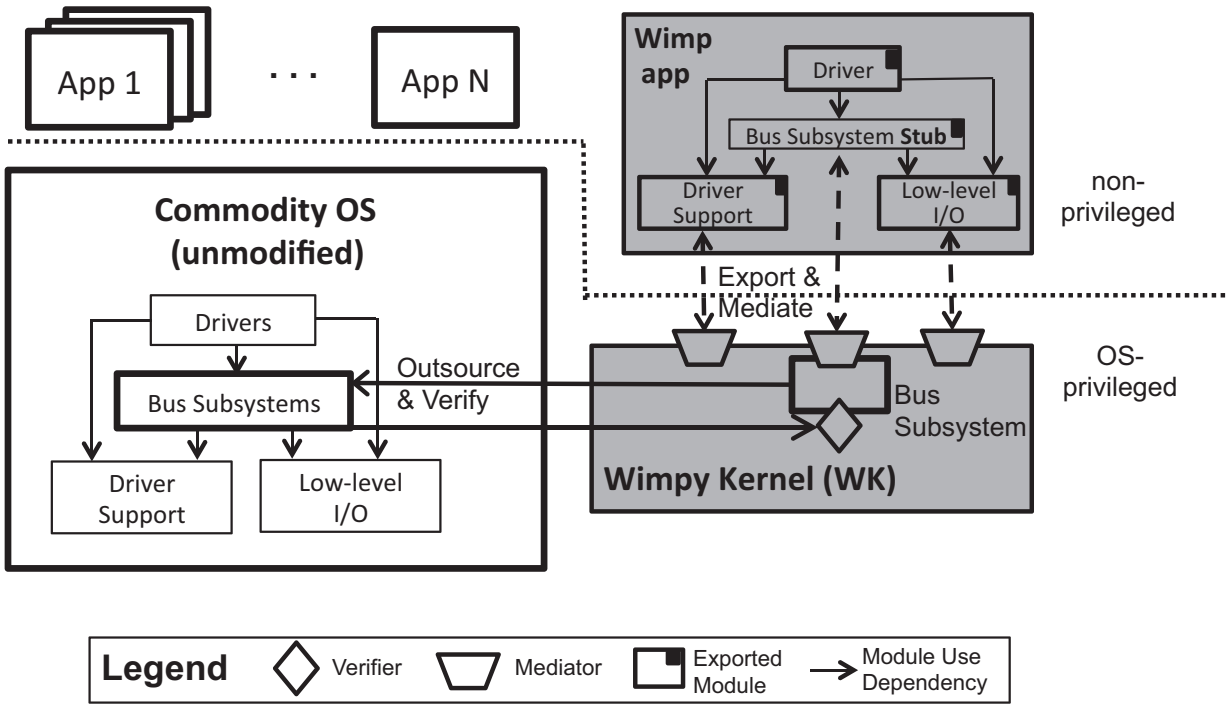


Figure 4.2: **Outsourced Functions and Exported Code of the Wimpy Kernel.**

work in Section 8.3. 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 6.2.3.1.

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 4.2. 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 misconfigure 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 5.2.2. 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 operations. In short, the outsource-and-verify approach enables us to substantially decrease the code base of the wimpy kernel and also avoid reliance on the untrusted OS.

Export-and-Mediate. We further minimize the wimpy kernel code base by exporting device drivers and bus subsystem code to isolated wimp applications[3] which would otherwise have to be supported in the wimpy kernel itself; e.g., the Bus Subsystem Stub of Figure 4.2 denotes bus subsystem code exported by the wimpy kernel to a wimp app. In Section 5.2.3, 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 4.2. Some code dependencies, such as those of synchronization functions for device multiplexing, disappear in the on-demand I/O model because the devices are exclusively owned by the wimp apps during their run time. Other functions, such as low-level I/O operations, are de-privileged to the wimp applications and may require mediation of the wimpy kernel after code export. For example, memory management functions are reimplemented for drivers in wimp apps without explicit mediation, since the wimpy kernel and the underlying TCB already isolate chunks of memory and pass it to wimp apps before wimp app execution. We illustrate how wimpy kernel performs low-level I/O resource isolation and driver support code exporting in Sections 5.1 and 5.3, respectively.

Generality of the Design Methods. The outsource-and-verify method, which we illustrate with the USB subsystem in this thesis, 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 [66]. 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

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

to reach a specific device, which can be directly accessed by the verification algorithm. In addition, for power management code (which comprises 7.4% of driver code on average), verifying the power state of bus controller and hubs/bridges are general operations for 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 various types of 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. This code can be 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 does *not* need to mediate wimp app operations that affect functional properties or the 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 5.3) applies to all devices and buses. However, drivers for different types of devices and buses may have different dependencies on support code.

Chapter 5

System Design

This chapter first presents how to use the outsource-and-verify and the export-and-mediate design methods to perform on-demand I/O isolation and to minimize low-level I/O functions (Section 5.1), bus subsystems (Section 5.2), and driver support code (Section 5.3) in the wimpy kernel. We then describe the wimp-OS communication service provided by the wimpy kernel (Section 5.4) and major operations in the wimp application's lifecycle (Section 5.5). After that, we discusses the system design to support user verifiability and the development effort of wimp apps in Sections 5.6 and 5.7, respectively.

5.1 Isolating Low-level I/O Resources

This section illustrates the detailed mechanisms of isolating the low-level I/O resources (e.g., I/O ports, device-memory, configuration space, interrupts) of the protected devices from those of the other devices.

5.1.1 Protection of I/O-port Access

Software programs use the IN/OUT family of CPU instructions to read/write data from/to the I/O ports of devices. To control access to device I/O ports, the wimpy kernel must prevent the device-port-mapping conflicts that may be intentionally created by the compromised OS, and mediate the I/O port access from both the wimp apps and the untrusted OS.

Preventing Port-mapping Conflicts. The compromised OS can re-map a manipulated device's I/O ports to overlap those of a wimp device. Read/write access from/to those I/O ports will have unpredictable results, since both the manipulated device and wimp device will respond to the I/O

access command. Thus, a manipulated device (and a compromised OS) may potentially intercept or tamper with the I/O port data of the wimp app.

To address this problem, the wimpy kernel isolates the wimp device's I/O ports from the shared I/O port space of the platform. Specifically, before executing the wimp app, the wimpy kernel scans through all I/O port mappings of the chipset hardware and enumerates all plug-and-play (PnP) devices to detect their configured I/O ports. For example, the wimpy kernel accesses the PCI configuration spaces of all PCI devices in the system and parses their I/O port settings via the PCI Base Address Registers in the configuration spaces. If any of the above port settings conflict with those of the wimp device, the wimpy kernel issues an exception to the wimp app. The wimpy kernel must protect all I/O port mappings in the device configuration space from modification by a compromised OS or manipulated devices throughout the run-time of the wimp app. We defer the details of scanning and protecting device configuration space to Section 5.1.3.

Mediating I/O-port Access. The wimpy kernel configures the I/O port-access-permission bitmap of the Task State Segment (TSS) used by the wimp app, so that the wimp app can only access the I/O ports of its associated wimp device(s). To prevent the untrusted OS from accessing the I/O ports of the wimp device(s), the wimpy kernel invokes the *mHV* to configure the I/O port-access-interception bitmap in the hypervisor control block that describes the OS's execution environment (a standard feature of x86 hardware virtualization support [8, 58]). Any unauthorized access from the OS to the I/O ports of the wimp device(s) will be trapped and filtered out by the *mHV*.

5.1.2 Protection of I/O-memory Access

There are two methods for the wimp app to interact with the wimp device via physical memory space: Memory Mapped I/O (MMIO) and Direct Memory Access (DMA). The compromised OS and manipulated devices can breach the isolation of the wimp device-associated physical memory regions in three ways: via an MMIO mapping attack, through unauthorized CPU-to-memory access, or via unmediated DMA.

Preventing MMIO Mapping Attacks. The compromised OS can launch a *MMIO mapping attack* on the MMIO memory regions associated with a wimp device, as shown in the left half of Figure 5.1. To defend against this attack, the wimpy kernel must ensure that all MMIO memory

ranges used by the chipset hardware and other peripheral devices do not overlap with those of the wimp device.

Before executing a wimp app, the wimpy kernel scans through all MMIO memory mappings specified by the chipset hardware and enumerates all PnP devices to discover their MMIO memory ranges (e.g., by checking the PCI Base Address Registers in the PCI configuration space). If overlaps with the wimp device's memory ranges exist, an MMIO mapping attack may be in progress, and the I/O isolation may be violated. Upon detection, the wimpy kernel issues an exception to the wimp app. During the wimp app's execution, the wimpy kernel must prevent the compromised OS and manipulated devices from modifying the MMIO memory mappings of all devices, by calling the *mHV* to protect the device configuration space. We elaborate on our mechanisms for protecting device configurations in Section 5.1.3.

Preventing Unauthorized Memory Access. The compromised OS can directly access the wimp device's MMIO and DMA memory regions or manipulate a device outside the isolated I/O channels to issue unauthorized DMA requests to access those regions. The wimpy kernel and *mHV* defend against these attacks by leveraging standard features for x86 CPU and chipset hardware virtualization support. For example, the *mHV* configures the access permissions in Nested Page Tables (or Extended Page Tables) [8, 58] to prevent unauthorized CPU-to-memory access. The *mHV* also sets up the IOMMU [7, 61] to prevent other devices from performing DMA access to the memory regions associated with the wimp app. Note that IOMMU protection relies on the assumption that it can correctly identify DMA requests from the devices. We discuss DMA request ambiguity and its influence on I/O isolation in Section 9.1.

5.1.3 Protection of Device Configuration Space

A fundamental building block of our prevention mechanisms against I/O port conflicts (Section 5.1.1) and MMIO mapping attacks (Section 5.1.2) is protecting the device configuration space. Specifically, the hypervisor intercepts all accesses to the device configuration space of the wimp device(s) throughout the execution of wimp applications. The *mHV* grants the wimp app *only* the access permissions to the configuration space of its devices, and prevents the OS and manipulated devices from modifying the I/O ports and MMIO memory mappings of *any* device.

For the x86 I/O architecture, the device PCI/PCIe configuration space is accessed via special

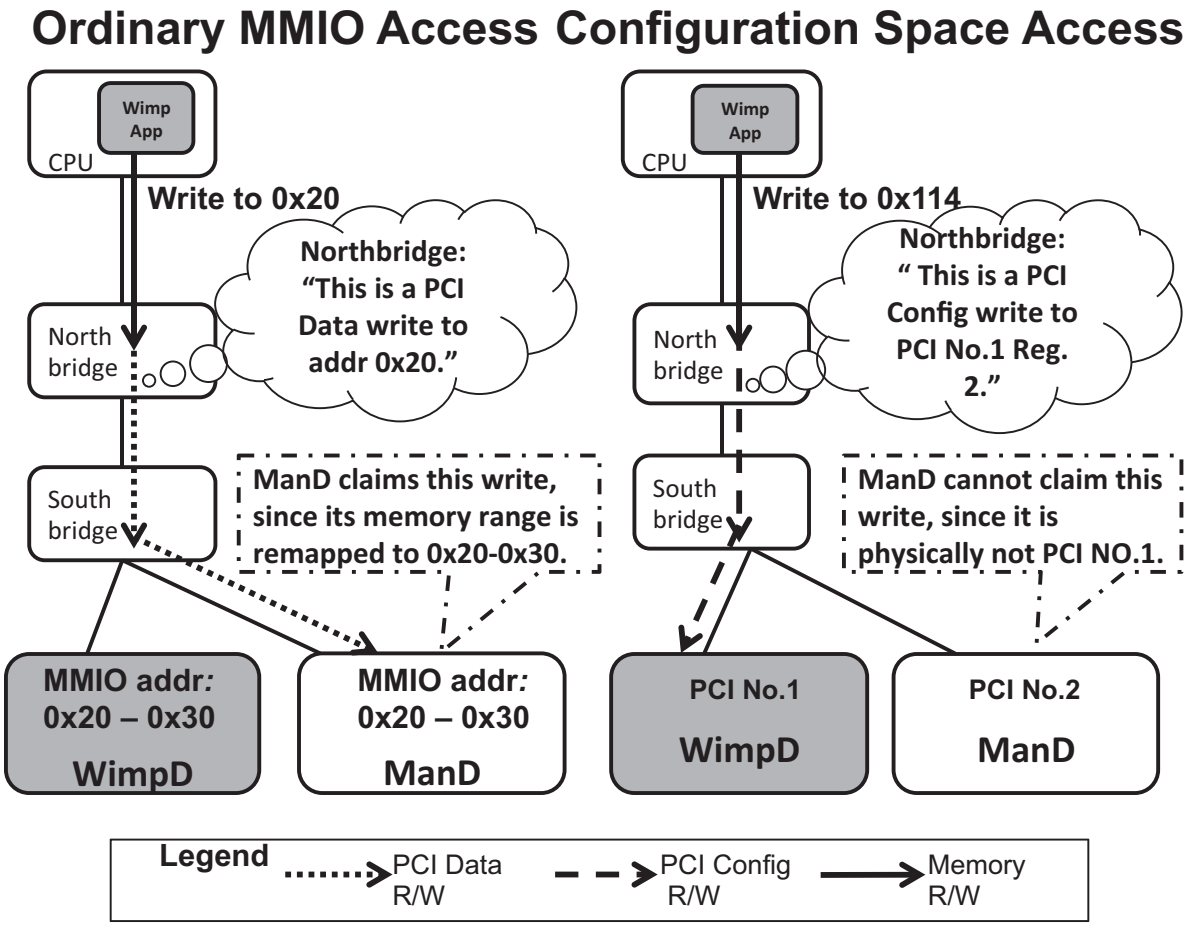


Figure 5.1: **MMIO Mapping Attack against I/O Memory Isolation.** In the left half of the figure, ManD's MMIO memory is remapped to overlap that of the wimp device (0x20-0x30), and the MMIO mapping attack will succeed. The right half shows that MMIO mapping attacks cannot compromise the access to PCI/PCIe configuration space.

I/O ports [19, 109] or through reserved MMIO memory regions [19]. At first glance, this appears to lead to a cyclic dependency: protecting the device configuration space, in reverse, relies on protecting the special I/O ports and MMIO memory regions.

However, this seemingly cyclic dependency can be resolved. The key observation is that I/O port conflicts and MMIO memory mapping attacks *cannot* corrupt the access to the device configuration space (shown in the right half of Figure 5.1). Even if some manipulated device has its I/O ports or MMIO memory regions overlapping those of the configuration space, the manipulated device still can *not* intercept any configuration space access destinated to other devices. Specifically, both the special I/O port numbers and the base address of the configuration

space MMIO memory are located in dedicated registers in the northbridge chipset [38]. The northbridge interposes on every port and memory access from the CPU(s). If the requested ports or memory regions fall into those of the configuration space, the northbridge transforms the access requests into PCI/PCIe configuration bus cycles with special address information. This address information is only correlated with the targeted device's static geographic position in the system hierarchy where the targeted PCI/PCIe device is hard-wired or plugged. I/O ports and MMIO memory remapping *cannot* manipulate device hierarchic positions, and thus *cannot* cause the manipulated devices to claim the configuration space cycles of other devices.

Therefore, during the execution of the secure application, the *mHV* only needs to configure the I/O port-access-interception bitmap (Section 5.1.1), Nested/Extended Page Tables, and IOMMU (Section 5.1.2), to prevent unauthorized CPU-to-memory access and DMA to the whole device configuration space. After that, the wimpy kernel can securely scanning through the I/O port and memory settings in the configuration space to detect possible I/O-port-mapping or MMIO mapping attacks. Protection of the device configuration space remains active until the wimp app is torn down.

5.1.4 Interrupt Isolation and Delivery

Our system handles three types of device interrupts, including hardware interrupts managed by the [Advanced] Programmable Interrupt Controller ([A]PIC), Message Signaled Interrupts (MSI), and Inter-Processor Interrupts (IPIs). The wimpy kernel should fulfill the following two requirements for device interrupt isolation: (1) Interrupts should be correctly routed, i.e., interrupts from the wimp device are exclusively delivered to the CPUs that run the respective wimp app, and other interrupts should not arrive at those CPUs. [1] (2) Spoofed interrupts should not compromise the I/O isolation.

A common pitfall in interrupt isolation is ignoring requirement (2). One may argue that (2) is unnecessary, because the driver of the trusted-path device endpoint can verify the identity of the interrupts it receives. When the wimp app receives an interrupt that appears to originate from the

[1] However, this mechanism does not apply to the interrupts which are shared by the untrusted OS and wimp apps and cannot be separated using Message Signaled Interrupts (MSIs). Similar to other related work [15, 40], we assume that this limitation will disappear once MSI becomes mainstream. Note that the PCI Express (PCIe) specification already mandates the MSI implementation on all PCIe devices.

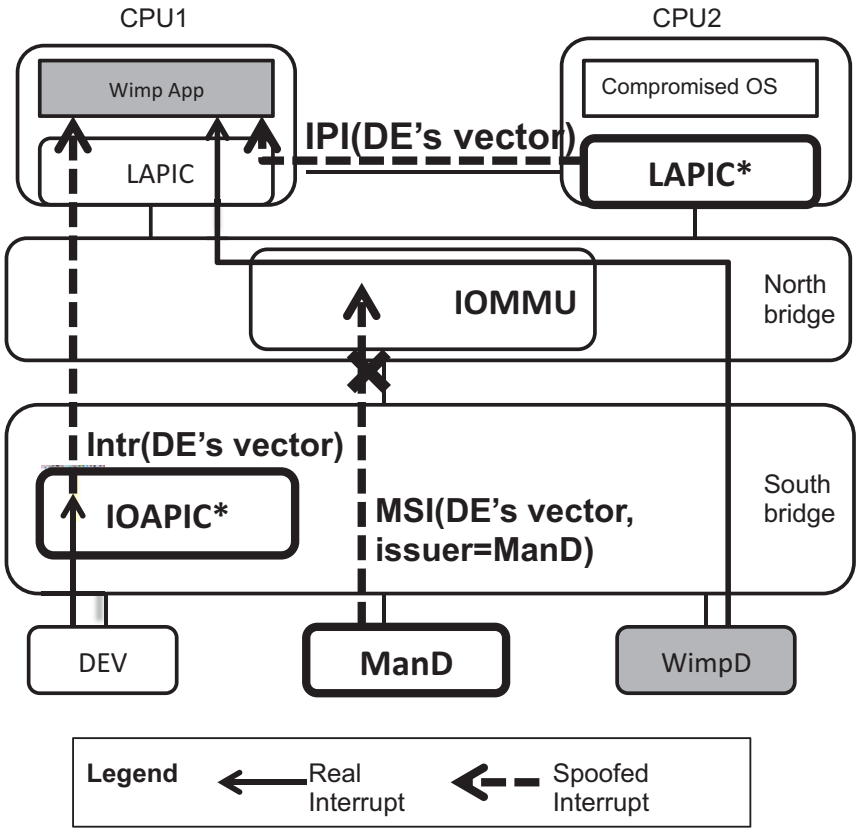


Figure 5.2: **Interrupt Spoofing Attacks.** `IOAPIC*/LAPIC*` denotes the interrupt controllers manipulated by the compromised OS. `Intr(DE's vector)` represents spoofed hardware interrupts with the DE's interrupt vector. When the IOMMU interrupt remapping feature is enabled, spoofed MSIs with incorrect issuer identifiers will be filtered out by the IOMMU (Section 5.1.4.2).

wimp device, it communicates with the wimp device to check whether the wimp device indeed has a pending interrupt. If not, the wimp app refuses to service this interrupt. However, *not* all wimp device device drivers are robust against spoofed interrupts. For example, MSI device drivers often assume that the OS avoids interrupt conflicts when initializing MSI-capable devices. As a result, a spoofed MSI may cause device driver misbehavior. MSI device drivers that receive a spoofed DMA Finish interrupt, without checking with the interrupting device, may operate on incomplete or inconsistent data.

To meet both interrupt isolation requirements, the wimpy kernel must modify the configurations of the interrupt controllers, MSI-capable devices, and other chipset hardware along the interrupt delivery route during the establishment of isolated I/O channels. The compromised OS

may subvert those configurations during the execution of the wimp apps, in order to misroute device interrupts or launch interrupt spoofing attacks (Figure 5.2). We will detail the protection mechanisms for hardware interrupts and message signaled interrupts in Sections 5.1.4.1 and 5.1.4.2, respectively. Interprocess-interrupts handling mechanisms are deferred to Section 5.4.

Note that for interrupt delivery to the wimp app, WK implements a signal notification mechanism similar to that of the Linux signals [17]. Using this mechanism, wimp apps register their interrupt handler with the WK, either at wimp registration, if the wimp device has fixed hardware interrupts, or during run-time, if the interrupts are software-initiated (e.g., MSIs). The WK intercepts the wimp device interrupts, and generates corresponding signals to the wimp apps.

5.1.4.1 Isolating Hardware Interrupts

Hardware interrupts are managed by a PIC on uni-processor systems, and by an IOAPIC and per-processor Local APICs (LAPIC) on multi-processor platforms. The PIC and IOAPIC are deployed with redirection tables that map device hardware interrupts to their corresponding interrupt vectors (with vector numbers and delivery destinations). The PIC or LAPICs then decide when and whether to deliver messages with those interrupt vectors to targeted CPU(s). During the establishment of isolated I/O channels, the wimpy kernel isolates hardware interrupts by the following steps:

- Modify the redirection table on PIC or IOAPIC to reroute wimp device interrupts and to remove any interrupt-to-vector mapping conflicts between the wimp device interrupts and others.
- Setup corresponding the PIC/LAPIC registers of the CPU that runs the wimp app to enable delivery of the wimp device's interrupts and to temporarily disable interrupts from other devices.
- Manipulate the Interrupt Descriptor Table (IDT) so that the wimp device interrupts will trigger the interrupt handler in the wimpy kernel.

While the wimp app is running, the *mHV* provides run-time I/O port and memory protections to the redirection tables, the interrupt controller registers, and the IDT (using the mechanisms

described in Sections 5.1.1- 5.1.3). Note that no run-time protection is needed on uni-processor systems, since the OS is held in a pending state during the execution of the wimp app.

5.1.4.2 Isolating Message Signaled Interrupts

An MSI-capable device can generate MSIs by writing a small amount of data to a special physical memory address. A chipset component interprets the special memory write and delivers the corresponding interrupts to the targeted processor(s) [19, 109].

Challenges. MSI-capable devices use Message Address Registers to store the memory address range, and Message Data Registers to store the data that defines the interrupt vectors. To launch an MSI-spoofing attack against a wimp app-wimp device trusted-path, a compromised OS can change the Message Data Registers of other devices to include the wimp device's interrupt vector. By programming the device's DMA scatter-gather unit, the OS can also spoof arbitrary MSI messages, without modifying any Message Address/Data Register on any device [132].

The software-configurable nature of MSIs and the complexity of the potential spoofing attacks make MSI isolation extremely difficult. Enumerating every MSI-capable device in the system and configuring their MSI control registers is not only time-consuming and inefficient, but also fails to defend against the above "scatter-gather attack" [132].

Solution. We design a comprehensive and efficient solution for isolating MSIs, which does not require controlling any MSI-capable devices other than the wimp device. Our solution leverages the Interrupt Remapping features in the IOMMU [7, 61]. For example, with Intel VT-D Interrupt Remapping, MSI messages are embedded with a specified handle [61]. Upon receiving an MSI message, the IOMMU uses that handle as an index to locate a corresponding Interrupt Remapping Table entry, which stores a device-specific interrupt vector.

To re-route MSIs from the wimp device, the wimpy kernel modifies the wimp device's MSI message handle to point to a specific interrupt vector with a chosen vector number and delivery destination (only the CPU(s) executing the wimp app). The wimpy kernel also configures the LAPIC registers and IDT entries to ensure that MSIs are delivered to, and serviced by, the correct interrupt handlers. Note that the chipset hardware that interprets MSI messages (e.g., the PCI host controller on the southbridge) often sits between the devices and the IOMMU on the northbridge. The compromised OS and manipulated devices may modify the configuration of this hardware to

Table 5.1: **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

suppress or mistransform MSI signals. Thus, the wimpy kernel must also configure and protect the corresponding registers on that interpreting hardware to enable and correctly transform MSI messages.

To defend against the MSI spoofing attacks, the wimpy kernel configures the corresponding interrupt remapping table entry of IOMMU to only accept MSI messages with the wimp device's device identifier. As shown in Figure 5.2, spoofed interrupts generated by manipulated devices do not have the interrupt identifier of the wimp device, and thus are filtered out by the IOMMU. Note that this defense mechanism relies on the assumption that the IOMMU can correctly identify the MSI issuer (similar to identifying the sender of DMA requests). We discuss this assumption in more detail in Section 9.1.

Throughout the execution of wimp apps, the wimpy kernel protects all the above registers and tables using the mechanisms described in Sections 5.1.1 - 5.1.3.

5.2 Decomposing Bus Subsystem

In this section, we chose the USB subsystem to illustrate the wimpy kernel design of bus subsystem decomposing and code minimization method for two 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 [66]. Second, 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., 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 de-

vice addresses. These channel control components can be directly accessed and protected by the wimpy kernel. Specifically, in PCI bus, the devices can be directly accessed via low-level I/O resources (e.g., the I/O ports and MMIO memory), we can use the mechanisms described in Section 5.1 to isolate the I/O resources of the target devices. The PCI devices have their separated control channels, the PCI configuration space, which can be decided by the physical positions of the devices in the bus (e.g., where the devices are hardware-wired or which slots the devices are plugged in) and the configurations of PCI bridges. Once the wimpy kernel identifies, verifies and protects the control channels, the untrusted OS cannot compromise them again. The wimpy kernel can further verifies the data channel configurations via the control channels to guarantee the I/O channel isolation. For example, the wimpy kernel verifies the MMIO settings in the PCI configuration space of all devices to avoid MMIO mapping attacks on the protected wimp device (Section 5.1.2). The wimpy kernel also verifies the MMIO memory range settings on the on-path PCI bridges and make sure that MMIO access to the wimp device is correctly routed.

5.2.1 Example: Linux 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 5.1.

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 5.2.2). 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 5.2.3).

5.2.2 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 6.2.2.1.

5.2.2.1 Attacks

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 5.3. The ultimate purpose of this type of device misconfiguration 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 and integrity of a wimp device with the same address.

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.,

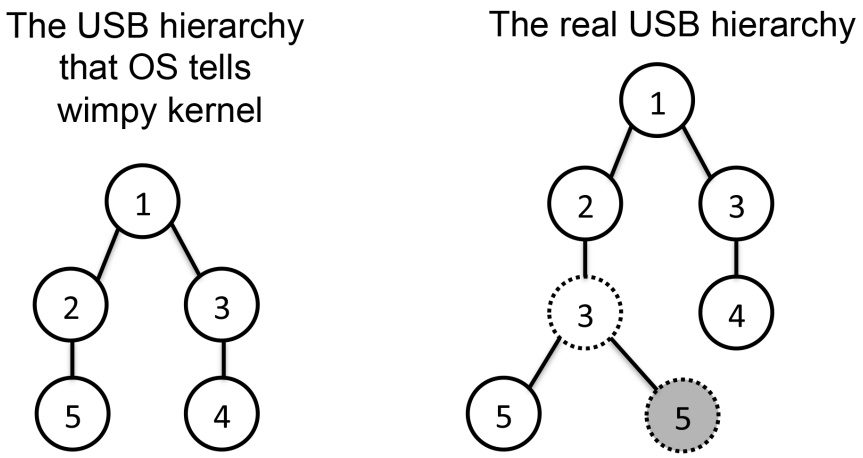


Figure 5.3: **USB Address Overlap and Remote Wake-up Attacks.** *Legend:* The root of the USB bus denotes the USB host controller, the leaves the USB devices, and the intermediate nodes the USB hubs. The number of each tree node denotes the USB device address. The dotted nodes represent the USB devices 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).

the dotted node No.3 in Figure 5.3), 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.

Proof-of-concept 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.

5.2.2.2 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 3.2.2). 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.

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 downstream 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.

Extensions to Support Multiple Wimps. The same USB hierarchy may be shared by multiple wimp applications. The above algorithm is used for the first wimp application. For the subsequent applications, we add the following two preliminary steps before running the algorithm.

(1) WK notifies the previously registered wimp apps and suspends their USB paths.

(2) WK activates the USB paths of the requesting wimp app.

Step (1) is necessary, because the USB paths activated in (2) may have hidden devices that conflict with the devices in the USB paths of the previous wimp apps.

5.2.2.3 Algorithm Analysis

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

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.

5.2.2.4 Discussion

The two main advantages of the USB hierarchy verification algorithm are as follows: (1) it only uses a few standard operations of the USB host controller and hubs; (2) it does not use the driver

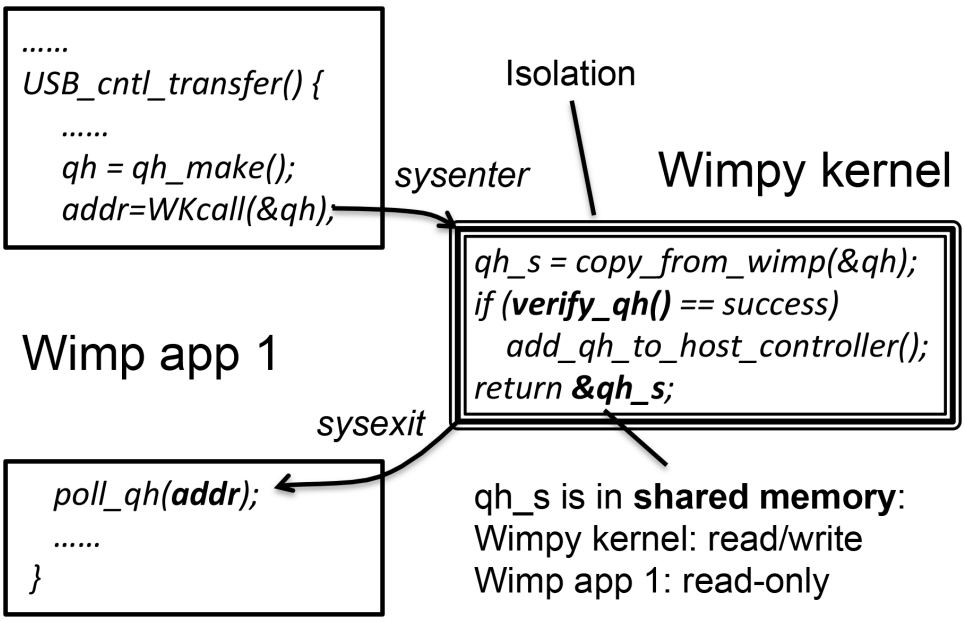


Figure 5.4: **USB Transfer Descriptor Verification by the Wimpy Kernel.**

of any other device that shares the same USB bus with the wimp device. Note that some USB host controller and hubs may have device-specific operations that can violate the I/O channel isolation. For example, some host controllers or hubs may be configured to record a few of their latest data transfers for debugging purpose. This feature may be abused by the untrusted OS to reveal some secret data of a isolated I/O channel. The algorithm should verify the configurations of these device-specific operations. A future work is to develop an automatic device specification checker to scan through the open specifications of all host controllers and hubs and to identify the sensitive device-specific operations. For devices that have no open specifications, there is no guarantee that we can use some black-box fuzz testing technique to identify the sensitive operations. Thus the verification algorithm should warn the users of the isolated I/O channels about the potential risks. Users that have higher security concerns can choose to avoid these devices on their platforms. This is one example of how users adapt the I/O isolation system for different usage models that could have various levels of security requirements.

5.2.3 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 5.4, 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 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 6.2.2.2 presents the details of USB transfer descriptor verification.

5.3 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 [66]. The wimpy kernel must provide the necessary driver support code to the wimp apps with small and simple code base.

To analyze the driver dependencies, this thesis 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 [66]) 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

Table 5.2: **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**

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 [29, 47, 48], whereby they use either authenticated-encryption or MAC modes[2] 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.

5.3.1 Exporting Decisions

Table 5.2 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. We present the results for each category of driver dependencies as follows:

(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]An isolated subsystem for key distribution is presented by Zhou *et al.* [138].

(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[3], wimp apps manage the signals between their processes, using user-space signal implementation.

(3) Kernel library for utilities, timers, debugging and book-keeping 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 run-time 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.

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

Table 5.3: **Driver Support Code Minimization Results**. "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

5.3.2 Methodology and Results

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 [66], and the CodeSurfer software analyzer [11][4].

We manually study those driver dependencies, and present the result of deprivileging the relevant OS support code in Table 5.3, following the interface categories defined in [66]. According to the analysis presented in Section 5.3, 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

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

WKcall interface for this instruction.

5.4 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. 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 [14, 49, 98, 104, 118], recent micro-hypervisors [69, 112, 119], and traditional hypervisor

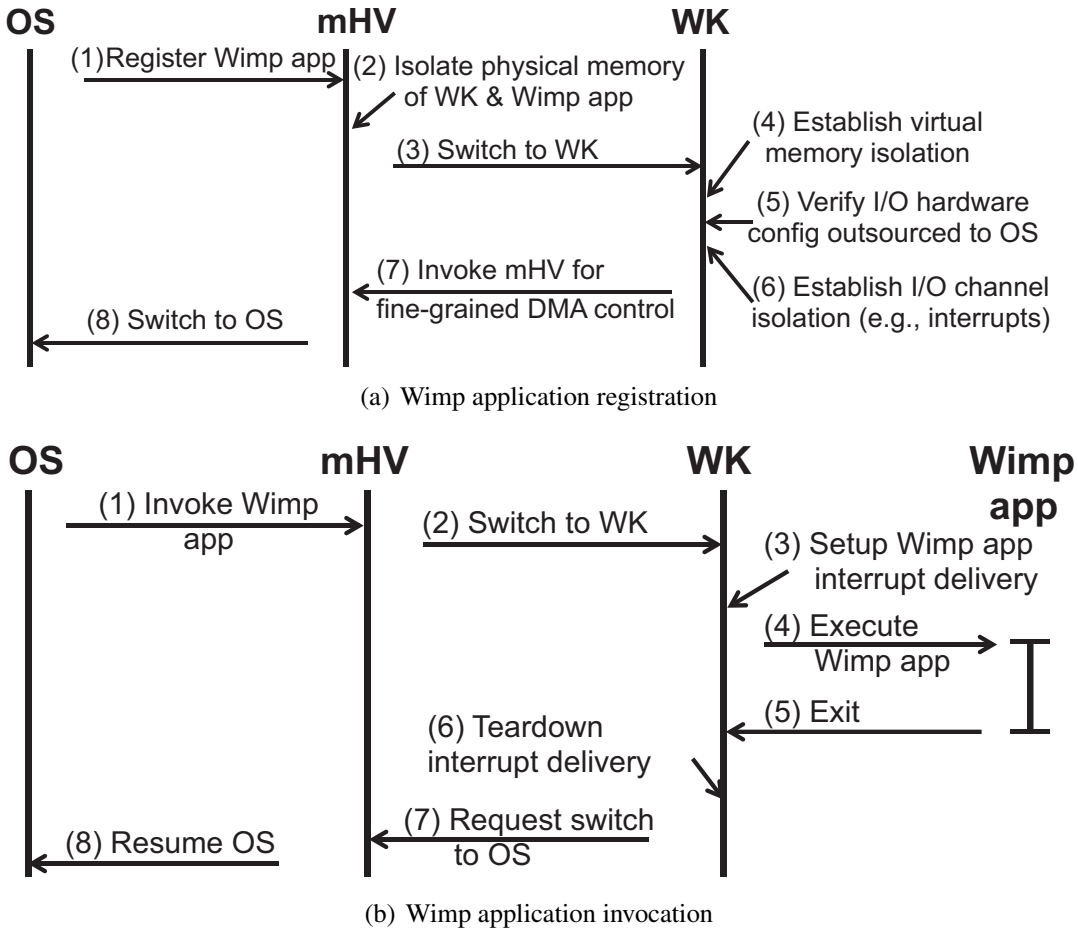
designs [21, 22, 24, 54]. We demonstrated its efficiency in Section 6.2.3.2. Specifically, the wimp app provides an OS service number, inputs, and a completion call-back function to WK. The WK signals the OS running on other CPUs using Interprocessor Interrupts (IPIs) [8, 62], which is a standard facility 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.

5.5 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 5.5.

Registration. The untrusted OS provisions the code, data and I/O memory required by the wimp app, configures the isolated I/O channels to the wimp devices, and explicitly registers the wimp app through an OS-hypervisor interface. During registration, the *mHV* isolates the wimp app's memory, 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 isolated I/O channels to wimp devices, and connects the wimp devices to the wimp app (but not delivers the wimp device interrupts). Until unregistration, the untrusted OS can no longer tamper with the memory regions and I/O resources of the registered wimp apps.

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



(a) Wimp application registration

(b) Wimp application invocation

Figure 5.5: **The Life-cycle of Wimp Applications.**

execution, the wimp app suspends its devices and transfers control to the 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. 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. Note that the wimp app can outsource the initialization of the wimp devices to the untrusted OS, if the wimp app is capable of verifying the OS's configurations. Otherwise, the wimp app should reset the wimp devices and reinitialize them to known good states, in order to defend against misconfigurations of the untrusted OS. The device resetting techniques depend on the implementation of the wimp devices and their device drivers. For example, for some USB devices, it is sufficient to simply deconfigure the devices and switch them back to the Address state [4]. For some PCI devices, the driver would set the device to ACPI D3 power state and then switch back to ACPI D0 power on state.

Unregistration. The OS explicitly requests wimp app unregistration via the wimp-OS channel, which is faster than via an OS-hypervisor interface. The wimp app finishes its execution and puts the wimp devices to a clean state. The WK tears down the isolated I/O resources of the wimp app with the help of the *mHV* and returns the CPU, memory regions and I/O resources of the wimp app to the OS. The OS may resume the original states of the wimp devices based on the wimp device state returned by the wimp app, if the OS can verify that the wimp device states are recoverable and will not compromise the OS (e.g., via DMA misconfiguration). The OS may also directly reset the wimp devices and resume the correct wimp device states. The detailed resuming mechanisms depend on the implementation of the wimp devices and the OS plug-in that works with the wimp app.

5.6 User Verifiablitiy

Our design enables verification of the system state (e.g., correct configuration and activation) to a third party who is often a human user. We use two simple devices for this task: a TPM that is widely accessible in many commodity computers, and a simple hand-held verification device.

Our hand-held device is simpler and more widely applicable than the special I/O devices

in some related works (Section 8). First, the standard remote attestation protocol is identical for different configurations, and thus a general-purpose verifier suffices to work for all secure applications. Second, our device only performs standard public-key cryptographic operations (e.g., certificate and digital signature verification) and a few cryptographic hash operations. It does *not* store any secrets. Third, our device outputs the verification result to the user via only one red-green, dual-color LED. The green light indicates the correct secure application-device connection state [**?**]. Moreover, our design can also support multiple I/O channels to different secure applications on a platform using just one simple device.

Note that all user-verification of the isolated I/O channel state in the presence of malware *requires some external trusted device*. Otherwise a user cannot possibly obtain *malware-independent verification* that the output displayed on the video display originates from a correctly configured and isolated trusted component, rather than from malware.[5]

Channel Status Verification Protocol. We describe a simple protocol for *user verification* of the I/O channel state. The hand-held verifier starts remote attestation by sending a pseudo-random nonce (for freshness) to an untrusted application on the host platform. Upon receiving the nonce from the untrusted application via some pre-reserved shared memory region, the isolated security-sensitive application requests a TPM Quote containing cryptographic hashes of the code and static initialized data of the wimp app, the wimpy kernel, and the *mHV* and a digital signature of the hashes generated by using a TPM-based key. The wimp app returns the signed quote to the untrusted OS and the untrusted OS sends the quote to the hand-held verifier. The verifier checks the validity of the signature and cryptographic hashes and displays the result to the user via a red-green dual-color LED. If the green LED is on, the user knows that the intended *mHV*, WK and wimp app are running on the host platform and the isolated I/O channels have been established. If the red light comes on, the security properties of the isolated I/O channels are not guaranteed.

[5] To obtain malware-independent verification of the I/O channel state we must detect the effect of the *Cuckoo attack* [90], which exploits the difficulty of a human in possession of a physical computer to guarantee that s/he is communicating with the true hardware TPM inside that computer. This is a generic attack for all attestation schemes that use TPMs, and we address it by requiring that (1) the public key (certificate) of the TPM be loaded in the verifier device before that verifier is used for the first time, and (2) the verifier checks the validity of the signatures originating from the local TPM.

Supporting Multiple Application-device I/O Channels. If activation of multiple isolated I/O channels to different program endpoints is desired from the same hand-held device, we envision that a single isolated I/O channel to a trusted shell [52] can first be executed on the target platform. This trusted shell, together with the wimpy kernel and the underlying micro-hypervisor, can be verified by the user as explained above. All other isolated application can then be registered using some trusted shell commands. A user can also identify, select, invoke, manage, monitor, and tear down any desired secure application via the trusted shell. Because the isolated I/O channels for the trusted shell has already been verified by the user using a hand-held device, the user input and the screen output by the shell can be trusted, so there is no need to use the external device to verify any subsequent isolated I/O channels.

5.7 Application Development

Our system design calls for the implementation of the device drivers of the wimp devices within the wimp applications for three assurance reasons. *First*, our goal is to produce a small and simple TCB, which can be verified with *a significant level* of assurance; i.e., assurance based on formal verification techniques. Including *all* device drivers would enlarge the hypervisor or the wimpy kernel beyond the point where significant assurance could be obtained. *Second*, placing the wimp device's driver within a wimp app is a natural choice: wimp device driver isolation can leverage all the mechanisms that protect the wimp app code and data from external attacks. *Third*, wimp devices are dedicated devices for a specific wimp application during its execution. Consequently, the device drivers are typically simpler than their full-fledged, shared-device versions. That is, we have the freedom to customize the wimp device driver for the specific needs of wimp applications.

The alternative of placing a wimp device device driver in a separately isolated domain in user or OS space would have two maintainability advantages over our choice. First, it would allow the driver to be updated or even replaced with a different copy without having to modify application code. Second, it would remove the need to maintain two versions of a device driver (one within the commodity OS and the other within the wimp app).

However, this alternative would have at least two security disadvantages. *First*, an addi-

tional protected channel would become necessary between the isolated wimp device driver and separately-isolated wimp app, and an additional protection boundary would have to be crossed and checked—not just the one between the hypervisor wimpy kernel and wimp app. *Second*, driver isolation in separate user or system space would require extra mechanisms in addition to those for wimp app isolation. For example, an additional protection mechanism would become necessary to control the access of application wimp apps to isolated drivers in user space. Furthermore, serious re-engineering of a commodity OS/hypervisor would become necessary [27, 71], which would run against our stated goals.

In balance, we pick the "driver-in-wimp-app" model since security and ease of commodity platform integration have been our overriding concerns in developing wimp apps.

Chapter 6

Application: Trusted Path System

6.1 A Simple Hypervisor-based Trusted Path

We implement a user-oriented trusted-path using a simple hypervisor-based system (directly implement I/O functions in the underlying micro-hypervisor *mHV*) and evaluate its performance. This trusted-path system protects a user's keyboard input sent to an isolated software module, and the output from the software module to the computer's display. The system defends against attacks launched by the compromised OS, other applications, and manipulated devices.

We implement the trusted-path system and perform all measurements on an off-the-shelf desktop machine with an AMD Phenom II X3 B75 tri-core CPU running at 3 GHz, an AMD 785G northbridge chip, and an AMD SB710 southbridge chipset. The machine is equipped with a PS/2 keyboard interface, an STMicro v1.2 TPM, and an integrated ATI Radeon HD 4200 with VGA compatible graphics controller 9710. This machine runs 32-bit Ubuntu 10.04 as its Desktop OS.

6.1.1 Hypervisor Implementation

We implement our hypervisor by extending a earlier version of TrustVisor [83]. Our extension includes configuration access protection, device I/O ports, MMIO and DMA memory protection, and interrupt redirection and isolation. Our wimp devices are a PS/2 keyboard and a VGA-capable integrated graphics controller. Specifically, the *mHV* protects the device configuration space (Section 5.1.3), and then securely enumerates all devices. The *mHV* also sets up the IOAPIC and LAPIC to deliver the keyboard interrupt to the CPU that runs the wimp app (Sec-

Table 6.1: **A Comparison of Hypervisor Code Bases**.

	Debug Code	C/Assembly Code	Header Files
Our Hypervisor	513	12556	2918
TrustVisor	468	11704	2566

tion 5.1.4), and protects the IOAPIC and LAPIC configuration using the mechanisms described in Sections 5.1.1 and 5.1.2. In addition, the *mHV* also downgrades the graphics controller to basic VGA text mode, identifies the corresponding VGA display memory region, and protects both this memory region and the entire graphics controller MMIO region by configuring the IOMMU and Nested Page Tables. Note that we have not implemented the MSI interrupt protection mechanisms and the LAPIC x2APIC mode virtualization (Section 5.1.4).

Small TCB. We use the sloccount[1] program to count the number of lines of source code in TrustVisor and our hypervisor. As shown in Table 6.1, our hypervisor implementation adds only 1,200 lines of code to TrustVisor's codebase, among which around 200 lines of code are for controlling the device configuration space (Section 5.1.3), 450 lines are for the interrupt protection mechanisms in Section 5.1.4, and 300 lines are for the I/O port and memory protection mechanisms in Sections 5.1.1 and 5.1.2. Our software TCB for the hypervisor (not including the source code for debug purposes) is about 15,500 lines of code in total.

6.1.2 Application Implementation

The wimp app comprises a PS/2 keyboard driver, which handles the keyboard interrupt, receives and parses keystroke data, and a VGA driver, which writes keystroke data to the VGA display memory. The wimp app runs in CPU Ring 3. This unprivileged setting allows for more efficient isolation mechanisms between the wimp app and the rest of the system. That is, instead of trapping every port access from the wimp app (Section 5.1.1), the *mHV* simply configures the OS's I/O permission bitmap in the Task State Segment to confine the wimp app's access to only the wimp device's I/O ports.

However, running the wimp app in Ring 3 makes wimp device driver porting more difficult. First, some sensitive I/O instructions (e.g., IN, OUT) and some critical device-driver instruc-

[1]Developed by David A. Wheeler, http://www.dwheeler.com/sloccount

tions (e.g., IRET) can only execute with system privileges (CPU Ring 0). Second, device drivers sometimes need to perform operations directly on physical memory addresses (e.g., to manipulate device registers), but drivers running unprivileged within a wimp app do not have access to the mappings between virtual addresses and physical addresses. Third, some of the physical memory pages are protected so that only privileged system code (running at CPU Ring 0) can access them.

During program endpoint implementation, we minimize invocations of the *mHV* for the above operations, while still maintaining the isolation of the wimp app from the OS. For example, the graphics card in VGA mode provides an MMIO memory region where software writes the contents that are displayed on the screen. To perform memory writes to this physical memory region, the wimp app reserves a region in its virtual memory space and then makes a hypercall to the *mHV*. The *mHV* re-maps the reserved wimp app memory pages to the VGA display memory region. After this hypercall, the wimp app has direct access to that memory region without any additional hypercalls.

Note that the hypervisor still needs to emulate some privileged instructions, e.g., when the wimp device interrupt handler finishes execution and returns control back to the wimp app, the interrupt handler should run a return-from-interrupt (IRET) instruction. The *mHV* provides a hypercall that emulates this IRET instruction.

6.1.3 Micro-benchmarks

We present micro-benchmark results to demonstrate: (1) the overhead of trusted-path establishment and tear-down is reasonable, and (2) our optimized wimp app implementation can achieve good performance by minimizing invocations of the *mHV*.

Trusted-path Setup and Tear-down. To measure the *mHV* overhead for trusted-path establishment, we compared the time required for the creation of a wimp app's isolated environment and a trusted path between a wimp app and a wimp device with the time required to create only a wimp app's isolated environment using TrustVisor. As shown in Table 6.2, TrustVisor took about 1.752 milliseconds to create the isolated environment, while our *mHV* took about 1.925 milliseconds to create the same environment *and* establish the trusted-path. Thus, trusted-path establishment adds about 9.8% overhead to the original TrustVisor implementation.

Table 6.2: **Trusted-path Setup and Tear-down Overhead.** Average (in ms) of 10,000 trials.

	TrustVisor	HV
Trusted-path Setup	1.752±1.7%	1.925±2.2%
Trusted-path Tear-down	0.436±1.9%	0.528±1.8%

Table 6.3: **Wimp Device Driver Performance.** Average latency overhead (in μs) of 100,000 trials.

	Direct Access	Invoking HV
I/O Port Access (INB)	18±6.2%	40±3.4%
I/O Port Access (OUTB)	19±5.4%	40±3.7%
VGA Display Memory Write	15±3.2%	39±2.7%

We also measured the *mHV* overhead incurred in trusted-path tear-down after the wimp app completes all of its operations. In 10,000 trials, TrustVisor tore down the isolated environment of the wimp app in approximately 0.436 milliseconds, while *mHV* tore down the same isolated environment and the trusted path in approximately 0.528 milliseconds. Thus, the trusted-path tear-down adds approximately 21% overhead to the original TrustVisor implementation. This is because it takes much less time to tear-down an isolated environment than to create one, while setting up and protecting the APICs and graphics controller configuration during trusted-path establishment takes roughly the same time as restoring and unprotecting them during trusted-path tear-down.

In our experiments, both the latency overhead of trusted-path establishment and tear-down were negligible compared to the duration of an ordinary TP session, which often lasts for seconds or more.

Device Driver Performance. We measure the *mHV* overhead in emulating the INB and OUTB operations to a device (PS/2 keyboard in this case study) and data writes to MMIO memory (VGA display memory region in this case study), since these are common operations for most trusted-path applications. The measurements in Table 6.3 illustrate that our optimized implementation of user-level wimp device drivers can achieve good performance by minimizing the frequency of *mHV* invocations for operations that require system-level privileges. Our optimized wimp app implementation took only 18 microseconds to perform INB and 19 microseconds for OUTB. In contrast, invoking the *mHV* and performing a same operation would take around 40

microseconds. wimp app writes to VGA display memory take approximately 15 microseconds, but it would take more than 39 microseconds to invoke the *mHV* to perform the same operation. This implies that a context switch between the trusted-path program endpoint and the hypervisor takes roughly 23 microseconds.

6.2 A Wimpy-kernel-based USB Trusted Path

In this section, we present the implementation of a more complex trusted path application using the wimpy-kernel-based architecture. The trusted path application accepts input from a USB keyboard.

6.2.1 Micro-hypervisor Implementation

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 6.2.3.1.

6.2.2 Wimpy Kernel Implementation

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 6.2.3 show the minimality and efficiency of the WK.

6.2.2.1 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.

6.2.2.2 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.

6.2.2.3 Wimpy-Kernel Interfaces

We implement the WKcall interface using the standard x86 Fast System Call instruction [8, 62] (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*.

6.2.3 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

Table 6.4: **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

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.

6.2.3.1 Code Base Size Evaluation

We use sloccount to calculate the Source Lines of Code (SLoC) of the *mHV* and the WK. As shown in Table 6.4(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 [127]. The code base of our micro-hypervisor is smaller than other micro-hypervisors, and much smaller than full functioning VMMs/hypervisors[2].

Table 6.4(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 6.5 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

[2] Fides [114] has 7.2K SLoC, but without DMA protection, multi-core and AMD x86 virtualization support. The new version of TrustVisor based on XMHF [127] has about 24K SLoC without implementing fine-grained DMA protection. Guardian [23] has approximately 25K SLoC. NOVA's code base contains 36K SLoC [113]. BitVisor [112] 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 6.5: **A Comparison of USB Code Base of Wimpy Kernel and 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*

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.

6.2.3.2 Micro-benchmarks

USB Hierarchy Verification. Table 6.6 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.

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 6.7 illustrates the latency overhead of two main wimpy kernel interfaces; i.e., the WKcalls 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 WKcalls 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 6.8 presents the latency overhead of the registration,

Table 6.6: **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 6.7: **Latency Comparison of WK- and Hypervisor-involved Context Switches**.

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

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.

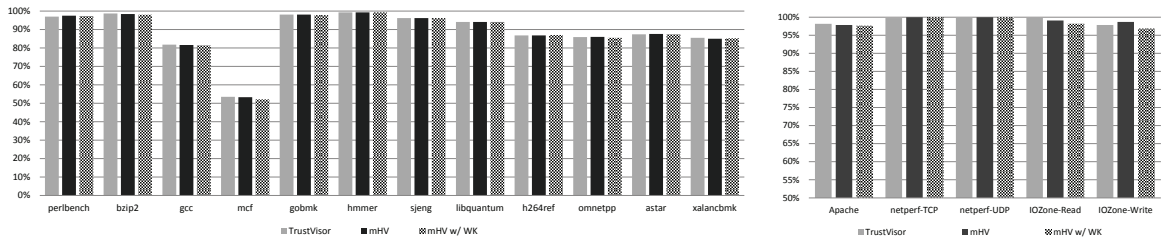
6.2.3.3 Macro-benchmarks

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 6.1), 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

Table 6.8: **Latency of Wimp-app Life-cycle Operations**.

	Registration	Invocation	Unregistration
Time (ms)	583.79	0.26	0.97



(a) CPU Benchmarks (SPECint 2006)

(b) I/O Benchmarks

Figure 6.1: **CPU and I/O Macro-benchmark Results**.

Figure 6.1), by registering the wimp app but not invoking it. The results shown in Figure 6.1 are all normalized to the benchmark results on the vanilla OS.

CPU Benchmarks. As shown in Figure 6.1(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 6.1(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.

Chapter 7

Application: Corporate Key Management System

Deploying a corporate key management system faces fundamental challenges, such as fine-grained key usage control and secure system administration. None of the current commercial systems (either based on software or hardware security modules) or research proposals adequately address both challenges with small and simple Trusted Computing Base (TCB). This chapter presents a new key management architecture, called KISS, to enable comprehensive, trustworthy, user-verifiable, and cost-effective key management. KISS protects the entire life cycle of cryptographic keys. In particular, KISS allows only authorized applications and/or users to use the keys. Using simple devices, administrators can remotely issue authenticated commands to KISS and verify system output. KISS leverages readily available commodity hardware and trusted computing primitives to design system bootstrap protocols and management mechanisms, which protects the system from malware attacks and insider attacks.

7.1 Motivation and System Overview

As consumers and corporations are increasingly concerned about security, deployments of cryptographic systems and protocols have grown from securing online banking and e-commerce to web email, search, social networking and sensitive data protection. However, the security guarantees diminish with inadequate key management practices, as exemplified by numerous real-world incidents. For example, in 2010 Stuxnet targeted Iranian uranium centrifuges, installing device drivers signed with private keys *stolen* from two high-tech companies [35]. In another incident, the private keys of DigiNotar, a Dutch certificate authority, were maliciously *misused*

to issue fraudulent certificates for Gmail and other services [126]. Even high-profile, security-savvy institutions fall prey to inadequate key security, let alone companies with a lower priority for security.

Despite its indisputable significance, *none* of the current corporate key management systems (KMS) – either industrial solutions based on software, or hardware security modules (HSM), or research proposals known to us – provide comprehensive key management *with small and simple trusted computing base (TCB)*. There are at least two significant challenges that lead to the insufficiency of the KMS, as shown in Table 7.1.

Fine-grained Key Usage Control. A comprehensive life-cycle KMS should enforce *fine-grained key usage control* (i.e., whether an application operated by a user has the permission to access a specific cryptographic key). This problem is exacerbated with the current trend of Bring Your Own Device (BYOD), which allows client devices (e.g., tablets and laptops) to increasingly host both personal and security-sensitive corporate applications and data.

Although commercial HSMs [55, 89, 97, 99, 120] provide high-profile physical protection of cryptographic keys and algorithms, they fail to control key usage requests from outside their physical protection boundary (e.g., the users and applications on other client computers). The attackers can cause key misusage [126] by compromising client computers and submitting fake key usage requests to the HSMs. Some HSMs enable porting key usage applications to an in-module secure execution environment [99, 120]. This method only provides application-level key usage control, and is not scalable due to the limited resources of the dedicated environment. Some HSMs enforce key usage control by accepting requests from client machines that deploy special hardware tokens only. This mechanism is insecure because it cannot block requests from a compromised operating system (OS) or an application on an authenticated machine.

Cost-sensitive companies commonly deploy *key management software* [57, 88, 115] on commodity servers, and rely heavily on the underlying OS services to protect cryptographic keys and management operations. These systems are untrustworthy because modern OSes are large and routinely compromised by malware.

Research proposals (e.g., credential protection systems [20, 41, 73] and hypervisor-based solutions [83, 84]) leverage Trusted Platform Modules (TPM) sealed storage. It assures that the

Table 7.1: **A Comparison between KISS and Current Key Management Systems.** "HSM", "SW", and "TPM" represent the KMS that are based on HSM, software packages, and TPM seal storage, respectively. "ROT" denotes the root of trust of the systems.

Systems	Key Usage Control	Administration Interfaces	TCB	ROT
HSM [18,20,7,16,17]	coarse-grained (application or machine control)	HSM & complex admin dev, non-verifiable	large	HSM, admin dev
SW [15,19,9]	insecure (rely on OS)	keyboard/display, non-verifiable	large	OS
TPM [5,11,2,13,14]	coarse-grained (only application control)	keyboard/display, non-verifiable	large	TPM
KISS	fine-grained (both application and user control)	trusted path & simple admin dev, verifiable	small	TPM, admin dev

cryptographic keys sealed by an application can only be accessed by the same software. However, this approach is coarse-grained; it does not enforce any user authentication of the sealed keys.

Secure System Administration. A trustworthy KMS should allow benign administrators to securely manage the system and defend against attacks from malicious insiders. It must guarantee the authenticity of the communication between the administrators and the KMS. Otherwise, an adversary can cause unintended key management operations by stealing administrator login credentials, modifying or spoofing the administrator command input or the KMS output (e.g., operation result, system status).

The HSMs usually mandate the administrators to perform management operations via the I/O devices (e.g., keyboard and display) that are physically attached to the modules. For remote administration, they need complicated management software running on a commodity OS or a dedicated administrator device. Both mechanisms significantly increase system TCB and thus exposes larger attack surface. For software-based KMS, the I/O interfaces and authentication-relevant devices are controlled directly by the underlying OS, which means that the administrator credentials, input commands, and KMS output can easily be compromised by malware in the OS. Similarly, research proposals [20, 41, 73] do not support trustworthy remote management mechanisms. More importantly, none of KMS solutions provide intuitive ways for administrators to verify the status of the administration interfaces. Without such verification, administrators cannot trust any displayed system output and may mistakenly perform operations.

Contributions. To address the above challenges, this chapter presents KISS (short for "Key it

Simple and Secure"), a comprehensive, trustworthy, user-verifiable, and cost-effective enterprise key management architecture. Table 7.1 compares KISS with mainstream KMS and research proposals. Among them, KISS is the first KMS that supports *fine-grained key usage control* based on users, applications, and configurable access-control policies. To do this, KISS isolates authorized corporate applications from the untrusted OS and measures the code identities (cryptographic hash) of the protected applications. KISS also directly accepts user authentication by isolating user-interface devices and authentication relevant devices from the OS. Moreover, KISS enables *secure system administration*, leveraging a simple external device with minimal software/hardware settings. The KISS administrators execute thin terminal software on commodity machines. The thin terminal accepts administrator input via trusted paths, remotely transfers the input to and receives system output from the KISS system. The administrators use the external devices to *verify* the execution of the thin terminal and trusted paths and guarantee the *authenticity* of the input/output.

KISS leverages hypervisor-based code isolation and wimpy-kernel-based I/O isolation to protect the key management software and cryptographic keys from the large untrusted OS, applications, and peripheral devices. The administrators securely bootstrap the KISS system using the simple administrator devices and lightweight protocols, regardless of malware attacks and insider attacks from malicious administrators. These mechanisms together significantly reduce and simplify the KISS TCB, enabling higher security assurance. Because KISS leverages commodity hardware and trusted computing techniques, it is *cost-effective* and makes the wide adoption of KISS in small- and medium-sized business possible, in addition to financial or governmental institutions. KISS showcases how trusted computing technologies achieve tangible benefits when used to design trustworthy KMS.

Chapter Organization. First, we describe the KISS attacker model in Sections 7.2. Section 7.3 describes in detail the KISS system model and administrative policies. In Section 7.4, we illustrate the unified architecture on different KISS components and the simplicity of the external administrator devices. Sections 7.5, 7.6, and 7.7 introduce the detailed mechanisms for system bootstrap, secure administration, and fine-grained key usage control, respectively. We analyze potential attacks on KISS and our defense mechanisms in Section 7.8. Section 7.9 discusses

KISS extensions with stronger security properties or address real-world application issues. We then compare our solution with related work (Section 7.10) and conclude the chapter.

7.2 Attacker Model

Aside from the adversary model described in Section 2.2, we also consider *insider attacks* from malicious administrators that attempt to leak, compromise, or misuse the cryptographic keys. They can actively issue unauthorized key management operations, intentionally misconfigure the KMS and corporate applications, or steal the administrator devices or credentials (e.g., password, smart cards) of benign administrators. However, benign administrators are trusted to protect their administrator devices and credentials and comply with the KISS protocols.

7.3 System Overview

Corporate key management in this chapter refers to the establishment and usage of cryptographic keys in corporate and distributed environments. In this section, we provide a high-level overview of KISS system entities and model, and demonstrate how this model enables scalable and hierarchical enterprise key management.

7.3.1 System Entities

Figure 7.1 shows the four major entities in the KISS system.

Key Management Server (KISS Server). A commodity server machine that executes the key management software to perform server-side key life-cycle operations (e.g., key generation, registration, backup, revocation, de-registration, and destruction).

Key Management Clients (KISS Clients). Distributed machines (e.g., employees' desktops or corporate web servers) that install the KISS client software to receive cryptographic keys from the KISS server and use the keys to provide services to corporate applications. For example, On employees' desktops, the cryptographic keys stored in the KISS client software can be used to encrypt confidential documents. For a corporate web server, the keys are used to authenticate the outgoing network traffic.

Remote Managers (KISS Managers). Commodity machines used by KISS administrators to perform remote management. These machines install the KISS manager software to securely

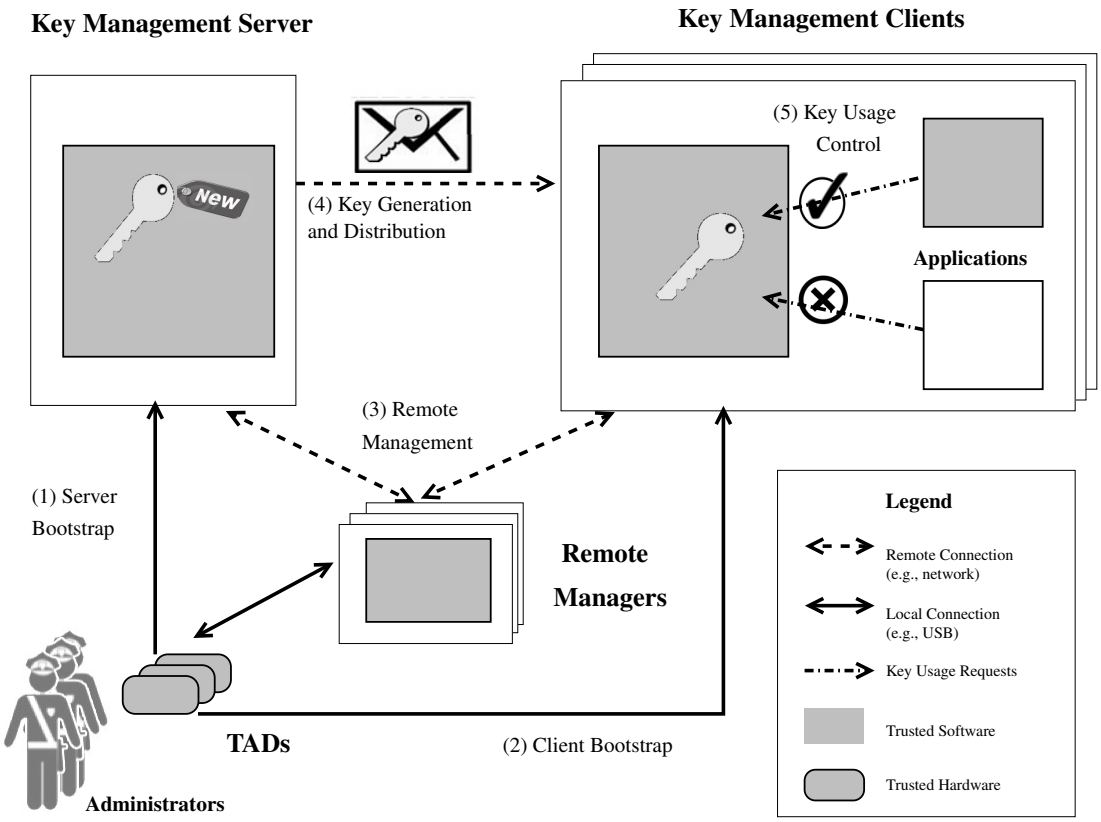


Figure 7.1: KISS System Model. Major operations include: (1) Administrators perform key management operations on KISS server and clients (e.g., key generation, backup, update, and revocation). (2) KISS server securely distributes cryptographic keys to clients. (3) On KISS client, cryptographic keys are isolated in key management regime, and are used to decrypt protected corporate data. (4) Users operate applications in isolated corporate regime of the clients to access corporate data.

transfer administrative commands to and receive system output from the KISS server or clients.

Trusted Administrator Devices (KISS TAD). Small, dedicated devices that are directly connected (e.g., via USB) to the KISS server or clients for local administration, or connected with the KISS managers for remote management.

7.3.2 System Model

Figure 7.1 also demonstrates a basic workflow of bootstrapping and using the KISS system. In Steps (1) and (2), administrators install and execute the KISS software on the server or clients, and perform bootstrap protocols to establish cryptographic channels between the server software, client software and TADs. We design our system to protect the server/client software and

the channel keys against malware attacks (see Section 7.4). The bootstrap protocols must be performed by a quorum of administrators to defend against malicious insider attacks. Each participating administrator use his/her TAD to confirm that the KISS bootstrap process succeeds. After bootstrap, the KISS server software starts recording subsequent system operations in a tamper-evident audit log, which helps the administrators detect insider attacks. Section 7.5 illustrates the KISS bootstrap protocols, cryptographic channel establishment, and audit log in detail.

In Step (3), the administrators remotely manage the KISS server/client software, leveraging their TADs and KISS managers. The KISS system protects the manager software (acting as a thin terminal) and user-interfaces devices (e.g., keyboard, and display) against malware attacks from the KISS manager OS. The administrators can securely input commands and review system output via the KISS manager user interfaces. The administrators use their TADs to authenticate the outgoing commands, and verify the authenticity of the operation results back from the KISS server/client software. Section 7.6 describes the remote management process and how our design significantly reduces KISS TCB.

In Step (4), new cryptographic keys (which are our key management products) are generated in the KISS server and securely distributed to the clients via the cryptographic channels established in step (2). In Step (5), the KISS client software protects the distributed keys, and handles key usage requests from various applications. KISS enables more fine-grained control of key usage than previous key management systems and proposals. It isolates the applications (similar to the isolation of KISS server software from the server OS) and measures their code identities. It also provides protected channels between authentication devices and the KISS client software, so that the KISS client software can directly authenticate the users of the applications. If the requests are from authorized users (e.g., company employees) and corporate applications (e.g., corporate document editors), the KISS client software uses the corresponding cryptographic keys to process the requests (e.g., decrypt confidential documents). The KISS client software rejects any key usage request from unauthorized users (e.g., visitors that are not allowed to read any confidential document) or applications (e.g., personal web browsers, media players). Section 7.7 describes the detailed mechanisms of our fine-grained key usage control.

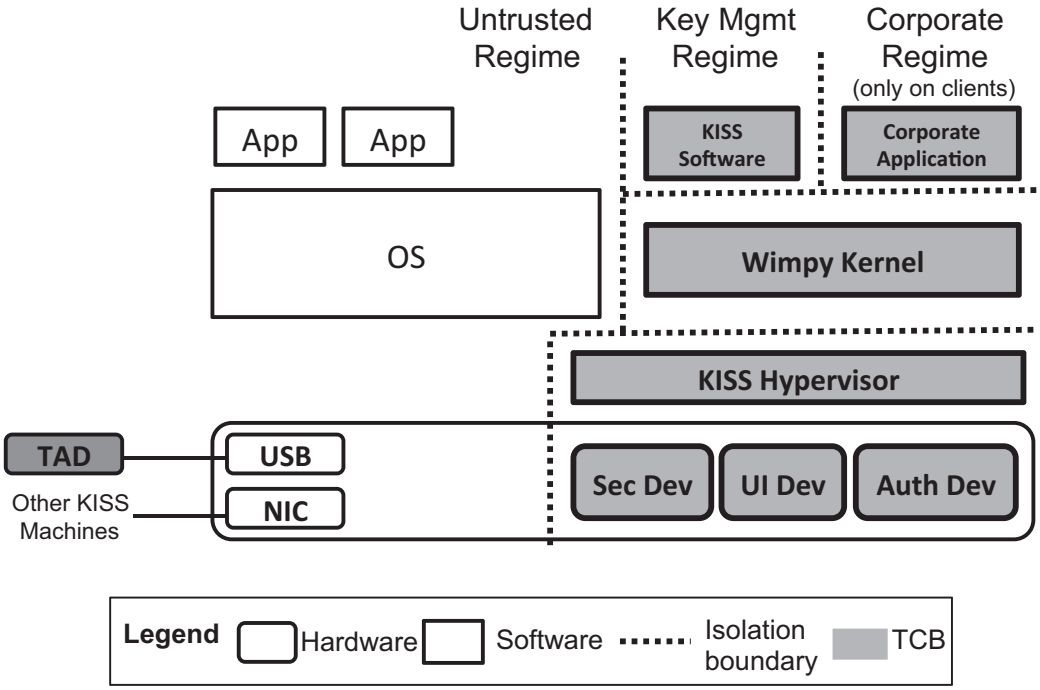


Figure 7.2: **System Architecture for KISS Client, Server, and Manager.** Sec Dev is the hardware (e.g., TPM) that provides trusted computing primitives. UI Dev denotes the user-interface devices, such as a keyboard and a display. Auth Dev is the device used for authentication (e.g., fingerprint scanner, and keypad). The KISS machines communicate via the network interface cards (NIC), and connects with TADs via USB interfaces.

The KISS client is necessary for collecting application and user information to perform key usage control. By receiving keys from the server, it also supports offline key usage, which reduces the key access latency and allows key usage when network connections are unavailable (e.g., while traveling on flights). However, offline key usage increases the risk of key abuse (e.g., when client machines are stolen). Companies might enforce special key usage policies to reduce this risk, such as requiring client machines to periodically obtain key usage permissions from the KISS server. Note that KISS can easily be modified to serve as the key usage control front end of the HSM. The KISS server software receives approved key usage requests from the clients, and securely transfers them to the HSM on the server machines via trusted paths. Both the cryptographic keys and algorithms are always protected inside HSM.

7.4 System Architecture

In this section, we introduce the unified architecture for the KISS server, client, and manager, and the hardware/software settings of TAD. We demonstrate how our architectural design significantly reduces and simplifies the TCB of the whole system, which is necessary for achieving high security assurance.

7.4.1 KISS Server, Client, and Manager

KISS server, client, and manager share the same architecture, hence we only illustrate the KISS client in detail here. As shown in Figure 7.2, the KISS hypervisor and wimpy kernel are dedicated to three main tasks:

Isolation. The KISS hypervisor divides the client to three isolated software regimes. The key management regime runs the KISS client software and stores all cryptographic keys during its run time. We also leverage TPM sealed storage to protect the cryptographic keys at rest. Each authorized application that uses the keys is isolated in its own corporate regime. The untrusted regime consists of the commodity OS, other applications, and devices.

Trusted Paths. When the administrators locally manage the client machine, the hypervisor and wimpy kernel establish trusted paths between the client software and the UI Dev or Auth Dev (Figure 7.2). The trusted paths protect the administrator command input and the client software output and safeguard the user authentication credentials. We defer the detailed explanation to subsequent sections.

Key Usage Control. The hypervisor and wimpy kernel helps the KISS client software to collect the identifier of the corporate applications and users that request key usage. When isolating the corporate applications in corporate regimes, the KISS hypervisor computes a cryptographic hash of the code and static data of the wimpy kernel and the corporate application, and transfers the hash value as application identifiers to the KISS client software. Trusted paths between the authentication-relevant devices and the KISS client software are established for user authentication. Section 7.7 describes the key usage control procedure.

7.4.2 Trusted Administrator Device

TAD is a small, dedicated, embedded device that assists system administration, both locally and remotely. TAD employs much simpler software/hardware than the typical administrator devices in current KMS. TAD does not need a full user-interface hardware for the key management command input and system output. Instead, the administrator can leverage the trusted paths provided by the KISS hypervisor and wimpy kernel on the server, client or manager. TAD does not implement complicated key management software to interpret operation input/output. These are directly handled by the KISS server/client software. During remote management, the KISS manager software only collects and transfers administrator input to server/client, and receives returning operation results.

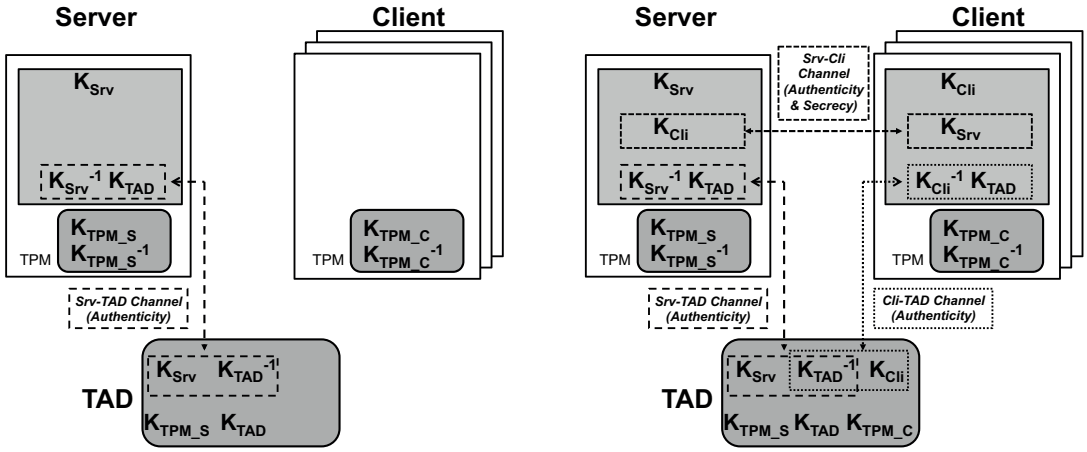
TAD implements software for the KISS bootstrap protocol, standard cryptographic primitives, remote attestation protocol, and necessary hardware drivers (note that the USB driver code is included, but not in the TCB). The TAD software is responsible for three tasks: (1) performing server/client bootstrap; (2) remotely attesting to the KISS server, client, and manager software; and (3) authenticating the administrator input and verifying the authenticity of the server/client output. To meet these functional requirements, TAD includes only a low-end CPU, small on-chip RAM and flash storage, a USB controller, a few buttons, a minimal display to show hexadecimal values, and a physical out-of-band channel receiver (e.g., QR code scanner).

7.5 System Bootstrap

In this section, we introduce the lightweight KISS bootstrap protocols. These protocols allow a quorum of administrators to verify that the "known good" KISS software is executing on the server/clients, and to establish cryptographic channels between their TADs, the server software and the client software. These channels (depicted in Figure 7.3) are used in secure system administration and key life-cycle operations. The bootstrap protocols are resilient against malware and insider attacks.

7.5.1 Server Bootstrap

During the KISS server bootstrap, a quorum of administrators execute authentic KISS server software and establish the Srv-TAD cryptographic channel (Figure 7.3(a)). Our lightweight server



(a) Server Bootstrap (b) Client Bootstrap and Registration

Figure 7.3: **Cryptographic Channels Established during KISS Bootstrap.** Before the bootstrap, the server and clients only have their TPM keys, and TADs has no pre-injected keys.

bootstrap protocol needs minimal administrator involvement. It does not require pre-sharing secrets in TAD (e.g., vendor-injected device private keys). After the bootstrap, the server software starts recording subsequent system operations in a tamper-evident audit log, which help the administrators detect insider attacks.

Bootstrap Protocol. Figure 7.4 illustrates the server bootstrap protocol. Before the protocol begins, we assume that the administrators creates the necessary configuration file, C_i, of the KISS server software independently and store them in TADs. The C_i includes the number of participating administrators, N, a quorum threshold, t, and other necessary server parameters. In Step 1, each administrator gathers the information of the hardware root of trust, i.e., the TPM public key K_{TPM_S} of the server, via a trusted out-of-band (OOB) channel. We suggest a secure and practical OOB channel, in which K_{TPM_S} is encodes as a tamper-evident physical label, e.g., an etched QR code on TPM chip surface. Each TAD_i securely attains K_{TPM_S} by scanning the QR code.

After that, each TAD_i generates a device key pair, $\{K_{TAD_i}, K^{-1}_{TAD_i}\}$, and sends C_i along with the public key, K_{TAD_i}, to the server (Steps 2 and 3). In Steps 4–6, the server executes the KISS hypervisor, wimpy kernel, and server software via late launch primitives [8, 60] Late launch resets a special Platform Configuration Register (PCR) of the TPM, and stores the cryptographic

1. TPM $\xrightarrow{OOB} TAD_i$: K_{TPM_S}
2. TAD_i : Generates device key pair $\{K_{TAD_i}, K^{-1}_{TAD_i}\}$
3. $TAD_i \rightarrow$ Server : $\{\mathsf{C}_i, K_{TAD_i}\}$, where C_i lists the configurations of the Server, e.g., # of involved administrators N, and quorum threshold t.
4. Server : Gathers N messages from TAD_i before timeout, late launches HYP and Server (their measurement is stored in TPM).
5. Server : Checks that all C_i are consistent, and $N \geq t$, generates Server key pair $\{K_{Srv}, K^{-1}_{Srv}\}$
6. Server $\rightarrow$ TPM : Stores the measurement of $\{K_{Srv}, C_i, \Lambda = \{K_1, \cdots, K_N\}\}$
7. $TAD_i \rightarrow$ TPM : Nonce R_i
8. TPM $\rightarrow$ Server : Signature $S_i = \{\mathsf{R}_i, \mathsf{M}\}_{K^{-1}_{TPM\_S}}$, where M is the measurement of $\{HYP, Server, K_{Srv}, \mathsf{C}_i, \Lambda\}$.
9. Server $\rightarrow TAD_i$: ID_i, S_i, Λ, K_{Srv}, where ID_i is a unique identifier for TAD_i
10. TAD_i : Verifies S_i and M, checks $K_{TAD_i} \in \Lambda$, $\#(\Lambda) = N$, and stores K_{Srv}

Figure 7.4: **KISS Server Bootstrap Protocol.** Each administrator possesses a TAD_i.

measurement of the HYP and the server software in this register for further remote attestation. After that, the server software generates a key pair, $\{K_{Srv}, K^{-1}_{Srv}\}$, and a key list, Λ, by receiving the public keys, K_{TAD_i}, from all participating TADs. The server software stores the measurement of K_{Srv}, C_i, and Λ into other PCRs of the TPM. The accumulated measurement, together with its signature generated by TPM attestation keys (linked with the TPM private key, $K^{-1}_{TPM\_S}$), are sent to the verifier during remote attestation (Step2 7– 9).

Upon receiving the attestation response, TAD verifies the signature using K_{TPM_S}, and trusts the authenticity of the accumulated measurement, M (Step 10). TAD re-computes M using its pre-installed knowledge (e.g., cryptographic hash of HYP and server software, configuration file C_i), the received K_{Srv} and Λ. If the verification succeeds, TAD trusts that the authentic hypervisor, wimpykernel and server instance are executing on the KISS server with the appropriate configurations, and that the server instance has the server private key and a correct list of TAD public keys. TAD also verifies that its own public key is included in the public key list, Λ, and the number of keys in Λ equals to the number of participating administrators. If all verification passes, TAD notifies its administrator via the display. The only task that each administrator needs to perform is to visually check that all TADs display verification success messages. KISS introduces an additional computational overhead (e.g., remote attestation and quorum checking) compared to traditional system bootstrapping. However, we argue that this cost is acceptable,

considering the security guarantees it achieves.

Audit Log. During the server bootstrap, malicious administrators may inject spurious configuration files with a small quorum threshold, or even forge administrator public keys. These administrators are then capable of passing the quorum check that is necessary for any key management operations. In KISS, the server software maintains an operation log to record all of the system administration operations, including bootstrap operations. This helps legitimate administrators/auditors detect any insider attacks during the server bootstrap. In addition, the audit log helps relaxes the quorum control and improves system usability. Becasue all key management operations are held accountable, KISS may allowing a smaller number of administrators or even merely one to perform operations.

The audit log is stored in the untrusted regime. The KISS server software maintains an aggregated hash of the log entries in the TPM non-volatile memory (NVRAM). The TPM NVRAM access-control (similar to sealed storage) ensures that only KISS server software can access/update that hash, Note that frequent NVRAM updates are impractical on TPM. To minimize NVRAM updates, we leverage an update mechanism that is similar to the PCR-NVRAM two stage update technique presented in [92]. During the audit procedure, the auditor verifies the integrity of the log by recomputing the aggregated hash and comparing it with the hash stored in TPM NVRAM.

7.5.2 Client Bootstrap and Registration

Bootstrapping a KISS client is similar to the server bootstrap. A quorum of administrators verifies the authenticity of the KISS hypervisor, wimpy kernel, client software, and its configuration file. The client software securely sends its public key, K_{Cli}, to each of the participating TADs, and collects the device public keys K_{TAD_i} (generated during the server bootstrap). The configuration file sent to the client software differs from the one established during the server bootstrap. It contains the server public key, K_{Srv}, and the client-side system parameters (e.g., access-control policies of key usage, user authentication information, and the corporate application information). These client-side configurations are used in the fine-grained key usage control (See Section 7.7). Upon a successful client bootstrap, TADs establish Cli-TAD cryptographic channels with the KISS client, which allows subsequent client administration.

The administrators then register the client to the server by sending the client software public

key, K_{Cli}, to the server software, via Srv-TAD cryptographic channels. This establishes the Srv-Cli cryptographic channel (see Figure 7.3(b)). This channel diffs from the Srv-/Cli-TAD channel in that it provides both secrecy and integrity protection to the data transferred between the server and the clients (e.g., KISS product keys).

7.6 Secure System Administration

This section describes how the KISS administrators perform local and remote operations using their TADs and remote managers. Unlike traditional KMS, our remote management mechanism introduces a very small TCB that consists of a thin terminal, the KISS hypervisor, the wimpy kernel, the user-interface devices on KISS manager, and the simple TADs. In addition, it enables flexible administrative policies for better usability.

Secure Local Management. Administrators physically present at the KISS server or client connect the TADs directly with the machines to perform management. TADs first perform remote attestation to verify that the connected KISS machine is executing the desired hypervisor, wimpy kernel, KISS software, and trusted paths. Thus, any command input (or KISS system output) is securely directed to (or displayed by) the KISS server/client software. The administrators also use the TADs to authenticate their command input, by allowing the KISS server/client to display the command input with its digest (a cryptographic hash, $H(input)$) to the administrators. The alleged digest $H(input)$ is sent to the TADs via untrusted USB connection. The administrator confirms that the digest value displayed on his/her TAD is identical to the one on the server/client display. Then, the administrator press a button on the TAD to generate an authentication blob (digital signature) on digest $H(input)$ with the Srv-/Cli-TAD channel keys. The KISS server/client software verifies this blob to ensure the authenticity of launched commands.

Secure Remote Management. Administrators not physically present at the KISS server or client leverage the KISS managers and the TADs to perform maintenance tasks. The KISS manager software is isolated from the untrusted regime, and connects with the user-interface devices via trusted paths. The administrators not only use their TADs to authenticate the command input (the same as in local management), but also to verify the authenticity of the system output returning from the KISS server or client software. The KISS server/client generate similar au-

Category	Operations	Local or remote?	Quorum or any?	Manual or automatic?
1	server bootstrap, adding administrators	local	quorum	manual
2	server software and config update, removing administrators	either	quorum	manual
3	client bootstrap	local	either	manual
4	client registration, software and config update (e.g., change key usage control policy)	either	either	manual
5	server/client key life-cycle operations (e.g., key generation, distribution, usage)	either	either	either

Table 7.2: **KISS System Operation Categorization.**

thentication blobs for each of their responses, using the Srv-/Cli-TAD channel keys. The KISS manager software recomputes the digest $H(response)$, and displays it to the administrators via the trusted paths. It also forwards the digest and the authentication blob to the TADs. The TADs verify the authenticity of the blobs, and display the digest on the screen. If the two digests are identical, the administrators trust that the response indeed originated from the KISS server/clients. Note that our remote management mechanism can be extended to protect the secrecy of the command input and system output, and avoid the hash computation overhead and comparison efforts (Section 7.9).

Administrative Policies. KISS fully considers the balance between security and usability when making administrative policies. We categorize different system operations according to their administrative requirements, as is shown in Table 7.2.

In KISS, only three operations require the physical presence of administrators at the KISS server/client; the majority of operations can be performed remotely. In Category 1, server bootstrap and adding new administrators require the physical presence of a quorum of administrators. These two operations bootstrap cryptographic channels between TADs and the KISS server software and require our server bootstrap protocol (Section 7.5.1). Client bootstrap also mandates the physical presence of administrators, because administrators scan the TPM public key to their TADs.

In KISS, only a few operations mandate a quorum of administrators. We require all server-side administrative operations in Category 1 and 2 to be performed by an administrator quorum

in an attempt to *prevent* malicious insider attacks on the KISS server. However, once the server audit log is bootstrapped, all subsequent client-side administrative operations in Categories 3 and 4 and server/client key life-cycle operations in Category 5 could possibly relax the quorum requirement, because we can always *detect* insider attacks by analyzing the audit log.

In addition, for efficiency and usability, all Category 5 operations can be automatically performed by the KISS server/client software, without the involvement of administrators. For example, once an authorized corporate application requests a new key, the KISS client software can immediately contact the server for the new key. These automatic operations are controlled by the administrator-configured key usage policies (see Section 7.7), and can be recorded in the server audit log (or similar audit logs on clients).

7.7 Fine-grained Key Usage Control

This section explains how the KISS client software, hypervisor, and wimpy kernel perform fine-grained control of key usage. Figure 7.5 presents a typical workflow where a user executes a KISS-capable application that uses the cryptographic key generated by KISS.

Application Verification. The user selects the corporate application he/she intends to run via the untrusted regime (e.g., via a pop-up dialog by the OS). The OS loads and executes the selected corporation application and notifies the KISS hypervisor of the application execution. The hypervisor creates a corporate regime and protects the executed application in this regime. The hypervisor then measures that application and the wimpy kernel and sends the measurement as the application identifier to the KISS client software. The software compares the received measurement with the known-good value in its application database and notifies the result to the user via trusted paths. Recall that the authorized application database in the KISS client software was configured during the client bootstrap and can be updated by the administrators via remote management.

The KISS-capable corporate applications are not legacy applications. They are developed to execute in corporate regimes, communicating with the wimpy kernel instead of the OS [83, 84]. Note that recent research [54] eases this development effort by allowing protected applications to securely use OS services. The corporate application should also be modified to communicate

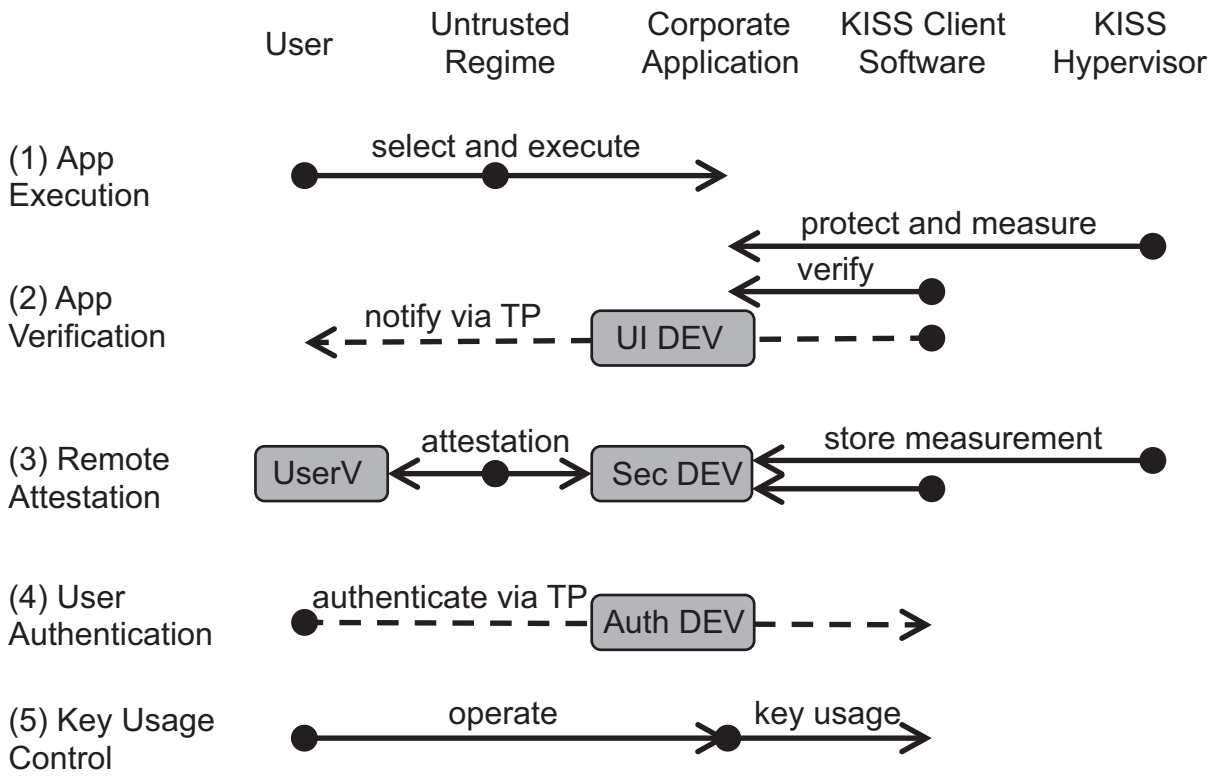


Figure 7.5: **Work Flow of Key Usage Control on KISS Client.** Dashed lines are interactions via trusted paths. UI, Sec, and Auth Dev are identical to those in Fig. 2. UserV denotes the users' dedicated verifier that can remotely attest to the KISS client.

with the KISS client software for key usage. While allowing key usage control, this introduces context switch overhead between the application and the KISS client software. A corporate application can be a stand-alone application (e.g., a KISS-capable document editor) or the security-sensitive modules of a legacy application that uses cryptographic keys (e.g., the ServerKeyExchange authentication module in an HTTPS server software). This is an application-specific design choice that depends on the application complexity (e.g., how the application is modularized and privilege-separated) and the strictness of the key usage control policy (application-wise or module-wise).

Remote Attestation. To trust the application verification results displayed in last step, and to defend against subtle user-oriented credential stealing attacks (e.g., tricking the user to input passwords), the users should leverage a small, dedicated device, called UserV, to attest that they are interacting with the correct KISS software and corporate applications. The UserV is similar

to, but much simpler than TAD. The only task of the UserV is to perform standard remote attestation to the KISS hypervisor, the wimpy kernel and the client software. It does not generate or store any secrets (e.g., shared secrets or private keys). It merely needs one button to start the attestation, and a LED to display attestation results [137]. Upon successful remote attestation, the user verifies that the application displayed is the one that he/she intends to run. Otherwise, the user should stop interacting with the corporate applications to prevent any sensitive information leakage.

User Authentication. In order to use the corporate application, the user needs to authenticate the KISS client software. If the authentication fails, the KISS hypervisor immediately teminates the corporate application. KISS can support all types of common authentication methods (knowledge, inherence, and ownership-based) and multi-factors authentication. For knowledge-based authentication (e.g., password, PIN) or inherence-based methods (e.g., fingerprint scanning, voice pattern recognition), the users should leverage the trusted paths between the authentication-relevant devices (e.g., keyboard, fingerprint reader) and the KISS client software. With the trusted paths, malware in the untrusted regime cannot intercept the users' credentials [1]. For ownership-based authentication, users usually carry certain authenticators (e.g., smart cards, security tokens) and rely on the embedded secrets to respond to the challenges of the KISS client software. No trusted path is needed between the authentication devices (e.g., smart card reader) and KISS client software. For all the authentication methods above, the KISS client software should be configured with necessary authentication information (e.g., password hash, fingerprint database, and keys to verify smart cards' responses) by the administrators during client bootstrap or remote management.

Key Usage Control. During execution, the corporate applications trigger key usage requests to the KISS client software via KISS hypervisor, or via similar communication channels as the Wimp-OS channels (Section 5.4). The key usage requests can be driven by the users (e.g., the user wants to encrypt a confidential document) or by the application itself (e.g., the HTTPS web server software digitally signs its ServerKeyExchange messages). Upon receiving the key usage

[1]Even if the attackers have the users' credentials, they still need to physically be present at the KISS client to input the credentials. The KISS client software takes inputs directly from the hardware devices via trusted-paths, not from any software.

requests, the KISS client software knows the identifiers of the requesting application and the user. The KISS client software leverages the pre-configured access control policies to decide whether to approve or deny the requests. KISS supports flexible access-control policies with different granularity. It can perform simple ON/OFF key usage control. For example, KISS allows user Alice to use the authorized document editor to decrypt her own documents, but restricts other users who are using the same editor or Alice using different software (e.g., an email client, or a compromised document editor) from accessing the documents. It can also support more complicated policies, such as rate limiting, access time restriction, and role-based access control. The administrators decide the access control policies, configure them in the client software during bootstrap, and update the policies via remote management.

7.8 Security Analysis

This section analyzes potential attacks on KISS and our defense mechanisms.

System Bootstrap. During the system bootstrap, malicious administrators or malware on KISS server/clients may tamper with the code or configurations of the hypervisor, the wimpy kernel, and the KISS software. The benign administrators can detect this attack via TAD remote attestation. Malicious administrators may also launch Sybil attacks by creating bogus administrator accounts during the bootstrap process. As described in Section 7.5, the administrators visually check that all TADs display success messages. This confirms that the server/client software receives only the public keys of the participating TADs, not any bogus key.

Key Life-cycle Operations. Malware in the server/client untrusted regime may try to modify the KISS software code, interfere with its execution, or access the cryptographic keys generated or stored by the software. The KISS hypervisor prevents these attacks by protecting the code and data memory of the KISS software from the untrusted regime. When the KISS software is at rest, the cryptographic keys are protected by the TPM sealed storage. Only the same KISS software can unseal the keys; the malware or the compromised KISS software cannot. Malware attacks that compromise the client software to trigger unintended KISS server operations also fail, because the client private key for authenticating operation requests is sealed by the TPM.

System Administration. Any manual administrative operation requires at least one authorized

TAD. The malware cannot steal the private keys in TADs, nor can it intercept other administrator credentials, such as bio-metric information or passwords, which are transferred to the KISS software via trusted paths, and/or Srv-/Cli-TAD authentic channels (Section 7.6). Similarly, the administration commands and system output are also transferred via trusted paths or Srv-/Cli-TAD channels. The attackers cannot modify any command or forge any system output. Though malicious administrators may use their TADs to execute operations that do not require the quorum, those operations are recorded in the server/client audit log and held accountable.

Key Usage Control. As described in Section 7.7, unauthorized applications and users cannot bypass the KISS hypervisor, the wimpy kernel, and the client software to use any cryptographic key. A malicious administrator may intentionally update the application and user database in the KISS client software to allow key mis-uses. However, this administrative operation is recorded in the client audit log and held accountable. The malware cannot steal users' authentication credentials, because those credentials are delivered to the KISS client software via trusted paths. The users also verify that they are communicating with the authentic KISS client software before inputting their authentication credentials.

7.9 Discussion

This section discusses the KISS system extensions that provide higher security guarantees and address some real-world application issues (e.g., cloud computing).

Administrative Operation Secrecy. Section 7.6 describes how KISS protects the authenticity of administrative inputs and system outputs. We could extend KISS to protect input/output secrecy by establishing encryption keys for Srv-/Cli-TAD channels, and an extra trusted path on KISS manager between the manager software and the USB controllers that connects the TAD. Note that this trusted path also avoids the hash computation overhead and comparison efforts described in Section 7.6, because it protects the authenticity of data between the TAD and the manager software.

TPM 2.0 Enhanced Authorization. The TPM 2.0 library specification [123] is currently under public review. It supports enhanced authorization by allowing the construction of complex authorization policies using multiple authorization qualifiers (e.g., password, command HMAC,

PCR values, NVRAM index values, and TPM time information). KISS can reduce its TCB by offloading some authorization checking to TPM 2.0, given that it can securely collect the authorization information, deliver it to the TPM, and protect it from the untrusted OS. However, it is not clear how the performance of TPM authorization checking compares to that of the KISS software.

Compatibility to Cloud Computing. The KISS hypervisor is a small, dedicated hypervisor that runs on bare metal. If the KISS servers and clients are deployed on an enterprise private cloud, we could consider (1) integrating KISS hypervisor with the full-functioning the hypervisor/VMM or (2) adding nested virtualization support [135] to KISS hypervisor and running the full-functioning hypervisor/VMM upon it. Option (1) has much larger TCB, but has better compatibility and performance than option (2).

7.10 Related Work

We review the state-of-the-art key management systems and related technologies. The first category of KMS solutions are **software-based solutions**, such as OpenSolaris Crypto KMS Agent Toolkit [88], IBM Tivoli Key Manager [57], and StrongKey open-source KMS software [115]. These rely on process isolation, user privilege control, and file permissions provided by the OS to protect cryptographic keys and control the applications' access to them. Their implementation of trusted paths for administrators is based on the OS services (e.g., Ctrl+Alt+Del command or trusted window manager). Compared with KISS, the software-only approaches are more cost-effective and easier to deploy on commodity computers (e.g., no hypervisor, work with legacy corporate applications, no security hardware requirement). However, they rely heavily on the large OS and thus fail to provide the same level of security assurance as KISS.

An alternative is leveraging high profile **HSMs** [55, 89, 97, 99, 120]. An HSM provides hardware-level tamper-resistant protection to cryptographic keys and algorithms for both run-time and at rest, while KISS provides hypervisor-based software isolation for keys and algorithm during run-time, and TPM level hardware protection for keys at rest. For performance, an HSM may employ customized hardware engine to accelerate cryptographic algorithms. It is more efficient than KISS and the software-only solutions. The downside of the HSM is that it fails to

provide the same secure level of key usage control as in KISS, as we have explained in Section 7.1. Indeed, the KISS system can be extended to serve as the key usage control front end of the HSM, which may achieve the benefits of both systems. For system administration, some high-end HSMs [99, 120] achieve the same level of security guarantees as KISS (e.g., quorum control, trusted paths using on-HSM I/O devices, remote management using administrator devices). However, their administrator devices introduce larger TCB than KISS (e.g., complicated key management software stack for interpreting commands and operation results). The HSM administrators usually blindly trust the devices, and have no intuitive way to verify their software status.

There are research proposals that seek to offer similar protections for user credentials in the key management systems. Wallet-based web authentication systems (e.g., [41]) isolate user credentials in an isolated domain (e.g., a L4 process upon L4 Micro-kernel) during run-time and protect the credentials at rest by TPM-based sealed storage. They only allow authenticated websites to access their own credentials. These systems have a reasonable TCB size, but do not provide fine-grained and flexible key usage control as in KISS (e.g., user-based control). Bugiel and Ekberg [20] propose a system that only allows the application to access its own credentials (protected in mobile trusted module). The On-board Credentials (ObC) [73] approach enables an isolation environment (like KISS) for both third-party credential algorithms/applications and credentials, on smartphones and conventional computers. However, one faces multiple challenges extending these systems for corporate key management. For example, ObC approach lacks protection mechanisms against malicious administrators and do not support trusted paths for administrator management. PinUP [32] binds files to the applications that are authorized to use them by leveraging the SELinux capability mechanisms. This suggests that PinUP introduces a larger TCB to provide security assurance on par with KISS.

7.11 Summary

In this chapter, we present a trustworthy key management system architecture by leveraging micro-hypervisor-based code isolation and wimpy-kernel-based I/O isolation. KISS aims to reduce cost by relying solely on commodity computer hardware, and minimize the system TCB

by the design of micro-hypervisor and wimpy kernel, and lightweight administrator devices. KISS is the first key management system to support fine-grained control of key usage. KISS is bootstrapped and operated in the face of software attacks from malware in the OS and insider attacks from malicious administrators. KISS provides user-verifiable trusted paths and simple dedicated external devices for secure system administration. KISS showcases the benefits of applying trusted computing techniques, program isolation, and I/O isolation to designing trustworthy systems. KISS offers trustworthy key management systems at a price point that enables wide-spread adoption beyond the security-sensitive financial or governmental institutions.

Chapter 8

Related Work

This chapter compares our on-demand I/O isolation system with the related work of code isolation, device driver isolation and I/O isolation, as shown in Sections 8.1, 8.2, and 8.3, respectively.

8.1 Isolated Execution Environments

Recent advances on isolated execution environments (IEEs), using both software [83, 84, 100, 113, 114, 127] and hardware architectures [86], 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 [91], file system and network services [21, 22, 24, 54, 80], inter-IEE communication [114], and limited user trusted path [137]. 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 [21, 22, 24, 54, 91] 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 [95] 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 ser-

vices is much larger than ours.

8.2 Device Driver Isolation and Decomposition

Several approaches [18, 42, 78, 79, 87, 117, 131] 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 [117]. LeVasseur *et al.* [79] and Nikolaev *et al.* [87] propose running unmodified device drivers of guest operating systems in separate virtual machines. SUD [18] moves device drivers to an emulated Linux kernel environment in user-space. Leslie *et al.* [78] 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.* [42] 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.* [131] 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 [70, 110] 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 [70] 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.

8.3 I/O Isolation Systems

8.3.1 Limited Device Support

Security kernels [14, 30, 46, 52, 104, 133], isolation kernels [94], and micro-hypervisors [23, 138] 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.* [137] illustrate a limited form of user-verifiable trusted paths to application-code modules, protected by a micro-hypervisor [83].

Filyanov *et al.* [37] proposes an isolated software module to control user-centric I/O devices (e.g., keyboard and display) and enables a remote server to verify that a transaction summary is confirmed by a local user's keyboard input. However, the UTP system does not provide local, user-verifiable evidence of the output trusted path; i.e., malicious code can display a fake transaction output to the user. Unfortunately, UTP does not defend against all the attacks we address, e.g., MMIO mapping attack, MSI spoofing, IPI spoofing, and attacks that exploit the DMA request ambiguity.

The DriverGuard system [25] protects the confidentiality of the I/O flows between commodity peripheral devices and some Privileged Code Blocks (PCBs) in device driver code. Moreover, DriverGuard does not claim that they protect the I/O data from MMIO mapping attacks. Thus, the I/O data in PCBs may still be revealed to a potentially compromised OS. In addition, DriverGuard's I/O port access isolation is incomplete. PCBs are in a higher privilege level than the OS kernel, and thus can access any I/O ports of any other devices. 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 simplicity. Our sys-

Table 8.1: **A Comparison of Different I/O Isolation Architectures.** "Dom0" denotes the monolithic root domain in the "split-driver" model [15]. "Structured dom0" represents the root domain in the hypervisor model where each device driver is separated into a VM domain ("Dev VMs" in this table).

	Monolithic OS/hypervisor	**Hypervisor w/ dom0**	**Hypervisor w/ dev pass-through**	**Hypervisor w/ drv domains**	**Our System**
Secure App	app	VM	app	VM	app
Device Driver	OS/hypervisor	dom0	guest OS	drv VMs	app
Others in TCB		hyp	hyp+dom0	hyp+Dev VMs	micro-hyp+WK
TCB Size (SLoC)	>10M	>1.2M	>1.2M	>1.2M	≈30K

tem 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.

8.3.2 Static Device Allocation

Separation kernels [49, 98, 118] 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 [69, 119] dedicates I/O devices with virtualization support (e.g., SR-IOV [63]) 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. LockDown [129] and SecureSwitch [116] partition system sources, such as CPU, memory, hard disks, network interface cards, and graphic cards, among a untrusted OS and a trusted OS, as proposed in Lampson's red/green system separation idea [75]. The resource partition is statically configured by a micro-hypervisor or the BIOS. Different from our system, these systems do not claim to support trusted execution environment for applications, nor do they support the isolation or partition of other plug-and-play devices, such as USB keyboards, cameras, thumb drives.

8.3.3 OS/Hypervisor-based Systems

Trusted path on the DirectX system [76] and the Trusted Input Proxy system [16] reserve dedicated areas of the screen to output the identity and status of the current applications. These systems are built atop large operating systems. The Not-a-Bot system [51] implements a software module to capture human keyboard inputs and to use them to identify human-triggered network traffic. This system builds a small code module upon a reduced version of the Xen hypervisor and mini-OS kernel, which is still around 30K SLoC. Saroiu and Wolman propose a system that runs a root virtual machine (e.g., a *dom0* in Xen) to read a mobile device's sensors [103]. This design trusts a full virtual machine monitor, and only protects data integrity. Similarly, Gilbert et al. propose a trustworthy mobile sensing architecture [44] that enables a remote data receiver to verify that the sensed data is from the intended sensors and has only been manipulated by trusted software (e.g., the intended sensing application, trusted OS, and VMM).

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 [15], then to hypervisors with disengaged privileged domains [27], and finally to hypervisors with isolated driver domains [40, 87] (Table 8.3.2). 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 [82]) or para-passthrough support (e.g., BitVisor [112]) 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.

Hypervisors that are based on the "split-driver" model [15] move device management from the hypervisor to a root domain, *dom0*, which is frequently large and unstructured [27]. Hence, it merely exposes the trusted-path to a different set of attacks from those possible in a monolithic OS (e.g., Windows) or VMM (e.g., VMware Workstation), but does not eliminate these attacks.

Equally undesirable is that a program endpoint typically communicates with the wimp device of a trusted path via the *untrusted* guest OS upon which it runs.

Hypervisors with device pass-through support [82] (e.g., Xen, KVM) or para-passthrough support (e.g., BitVisor [112]) enable exclusive assignment of I/O devices to a specific guest VM. However, the driver of the pass-through device is still in the guest VM and co-exists with the guest OS. There is no device driver isolation in this mechanism. Also, a compromised root domain, *dom0*, can still break the device isolation and communication path isolation. For example, typically the user must explicitly "hide" the pass-through devices from dom0 via some administrative settings in dom0.

Another recent advance is the ability to structure device drivers in a hypervisor-based system into *driver-domains*, giving different driver virtual machines (VMs) direct access to different devices [27, 96]. However, this work only demonstrates how to isolate device driver *address spaces* and Direct Memory Access (DMA). It does not fully isolate devices from compromised OS code in other administrative domains (e.g., system-wide configurations for I/O ports, Memory-Mapped I/O (MMIO), and interrupts remain unprotected). Manipulated devices may still breach the isolation between device drivers and gain unauthorized access to the registers and memory of the isolated 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.

8.3.4 Special Devices

Some systems take advantage of special hardware devices – equipped with data encryption capability – to establish secure I/O channels with isolated software [53, 72, 85, 130]. Saroiu et al. [103] propose another sensor reading protection system based on the assumption that the reading is digitally signed by a TPM on the sensor (c.f. [31]). The Zone Trusted Information Channel (ZTIC) is a dedicated device with a display, buttons and cryptographic primitives [77, 130]. ZTIC enables users to securely confirm their banking transactions via the dedicated display and button, completely bypassing the user's computer, which may be infected by malware. The Bumpy system requires a special keyboard that supports cryptographic primitives including encryption

and certificate validation [85].

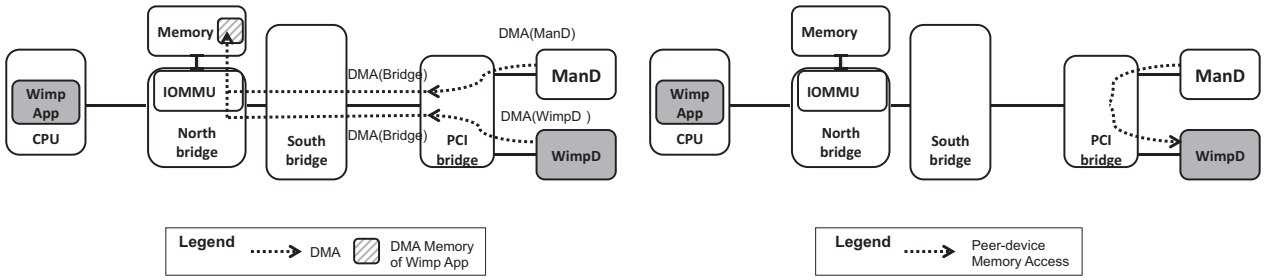
Solutions using cryptographic channels and special devices with cryptographic primitives often require the protection of secrets in user-level programs and/or commodity I/O devices, which is often impractical and raises fundamental usability concerns for commodity platforms. How could a user securely set or change the secret key within a trusted-path program endpoint without using some trusted path to reach that program? Our system avoids the attendant secure key management issues and special devices, and supports protected I/O channels to commodity peripheral devices.

Chapter 9

Discussion and Future Work

This chapter discusses various software design choices and I/O hardware modifications that may lead to simpler, higher performance, more secure and usable system implementations in the future.

9.1 I/O Hardware Modifications



(a) DMA Request Ambiguity (b) Unmonitored Peer-device Communication

Figure 9.1: **Architectural Challenges for I/O Isolation.** ManD denotes a device outside wimps that is manipulated by compromised OS.

DMA Request Ambiguity. DMA-capable peripherals that are the downstream of one or more PCI/PCI-to-PCIe bridges cannot be uniquely identified by the system's IOMMU, enabling devices in such locations to impersonate other nearby devices. Manipulated devices may leverage this attack to violate the isolation of the DMA memory region of the isolated device [101].

We first describe a software work-around to this DMA request ambiguity problem, which provides the desired security properties but incurs significant performance overhead. The wimpy

kernel identifies all devices behind the same PCI/PCI-to-PCIe bridges that connect the wimp device by enumerating the PCI configuration space. Before executing the wimp app, these devices are put into a quiescent state (e.g., sleep, or a pending state). The wimpy kernel can verify the devices' quiescent state by reading device-specific status registers before approving the execution of the wimp app. During the wimp app's execution, the wimpy kernel prevents the compromised OS from waking the pending devices by interposing on the relevant I/O ports and memory ranges (Sections 5.1.1 and 5.1.2).

However, quiescing all devices sharing the same PCI/PCI-to-PCIe bridge with a wimp device reduces I/O performance. During the execution of the wimp app, an OS cannot communicate with any of those devices. To eliminate this uncomfortable trade-off between security and performance, we suggest several potential architectural changes. First, motherboard manufacturers can assign only one PCI device to each PCI or PCI-to-PCIe bridge. Alternatively, the PCI/PCI-to-PCIe bridge design specifications might be changed to transmit the identifiers of the originating devices when relaying I/O transactions. A third proposal is to enhance the DMA request ID specifications to include additional information, such as the contents of the PCI vendor ID and device ID configuration register fields. This information should not be changed or replaced by PCI/PCI-to-PCIe bridges.

Unmonitored Peer-Device Communication. Manipulated PCI/PCIe and USB 2.0+ devices may establish peer-to-peer connections with a wimp device, bypassing all isolation mechanisms implemented by the hypervisor [102, 121, 122]. PCI/PCIe peer-to-peer communication complies with the PCI/PCIe specifications [19, 109], and thus cannot be denied by the device itself. The MMIO protection can neither prevent nor detect peer-to-peer communication, since this communication operates directly on the internal memory of the communicating devices. In addition, the IOMMU cannot mediate communication for PCI and USB devices that are connected to the southbridge chip, because the IOMMU is integrated into the northbridge chip.

To prevent PCI peer-to-peer communication, we propose using the new PCIe Access Control Services (ACS) [9]. The ACS on an I/O bus/bridge will actively check the originator's identity in I/O requests, and prevent I/O command spoofing and unauthorized I/O access. The I/O isolation system configures the ACS on all corresponding bridges to prevent any peer-to-peer commu-

nication between the wimp device and other devices. The I/O isolation must also protects the ACS configuration using the mechanisms described in Section 5.1. The remaining problem is that ACS is not yet a common feature of the I/O architecture, and most current PCI bridges and chipset hardware do not implement it.

The prevention of USB On-The-Go (OTG) peer-to-peer communication [124] is easier, because the communication only succeeds when both communicating devices enable OTG and comply with OTG protocols. Thus, the wimpy kernel or wimp app can explicitly configure the wimp device to disable USB OTG.

9.2 Defend against Firmware Attacks

A future direction would be relaxing the adversary model and considering devices with compromised or malicious firmware. We integrate a software attestation mechanism [106, 107, 134] to verify wimp device's firmware authenticity and, if necessary, eliminate any known malware from its firmware [68, 121, 122]. However, verification of firmware authenticity cannot, and is *not* intended to, guarantee full *firmware correctness* with our system; e.g., the elimination of exploitable firmware vulnerabilities is an important, separate assurance exercise not addressed here. Here the correctness of device firmware means that (1) devices respond to few wimpy kernel and *mHV* commands, such as those to enter a quiescent state, report MMIO and PMIO mapping, and disable peer-device connectivity, and that (2) device firmware does not actively tamper with the I/O data or leak the data to external entities. Note that even if all device firmware is *correct* but devices are *not* isolated from untrusted OS code, devices could still be misused by the untrusted OS to breach I/O isolation.

At a high level, the software attestation process is applied to wimp device's firmware as follows [106, 107, 134]: A small verification function loaded in device wimp device proves to the verifier wimp app that it is operating correctly and is isolated from wimp device's firmware. To do this, the verification function uses a time-bounded, challenge response protocol. Then it computes a cryptographic hash over wimp device's internal memory content and sends the result back to the verifier wimp app, which compares the computed hash value with the expected value of the unmodified firmware to confirm the authenticity of the firmware.

9.3 Cope with Other I/O Architecture

QuickPath/HyperTransport. Intel's QuickPath Architecture [59] provides high-speed, point-to-point interconnects between microprocessors and external memory, and between microprocessors and an I/O hub. This architecture is designed to reduce the number of system buses (e.g., replace the front-side bus between the CPU and memory), and to improve interconnect performance between CPU, memory, and I/O peripherals.

However, QuickPath is not intended to, and indeed does not, solve the device isolation problems any more than the commodity x86-based I/O architecture. Our design can directly apply to the QuickPath architecture. The changes are merely in the composition of the chipset hardware: for the x86 architecture, a northbridge and a southbridge are involved, whereas in the QuickPath architecture, a QuickPath controller and an I/O hub are used. In addition, memory management units are directly embedded in QuickPath-enabled CPUs. Our I/O isolation system design is equally applicable to other similar I/O architectures, including AMD's HyperTransport [56].

ARM. Recent advances to ARM's TrustZone security extensions [6] and virtualization support [125] make the application of our I/O isolation design to ARM-based I/O architectures possible [128]. ARM's TrustZone Security Extensions [6] split a single physical processor state to safely and efficiently execute code in two separate worlds: a more-privileged secure world, and a normal world. System designers can leverage multiple hardware primitives, such as TrustZone-aware memory management units, DMA and interrupt controllers, and peripheral bus controllers, to partition critical system resources and peripheral devices and assign them to different worlds. In addition, with forthcoming virtualization support [125], it is possible to run a hypervisor in a special mode of the normal world that can optionally trap any calls from the normal world's guest OS to the secure world. Porting our I/O isolation system to the ARM architecture, and supporting a wide range of applications on mobile and embedded platforms, is future work.

Chapter 10

Conclusion

To be trustworthy, security-sensitive applications must be formally verified, and therefore they must be small and simple (i.e., wimps). High-assurance security-sensitive applications do not exist on commodity platforms today, where large and untrustworthy systems (i.e., giants) are typically chosen to address diverse user needs in different application scenarios.

We argue that the two mainstream mechanisms to guarantee trustworthiness of security-sensitive applications – patching and restructuring the giants – will remain ineffective. Constantly applying security patches to commodity giants and attempting to secure whole systems and the applications run upon them does not work because it is extremely difficult to obtain complete and accurate adversary definitions for large commodity systems, which gives attackers a persistent cost advantage over defenders. Fully restructuring commodity giants and building security from ground up is equally unattractive, because the immense technical complexity of restructuring would require years of design and engineering effort, which will never keep pace with the innovation speed in the mass market. In this thesis, we present a third mechanism, an on-demand add-on, to get rid of the daunting complexities of patching and restructuring and to achieve benefits of both giants and wimps. This mechanism keeps commodity giants unchanged, plugs in a minimal system to guarantee the giant-wimp code isolation and other security properties of the wimps, and unplugs when the applications are done.

Trustworthy wimp 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 thesis presents a security architecture based on a trusted add-on, called wimpy

kernel, which provides these services without bloating the underlying trusted computing base. It also presents a concrete implementation of the wimpy kernel for major I/O subsystems, namely the PCI and the USB subsystems, and a variety of device drivers. The size and complexity of the wimpy kernel are minimized by safely outsourcing I/O subsystem functions to an untrusted commodity operating system. In other words, wimps are not only isolated from giants, but also securely composed with them: they rely on giants' services but only after efficiently verifying their results. The reduction of the wimpy kernel is further achieved by exporting driver and I/O subsystem code to wimp applications. Experimental measurements show that we achieve the desired minimality goals for the wimpy kernel, which has never been done in any previous I/O isolation system. For example, over 99% of the Linux USB code base is removed from the wimpy kernel.

We demonstrate the power of this new architecture using two applications, secure user interface [137, 139] and trustworthy corporate key management systems [138]. These are the two most important primitives for any security-sensitive application [93, 136] because these applications always need a trusted path to reach the users and a infrastructure to bootstrap and distribute cryptographic keys for secure data storage and transfer. The systems designed in this thesis open up new possibilities and facilitate the wide-spread adoption of high-assurance security-sensitive applications in the commodity market.

Bibliography

All of the URLs listed here are valid as of May, 2014.

[1] "eXtensible Modular Hypervisor Framework," http://xmhf.org.

[2] "GNU Pth - The GNU Portable Threads," http://www.gnu.org/software/pth/.

[3] "SeaBIOS," http://www.coreboot.org/SeaBIOS.

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

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

[6] T. Alves and D. Felton, "TrustZone : Integrated Hardware and Software Security," *ARM white paper*, 2004.

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

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

[9] AMD and HP, "PCI Express Access Control Services (ACS): PCI-SIG Engineering Change Notice," 2006.

[10] J. P. Anderson, "Computer security technology planning study. volume 2," Air Force Electronic Systems Division, Tech. Rep. ESD-TR-73-51, 1972.

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

[12] C. R. Attanasio, P. W. Markstein, and R. J. Phillips, "Penetrating an operating system: a study of VM/370 integrity," *IBM System Journal*, vol. 15, no. 1, pp. 102–116, 1976.

[13] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proc. ACM Conference on Computer and Communications Security*, 2011.

[14] BAE Systems Information Technology LLC, "Security Target, Version 1.11 for XTS-400,

Version 6," 2004.

[15] 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.

[16] K. Borders and A. Prakash, "Securing network input via a trusted input proxy," in *Proc. USENIX Workshop on Hot Topics in Security*, 2007.

[17] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. OReilly, 2006.

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

[19] R. Budruk, D. Anderson, and E. Solari, *PCI Express System Architecture*. Addison-Wesley Professional, 2003.

[20] S. Bugiel and J. Ekberg, "Implementing an application-specific credential platform using late-launched mobile trusted module," in *Proc. ACM STC*, 2010.

[21] 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," Parallel Processing Institute, Fudan University, Tech. Rep. FDUPPITR-2007-0801, 2007.

[22] 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.

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

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

[25] 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*, vol. 16, no. 2, pp. 1–30, 2013.

[26] D. D. Clark and M. S. Blumenthal, "The end-to-end argument and application design: the role of trust," *Federal Communications Law Journal*, vol. 63, no. 2, pp. 357–390, 2011.

[27] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: Security and functionality in a commodity hyper-

visor," in *Proc. ACM Symposium on Operating Systems Principles*, 2011.

[28] Common Criteria for Information Technology Security Evaluation (CC), "Common methodology for information technology security evaluation," Version 3.1 CCMB-2009-07-004, 2009.

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

[30] Department of Defense, "Trusted computer system evaluation criteria (orange book)," DoD 5200.28-STD, 1985.

[31] A. Dua, N. Bulusu, W.-C. Feng, and W. Hu, "Towards trustworthy participatory sensing," in *Proc. USENIX Conference on Hot Topics in Security*, 2009.

[32] W. Enck, P. McDaniel, and T. Jaeger, "Pinup: Pinning user files to known applications," in *Proc. ACSAC*, 2008.

[33] D. R. Engler, M. F. Kaashoek *et al.*, "Exokernel: An operating system architecture for application-level resource management," vol. 29, no. 5, pp. 251–266, 1995.

[34] J. Epstein, C. Inc, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Danner, C. R. Martin, M. Branstad, G. Benson, and D. Rothnie, "A high assurance window system prototype," *Journal of Computer Security*, vol. 2, no. 2, pp. 159–190, 1993.

[35] N. Falliere, L. O. Murchu, and E. Chien, "W32.stuxnet dossier," Symantec, version 1.3, 2011.

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

[37] 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.

[38] S. Fleming, "Accessing PCI Express configuration registers using Intel chipsets," Intel White Paper no. 321090, 2008.

[39] J. Franklin, S. Chaki, A. Datta, J. M. McCune, and A. Vasudevan, "Parametric verification of address space separation," in *Proc. Conference on Principles of Security and Trust*, 2012.

[40] 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 Sys-*

tem and Architectural Support for the on demand IT InfraStructure (OASIS), 2004.

[41] S. Gajek, H. Löhr, A. Sadeghi, and M. Winandy, "Truwallet: trustworthy and migratable wallet-based web authentication," in *Proc. ACM STC*, 2009.

[42] 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.

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

[44] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall, "Toward trustworthy mobile sensing," in *Proc. Workshop on Mobile Computing Systems and Applications*, 2010.

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

[46] V. D. Gligor, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W.-D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan, "Design and implementation of secure Xenix," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 208–221, 1986.

[47] V. D. Gligor and B. G. Lindsay, "Object migration and authentication," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 607–611, 1979.

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

[49] I. GreenHills Software, "Integrity-178b separation kernel security target," http://www.niap-ccevs.org/st/st_vid10362-st.pdf, 2010.

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

[51] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy, "Not-a-bot: Improving service availability in the face of botnet attacks," in *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[52] M. S. Hecht, M. E. Carson, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterrer, V. D. Gligor, W. D. Jiang, A. Johri, G. L. Luckenbaugh, and N. Vasudevan, "UNIX without the superuser," in *Proc. USENIX Annual Technical Conference*, 1987.

[53] 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.

[54] 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.

[55] HP, "Enterprise Secure Key Manager," http://h18006.www1.hp.com/products/quickspecs/13978_div/13978_div.PDF.

[56] HyperTransport Consortium, "HyperTransport I/O link specification," Doc. no. HTC20051222-0046-0008 rev.3.10, 2006.

[57] IBM, "Tivoli Key Lifecycle Manager," http://www-01.ibm.com/software/tivoli/products/key-lifecycle-mgr.

[58] Intel, "Intel trusted execution technology – software development guide," Doc. no. 315168-005, 2008.

[59] Intel, "Intel's QuickPath architecture: A new system architecture for unleashing the performance of future generations of Intel multi-core microprocessors," 2008.

[60] Intel, "Intel trusted execution techonology," No. 315168-008, 2011.

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

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

[63] 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.

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

[65] Jeanne Meserve, "Sources: Staged cyber attack reveals vulnerability in power grid," http://edition.cnn.com/2007/US/09/26/power.at.risk/index.html, 2007.

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

[67] 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*, vol. SE-17, no. 11, pp. 1147–1165, 1991.

[68] K.Chen, "Reversing and exploiting an apple firmware update," in *Black Hat*, 2009.

[69] 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.

[70] 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.

[71] ——, "seL4: formal verification of an OS kernel," in *Proc. ACM Symposium on Operating Systems Principles*, 2009.

[72] 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/.

[73] K. Kostiainen, "On-board credentials: An open credential platform for mobile devices," Ph.D. dissertation, Aalto University, 2012.

[74] B. Lampson, "Software components: Only the giants survive," *Computer Systems: Theory, Technology, and Applications*, no. 9, pp. 137–145, 2004.

[75] ——, "Usable security: How to get it," *Communications of the ACM*, vol. 52, no. 11, pp. 25–27, 2009.

[76] H. Langweg, "Building a trusted path for applications using COTS components," in *Proc. NATO RTS IST Panel Symposium on Adaptive Defence in Unclassified Networks*, 2004.

[77] B. Laurie and A. Singer, "Choose the red pill and the blue pill: a position paper," in *Proc. Workshop on New Security Paradigms*, 2008.

[78] 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*, vol. 20, no. 5, pp. 654–664, 2005.

[79] 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.

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

[81] S. Lipner, T. Jaeger, and M. E. Zurko, "Lessons from VAX/SVS for high assurance VM systems," *IEEE Security and Privacy*, vol. 10, no. 6, pp. 26–35, 2012.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

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

[88] Oracle, "Opensolaris project: Crypto kms agent toolkit," http://hub.opensolaris.org/bin/view/Project+kmsagenttoolkit/WebHome.

[89] ——, "Oracle Key Manager," http://www.oracle.com/us/products/servers-storage/storage/tape-storage/034335.pdf.

[90] B. Parno, "Bootstrapping trust in a "trusted" platform," in *Proc. USENIX Workshop on Hot Topics in Security*, 2008.

[91] 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.

[92] ——, "Memoir: Practical state continuity for protected modules," in *Proc. IEEE Symp. on Security and Privacy*, 2011.

[93] B. Parno, Z. Zhou, and A. Perrig, "Using trustworthy host-based information in the network," in *Proc. ACM Workshop on Scalable Trusted Computing*, 2012.

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

[95] 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.

[96] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the art of virtualization," in *Proc. Ottawa Linux Symposium*, 2005.

[97] RSA, "RSA Data Protection Manager," http://www.emc.com/security/rsa-data-protection-manager.htm.

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

[99] SafeNet, "SafeNet hardware security modules," http://www.safenet-inc.com/products/data-protection/hardware-security-modules-hsms/.

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

[101] F. L. Sang, E. Lacombe, V. Nicomette, and Y. Deswarte, "Exploiting an I/OMMU vulnerability," in *Proc. International Conference on Malicious and Unwanted Software*, 2010.

[102] F. L. Sang, V. Nicomette, Y. Deswarte, and L. Duflot, "Attaques DMA peer-to-peer et contremesures," in *Proc. Symposium sur la Sécurité des Technologies de l'Information et des Communications*, 2011.

[103] S. Saroiu and A. Wolman, "I am a sensor, and I approve this message," in *Proc. Workshop on Mobile Computing Systems and Applications*, 2010.

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

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

[106] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla, "Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms," in *Proc. Symposium on Operating Systems Principals*, 2005.

[107] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Proc. IEEE Symposium on Security and Privacy*, 2004.

[108] N. Shachtman, "Exclusive: Computer virus hits u.s. drone fleet," http://www.wired.com/dangerroom/2011/10/virus-hits-drone-fleet/, 2011.

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

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

[111] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, "Design of the EROS trusted window system," in *Proc. USENIX Security Symposium*, 2004.

[112] 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.

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

[114] 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.

[115] StrongAuth, "StrongKey SKMS," http://www.strongkey.org.

[116] K. Sun, J. Wang, F. Zhang, and A. Stavrou, "SecureSwitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses," in *Proc. Network and Distributed System Security Symposium*, 2012.

[117] 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.

[118] W. R. Systems, "Wind river vxworks mils platform," http://www.windriver.com/products/platforms/vxworks-mils/MILS-3_PO.pdf, 2013.

[119] 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.

[120] Thales, "Thales hardware security modules," http://www.thales-esecurity.com/en/Products/Hardware%20Security%20Modules.aspx.

[121] A. Triulzi, "Project Maux Mk.II - "I own the NIC, now I want a shell!"," in *PacSec/core*,

2008.

[122] ——, "The Jedi Packet Trick takes over the Deathstar (or: "taking NIC backdoors to the next level")," in *CanSecWest/core*, 2010.

[123] Trusted Computing Group, "Trusted platform module library family "2.0"," 2011.

[124] USB Implementers Forum, *On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification, Revision 2.0 plus errata and ecn*, 2010.

[125] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," in *Proc. ACM SIGOPS Asia-Pacific Workshop on Systems*, 2011.

[126] VASCO, "Diginotar reports security incident," http://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx, 2011.

[127] 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.

[128] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: What security properties can my mobile platform give me?" in *Proc. International Conference on Trust and Trustworthy Computing*, 2012.

[129] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig, "Lockdown: towards a safe and practical architecture for security applications on commodity platforms," in *Proc. International Conference on Trust and Trustworthy Computing*, 2012.

[130] 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.

[131] 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.

[132] R. Wojtczuk and J. Rutkowska, "Following the white rabbit: Software attacks against intel VT-d technology," http://invisiblethingslab.com/resources/2011/SoftwareAttacksonIntelVT-d.pdf, 2011.

[133] C. Wright, C. Cowan, and J. Morris, "Linux security modules: General security support for the linux kernel," in *Proc. USENIX Security Symposium*, 2002.

[134] Yanlin Li and Jonathan M. McCune and Adrian Perrig, "SBAP: Software-based attestation for peripherals," in *Proc. International Conference on Trust and Trustworthy Computing*, 2010.

[135] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. ACM SOSP*, 2011.

[136] X. Zhang, Z. Zhou, G. Hasker, A. Perrig, and V. D. Gligor, "TrueNet: Efficient fault localization with small TCB," in *Proc. IEEE International Conference on Network Protocols*, 2011.

[137] 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.

[138] 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.

[139] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated I/O," in *Proc. IEEE Symposium on Security and Privacy*, 2014.