2012 IEEE Symposium on Security and Privacy

Building Verifiable Trusted Path on Commodity x86 Computers

Zongwei Zhou, Virgil D. Gligor, James Newsome, Jonathan M. McCune
ECE Department and CyLab, Carnegie Mellon University

Abstract

A *trusted path* is a protected channel that assures the secrecy and authenticity of data transfers between a user's input/output (I/O) device and a program trusted by that user. We argue that, despite its incontestable necessity, current commodity systems do not support trusted path with any significant assurance. This paper presents a hypervisor-based design that enables a trusted path to bypass an untrusted operating-system, applications, and I/O devices, with a minimal Trusted Computing Base (TCB). We also suggest concrete I/O architectural changes that will simplify future trusted-path system design. Our system enables users to verify the states and configurations of one or more trusted-paths using a simple, secret-less, hand-held device. We implement a simple user-oriented trusted path as a case study.

1 Introduction

A Trusted Path (TP) is a protected channel that assures the secrecy and authenticity of data transfers between a user's input/output (I/O) devices and a program trusted by that user. A 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 [13]; 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 could surreptitiously obtain sensitive user-input data by recording key strokes, modify user commands to corrupt application-program operation, and display unauthentic program output to an unsuspecting user to trigger incorrect user action. This is particularly egregious for embedded real-time systems where an operator would be unable to determine the true state of a remote device and to control it in the presence of a malware-compromised commodity OS [20, 35, 53].

For the past thirty years, only a few systems have implemented trusted paths with limited capabilities on boutique computer systems. These systems employ only a small number of user-oriented I/O devices (e.g., a keyboard, mouse, or video display), and a small number of trusted programs (e.g., login commands [5] and administrative commands [7, 15, 16, 19, 28, 30, 51]). Some instantiations include dedicated operating-system kernels [21, 55]. Given the incontestable necessity of trusted path as a security primitive, why trusted paths have not been implemented on *any commodity computer system using a small-enough Trusted Computing Base (TCB) to allow significant (i.e., formal) security assurance?*

While many operating systems (OSes) offer trusted path in the form of *secure attention sequences*—key-combinations (e.g., Ctrl+Alt+Del) to initiate communication with the OS—the trusted computing base for the end-points of that trusted path is the entire OS, which is large and routinely compromised. Such trusted paths, though users may be forced to trust them in practice, are not adequately *trustworthy*.

Recent research has demonstrated removing the OS from the TCB for small code modules [6, 43, 44, 57]. These mechanisms use a smaller, more trustworthy kernel running with higher privilege than the OS (e.g., as a *hypervisor* or as *System Management Mode (SMM) code*) to provide an isolated execution environment for those code modules. While this work isolates modules that perform pure computation, it does not provide a mechanism that enables isolated modules to communicate with devices without going through the OS, and hence fail to provide a satisfactory trusted-path mechanism.

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 [14, 47]. 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). Devices controlled by a compromised OS may still breach the isolation between device drivers and gain unauthorized access to the registers and memory of other devices (Section 4).

Challenges. Address-space isolation alone is insufficient to remove device drivers from each-others' TCBs, because substantial *shared device-configuration state* exists on commodity computers. A compromised driver in one virtual machine can manipulate that state to compromise the secrecy and authenticity of communication between drivers in other virtual machines and their corresponding devices. For example, a compromised driver can intentionally configure the memory-mapped I/O (MMIO) region of a device to overlap the MMIO region of another device. Such a Manipulated Device (ManD in Figure 1) may then intercept MMIO access to the legitimate trusted-path Device Endpoint (DE in Figure 1). The typical mechanisms protecting CPU-to-memory access or DMA *do not* defend against this "MMIO mapping attack" (Sections 4, 5.2 and 5.3).

DOI 10.1109/SP.2012.42

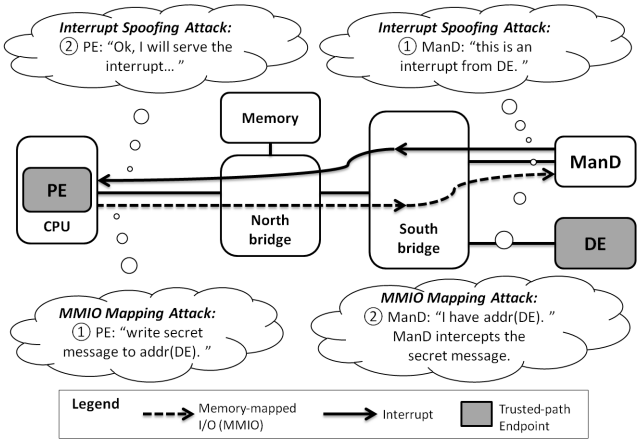

Figure 1: **Attacks against trusted-path isolation.** A manipulated device (ManD) launches an *MMIO mapping attack* (Section 5.2) and an *interrupt spoofing attack* (Section 5.4) against the path between the Program Endpoint (PE) and the Device Endpoint (DE).

Another significant challenge not met by address space isolation is *interrupt spoofing*. Software-configurable interrupts (e.g., Message Signaled Interrupts (MSI) and Inter-processor Interrupts (IPI)) share the same interrupt vector space with hardware interrupts. By modifying the MSI registers of the ManD, a compromised driver may spoof the MSI interrupts of the DE. As shown in Figure 1, the unsuspecting driver in the Program Endpoint (PE) for the DE may consequently perform incorrect or harmful operations by processing spoofed interrupts from the ManD (Sections 4 and 5).

Finally, another unmet challenge is to provide trusted-path mechanisms with verifiable isolation properties on commodity platforms without resorting to external devices that protect and manage cryptographic *secrets*.

Contributions. We show how to protect shared device-configuration state on today's commodity platforms. We use these techniques to build a general-purpose, trustworthy, human-verifiable, trusted path system. It is *general* in that it allows arbitrary program endpoints running on arbitrary OSes to be isolated from their underlying OS and to establish a trusted path with arbitrary unmodified devices. It is *trustworthy* in that the TCB is small—only 16K source lines of code (SLoC) in our prototype—and simple enough to put it within the reach of formal verification [24, 25, 36]. It is *human-verifiable* in that a human using the machine can verify that the desired trusted path is in effect (e.g., that the keyboard is acting as a secure channel to a banking program on that machine). We also propose modifications for the design of x86 platforms that enable simpler, higher performance, and more robust, trusted-path implementations. Finally, we present a case study of a simple trusted-path application that communicates with the keyboard and screen.

2 Problem Definition

This section presents the threat model, desired isolation properties, and assumptions for our trusted-path system.

2.1 Threat 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., 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., modify a device's MMIO region, 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 result of such attacks.

We do *not* consider firmware attacks, physical attacks on devices (see Section 2.3), or side-channel attacks. Denial-of-service attacks are also out of scope; we seek only to guarantee the secrecy and authenticity of the trusted path.

2.2 Desired Trusted-Path Isolation Properties

A Trusted Path contains three components: the *program endpoint* (PE), the *device endpoint* (DE), and the *communication path* between these two endpoints. The communication path represents all hardware (e.g., northbridge and southbridge chips in Figure 1) between the device endpoint and the system resources that support the execution of the program endpoint (CPU and memory). The I/O data (e.g., keyboard scan code, data written to a hard drive), commands (e.g., DMA write requests), and interrupts exchanged between the two endpoints are physically transferred along this path. Co-existing with the commodity OS and its applications, our trusted-path system must *isolate* these components from the compromised OS and manipulated devices. Specifically, we seek to meet the following isolation requirements.

Program Endpoint (PE) Isolation. A compromised OS and manipulated devices cannot interfere with the execution of the PE, and cannot reveal or tamper with any run-time data generated by the program endpoint of the trusted path.

Device Endpoint (DE) Isolation. The I/O data and commands transferred to/from the DE cannot be modified by, or revealed to, the compromised OS and manipulated devices. Interrupts generated by the DE must be delivered exclusively to the PE. Spoofed interrupts generated by the compromised OS or manipulated devices must not interfere with the PE.

Communication Path Isolation. All hardware along the communication path is treated in the same manner as a device endpoint. Thus, communication-path isolation is implemented by applying the same mechanisms that assure device endpoint isolation for all of the hardware devices along the communication path.

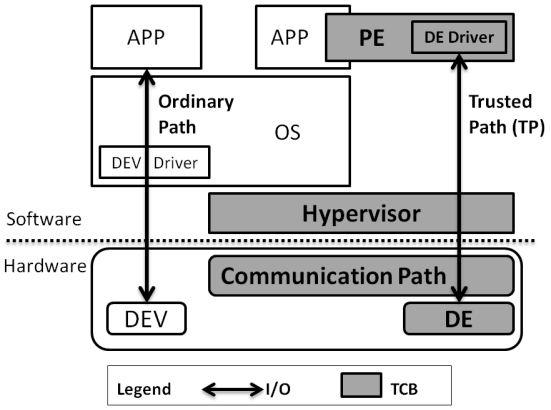

Figure 2: **Trusted path system architecture.** The ordinary path represents I/O transfers outside the trusted path. The shaded area denotes the trusted computing base (TCB) of the trusted path.

2.3 Assumptions

To setup a trusted path to a device, we must obtain accurate information about the chipset hardware (e.g., northbridge and southbridge in Figure 1) and how it is connected to the system. The necessary chipset hardware information includes chipset identifiers, internal register and memory layout and usage, connectivity and hierarchic location (e.g., how the chipset hardware is hard-wired together), and I/O port and memory mappings. Typically, this information is acquired from the system firmware (e.g., BIOS). For the purposes of this paper, we assume that the system firmware is trusted and provides us with this information. In principle, it is possible to validate this assumption if evidence of trustworthy configuration becomes available; e.g., configuration attestation provided by system mechanisms [41, 52], or by a trusted system integrator.

We also assume that all chipset hardware and I/O peripheral devices are *not malicious* in the sense that their hardware and firmware do not contain Trojan-Horse circuits or microcode that would violate the trusted-path isolation in response to an adversary's surreptitious commands. Instead, we assume that devices operate exactly following their specifications and do not perform unintended operations; e.g., intercept bus traffic that is not destinated to them, remain awake when receiving a "sleep" command, or write data to a memory address that is not specified in DMA commands. Such attacks are outside the scope of the present work.

3 System Overview

Our trusted-path system comprises four components: the program endpoint (PE), the device endpoint(s) (DE), the communication-path, and a hypervisor (HV). Figure 2 illustrates the architecture of our system and the trusted-path isolation from the untrusted OS, applications, and devices.

The trusted-path hypervisor HV is a small, dedicated hypervisor that runs directly on commodity hardware. Unlike a full-featured hypervisor (e.g., VMware Workstation, Xen [8]), the HV supports a single guest OS, and does not provide full virtualization [8] of all devices outside the trusted-path to the guest OS. Instead, the OS can directly operate on the devices outside the trusted-path without the involvement of the HV. For example, the leftmost application APP in Figure 2 can access the device DEV via ordinary OS support. Section 3.1 discusses our hypervisor design decisions in depth. The HV provides the necessary mechanisms to ensure isolation between program endpoints, device endpoints, and communication paths for trusted paths. In particular, the HV isolates trusted-path device state from the "shared device-configuration state" on the commodity platform (Section 4). The program endpoint PE of a trusted path includes the device drivers for DEs that are associated with that trusted path. In Section 3.2, we describe this "DE driver-in-PE" design in more detail.

3.1 Trusted-Path Hypervisor

From a whole-system perspective, one can think of our trusted-path hypervisor HV as a micro-kernel that runs at a higher privilege level than the commodity OS. As a starting point, rather than attempting to isolate every driver from each-other, which would require a huge engineering effort, we run a commodity OS as a process on top of our hypervisor, and allow that process to manage most of the devices most of the time, using the existing drivers in the commodity OS. A trusted-path program endpoint runs as a distinct isolated process (VM) directly on the hypervisor. We isolate only the relevant driver(s) and integrate them with the PE of the trusted path, as illustrated in Figure 2.

A valid design alternative would be to discard the hypervisor and instead restructure an OS to be natively microkernel-based. While this alternative may reduce total system complexity, it would explicitly run counter to our stated goal of building trusted path on *commodity* platforms, compatible with *commodity OSes*. The complexities of such a restructuring job for a commodity OS, both from a technical and business perspective, are immense. We are not aware of any successful attempt at restructuring a commodity OS to become natively micro-kernel based for the past three decades.

From an assurance perspective, our overriding goal is to build a hypervisor that is *small* and *simple* enough to enable formal verification. A small codebase is a necessary but insufficient condition for formal verification. Code-size limitations arise from the practical constraints of state-of-the-art assurance methods. To date, even the seemingly simple property of address-space separation, which is necessary but insufficient for trusted path isolation, has been formally proved only for very small codebases; i.e., fewer than 10K SLoC [24]. Simplicity of the codebase is another necessary but insufficient condition for formal verification. Our hypervisor's complexity is demonstrably lower than that of the formally verified seL4 microkernel [36]. Specifically, seL4 implements more com-

plex 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). In fact, the formal verification of address-space separation of ShadowVisor code (a shadow-page-table version of TrustVisor [43]) has already been achieved [24].

Our trusted path is *user-verifiable* since it allows a human to launch the hypervisor and PE on a local computer system and verify their correct configuration and state. We illustrate in Section 8 how to securely perform trusted-path verification for one or more trusted paths, using a simple handheld device that stores no secrets to verify attestations [43] and to signal the user that a trusted channel is in place.

3.2 Program Endpoint

Our trusted path design calls for the implementation of the device drivers of the DEs within the program endpoint for three assurance reasons. *First*, our goal is to produce a small and simple hypervisor, 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 beyond the point where significant assurance could be obtained. *Second*, placing the DE's driver within a program endpoint is a natural choice: DE driver isolation can leverage all the mechanisms that protect the PE code and data from external attacks. *Third*, trusted-path device endpoints are dedicated devices for a specific application and/or user interface. Consequently, the DE device drivers are typically simpler than their shared-device versions. That is, program endpoints have the freedom to customize the DE driver for their specific needs (e.g., some PEs clearly do not need full-fledged drivers, as illustrated in Section 9). In particular, they can tailor the driver's functions to those strictly necessary and minimize its codebase to obtain higher assurance of correct operation.

The alternative of placing a DE 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 PE).

However, this alternative would have at least two security disadvantages. *First*, an additional protected channel would become necessary between the isolated DE driver and separately-isolated PE, and an additional protection boundary would have to be crossed and checked—not just the one between the hypervisor HV and PE. *Second*, driver isolation in separate user or system space would require extra mechanisms in addition to those for PE isolation. For example, an additional protection mechanism would become necessary to control the access of application PEs to isolated drivers in user space. Furthermore, serious re-engineering of a commodity OS/hypervisor would become necessary [14, 36], which would run against our stated goals. In balance, we picked the "DE driver-in-PE" model since security and ease of commodity platform integration have been our overriding concerns.

The key challenge for developing a program endpoint is to isolate the DE driver from the untrusted OS. Since DE drivers *cannot* rely on the OS Application Program Interfaces (API) for I/O services, they must be modified from the commodity device driver to eliminate API dependencies. In Section 7, we analyze this design and offer guidelines for device driver development for our trusted-path system.

4 Device-Isolation Challenges

As suggested in the introduction, both device-driver [14, 47] and program isolation [6, 43, 44, 57] are insufficient for trusted-path protection from a compromised OS. The fundamental reason is that, aside from the address space containing the device driver and program endpoint, there is still substantial shared device-configuration state on the commodity platform. Protecting individual device configurations within the "shared device-configuration state" is necessary to provide device isolation for a trusted path. We identify three categories of "shared state" on current commodity platforms, and propose corresponding protection mechanisms for our hypervisor design.

I/O Port Space. All devices on commodity x86 platforms share the same I/O port space. The I/O port assigned to a particular device can be dynamically configured by system software. If that software is a compromised OS, the I/O port(s) of one device can be intentionally configured to conflict with those of other devices. Thus, unmonitored I/O port reconfiguration of any device on the platform may breach the I/O port access isolation of a device endpoint. We present isolation mechanisms for device I/O port access in Section 5.1.

Physical Memory Space. Devices' MMIO memory regions share the same physical address space. We present a new attack—the *MMIO mapping attack*—which breaches device memory isolation. This attack *cannot* be solved by *any* current mechanism for preventing unauthorized CPU access to memory (e.g., AMD Nested Page Table (NPT) [3]) or for preventing unauthorized DMA (e.g., Intel VT-D [34]). No existing trusted-path solutions (e.g., [11, 22, 56]) prevent this attack.

In the MMIO mapping attack, a compromised OS intentionally maps the MMIO memory of a manipulated device such that it overlaps the MMIO or DMA memory region of a DE. As a result, the data in DE memory becomes exposed to the manipulated device, and hence the compromised OS. 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 (where the graphics card is serving as the DE). Hence, the display output may be directly sent to a remote adversary via the network. We present our solution to prevent this attack in Section 5.2, and also propose some architectural changes that can help simplify our solution considerably (Section 6.3).

Interrupt Space. Software-configurable interrupts (e.g., MSIs, IPIs) share the same interrupt vector space with hardware interrupts. For example, a compromised OS can send out any spoofed interrupt to any CPU by writing the proper value to one register in the Local Advanced Programmable Interrupt Controller (LAPIC) [3]. By modifying a device's MSI registers, a compromised OS can manipulate that device to send out spoofed MSI interrupts to any CPU on the platform. The device endpoint isolation is violated, and an unsuspecting device driver may consequently perform incorrect or harmful operations when receiving spoofed interrupts. In Section 5.4, we discuss the interrupt spoofing attack in detail and present our solutions to prevent it from breaching trusted-path isolation. We further suggest concrete I/O architectural changes to improve our defense mechanisms in Sections 6.1 and 6.4.

For completeness, we also discuss two recently-reported I/O attacks that are caused by DMA request ambiguity [48] and by unmediated peer-to-peer device communication [49, 59, 60] (Section 6). We provide defense mechanisms against these attacks, and suggest potential I/O architectural changes that help simplify our mechanisms.

5 Hypervisor Design

This section illustrates the detailed hypervisor mechanisms which isolate the trusted-path device state within the shared device-configuration state. We start with the protection of I/O-port and device-memory access (Sections 5.1 and 5.2). Then we describe the fundamental building block of device I/O isolation, namely, the protection of the device configuration space in Section 5.3. Section 5.4 shows how we leverage the device I/O isolation to design device interrupt isolation mechanisms.

5.1 Protection of I/O-Port Access

Software programs use the IN/OUT family of CPU instructions to exchange data with devices' I/O ports. To control access to device I/O ports, the hypervisor HV must prevent device-port-mapping conflicts that may be intentionally created by the compromised OS, and confine the I/O port access from both the trusted-path program endpoint and the compromised OS.

Preventing Port-mapping Conflicts. The compromised OS can re-map a manipulated device's I/O ports to overlap those of the DE. Read/write accesses to those I/O ports have unpredictable results, since *all* devices that have port overlaps with the DE will respond to the I/O access. Thus, a manipulated device (and a compromised OS) can potentially obtain secret data from the trusted-path, or corrupt the execution of the PE.

To address this problem, the HV should isolate the DE's I/O ports from the shared I/O port space of the platform. Specifically, before executing the PE, the HV scans through all I/O port mappings relevant to the chipset hardware (as mentioned in Section 2.3) and enumerates all plug-and-play (PnP) devices to detect their configured I/O ports. For example, the HV accesses the PCI configuration space of all PCI devices in the system, and parses their I/O port settings via the PCI Base Address Registers in the configuration space. If any of the above port settings conflict with those of the DE, the HV issues an exception to the PE. The HV must protect all I/O port mappings in the device configuration space from modification by a compromised OS or manipulated devices, throughout the PE run-time. We defer the details of scanning and protecting device configuration space to Section 5.3.

Confining I/O-port Access. The HV should confine the PE so that it can only access the I/O ports of its associated DE (s). Specifically, the HV intercepts and filters out the port access requests by the PE to unassociated I/O ports. This is accomplished by configuring the I/O port-access-interception bitmap in the hypervisor control block that describes the PE's execution environment (a standard feature of x86 hardware virtualization support [3, 32]).[1] Similarly, the HV should also filter out access to the DE's ports from the OS running concurrently on different CPUs, by configuring the I/O port-access-interception bitmap for the OS's execution environment.

5.2 Protection of Device-Memory Access

There are two methods for the PE to interact with the DE 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 DE-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 an *MMIO mapping attack* on the DE's associated MMIO and DMA memory regions (recall Section 4), as shown in the left half of Figure 3. To defend against this attack, the hypervisor HV must ensure that all MMIO memory ranges used by the chipset hardware and peripheral devices outside the PE-DE trusted path are non-overlapping with those of the DE.

Before executing a PE, the HV scans through all MMIO memory mappings specified by the chipset hardware, and enumerates all PnP devices to discover their MMIO memory ranges (e.g., check the PCI Base Address Registers in the PCI configuration space). If overlaps with the DE's memory ranges exist, an MMIO mapping attack may be in progress, and the device isolation property of the trusted path may be violated. Upon detection, the HV issues an exception to the PE. During the PE's execution, the HV must prevent the compromised OS and manipulated devices from modifying the MMIO memory mappings of all devices, by protecting the device configuration space. We elaborate on our mechanisms for protecting device configurations in Section 5.3.

Preventing Unauthorized Memory Access. The compromised OS can directly access the DE's MMIO and DMA memory regions, and can manipulate a device outside the trusted path to issue unauthorized DMA requests to access those re-

[1] Section 9.1 presents an optimization mechanism when the PE executes at user privilege level (CPU Ring 3), which by default will not have sufficient privilege to access I/O ports.

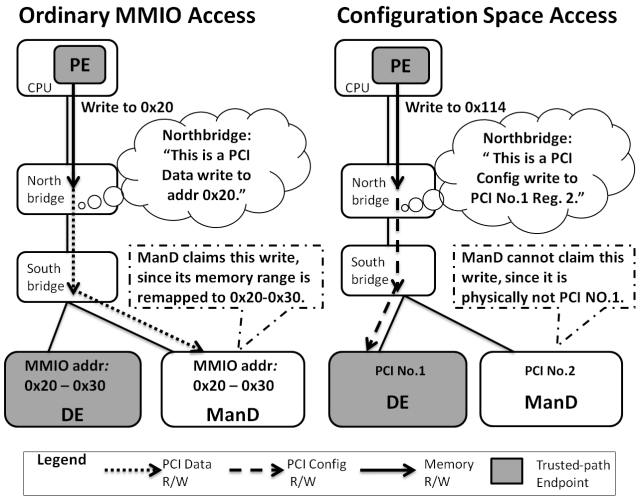

Figure 3: **MMIO mapping attack against the trusted path.** In the left half of the figure, ManD's MMIO memory is remapped to overlap that of the trusted-path DE (0x20-0x30), and the MMIO mapping attack will succeed. The right half shows that MMIO mapping attacks cannot compromise the access to PCI/PCIe configuration space.

gions. The HV can defend against these attacks by leveraging standard features for x86 hardware virtualization support. For example, the HV configures the access permissions in Nested Page Tables (or Extended Page Tables) [3, 32] to prevent unauthorized CPU-to-memory access. The HV also sets up the IOMMU [2, 34] to protect the DE-associated memory regions from other devices' DMA buffers. 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 the trusted path in Section 6.1.

5.3 Protection of Device Configuration Space

A fundamental building block of our prevention mechanisms against I/O port conflicts (Section 5.1) and MMIO mapping attacks (Section 5.2) is protecting the device configuration space. Specifically, the hypervisor intercepts all accesses to the device configuration space throughout the trusted-path session, including trusted-path establishment, run-time, and tear-down. The hypervisor grants the program endpoint *only* the access permissions to its device endpoints' configuration space, 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 I/O ports [10, 54], or through reserved MMIO memory regions [10]. 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 3). 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 [23]. 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 trusted-path establishment, the hypervisor only needs to configure the I/O port-access-interception bitmap (Section 5.1), Nested/Extended Page Tables, and IOMMU (Section 5.2), to prevent unauthorized CPU-to-memory access and DMA to the whole device configuration space. After that, the HV can securely enumerate all devices. Protection of the device configuration space remains active until the trusted path is torn down.

5.4 Isolation of Device Interrupts

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). In a trusted-path session, the hypervisor should fulfill the following two requirements for device interrupt isolation: (1) Interrupts should be correctly routed, i.e., interrupts from the DE are exclusively delivered to the respective PE, and other interrupts should not arrive at the program endpoint.[2] (2) Spoofed interrupts should not compromise the trusted-path.

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 PE receives an interrupt that appears to originate from the DE, it communicates with the DE to check whether the DE indeed has a pending interrupt. If not, the PE refuses to service this interrupt. However, *not* all DE device drivers are robust against spoofed interrupts. For

[2] An exception exists when the PE receives interrupts from the devices that physically share the same interrupt pin on the [A]PIC with the DE. To cope with the shared interrupts, the HV provides a specific hypercall interface to the PE for forwarding the shared interrupts to the OS on other CPUs. In a uniprocessor system, devices that share interrupts with the DE should be put into a sleep or pending mode before invoking the PE. Otherwise, interrupts from those devices are dropped during the trusted-path session, and those devices are not guaranteed to perform consistently across the trusted-path session.

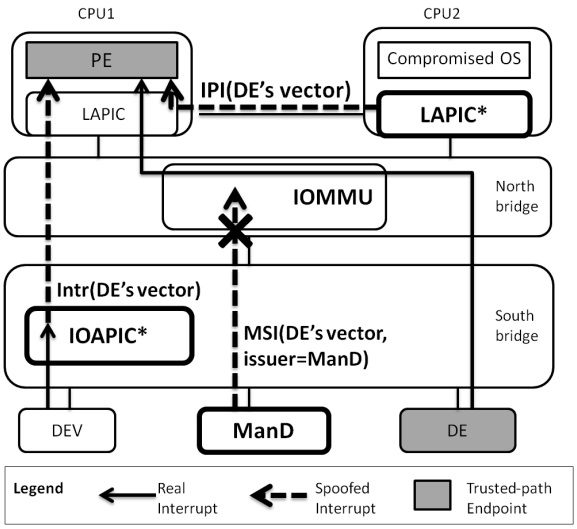

Figure 4: **Interrupt spoofing attacks against the trusted-path.** `IOAPIC*/LAPIC*` denotes the interrupt controllers manipulated by the compromised OS. `Intr(DE's vector)` represents spoofed hardware interrupts with the DE's interrupt vector. When the IOMMU interrupt remapping feature is enabled, spoofed MSIs with incorrect issuer identifiers will be filtered out by the IOMMU (Section 5.4.2).

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, our trusted-path system must modify the configurations of the interrupt controllers, MSI-capable devices, and other chipset hardware along the interrupt delivery route during trusted-path establishment. The compromised OS may subvert those configurations during the execution of the trusted-path program endpoint, in order to mis-route device endpoint interrupts or launch interrupt spoofing attacks (Figure 4). Thus, our trusted-path hypervisor should protect those configurations throughout the entire trusted-path session. We now detail our protection mechanisms for all three types of interrupts.

5.4.1 Isolating Hardware Interrupts

Hardware interrupts are managed by a PIC on uni-processor systems, and by an I/O APIC 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). HV isolates trusted-path device interrupts as follows during trusted-path establishment:

- Modify the redirection table to reroute DE interrupts, and to remove any interrupt-to-vector mapping conflicts between the DE and other devices.
- Setup corresponding PIC/LAPIC registers to enable delivery of the DE's interrupts, and to temporarily disable interrupts from other devices.
- Manipulate the OS's Interrupt Descriptor Table (IDT) so that the DE interrupts will trigger their corresponding interrupt handlers in the PE.

While the PE is running, the hypervisor provides run-time protections to the redirection tables, the interrupt controller registers, and the IDT (using the mechanisms described in Sections 5.1- 5.3). 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 PE.

5.4.2 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) [10, 54].

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 PE-DE trusted-path, a compromised OS can change the Message Data Registers of other devices to include the DE'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 [65].

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 does not defend against the above "scatter-gather attack" [65].

Solution. We design a comprehensive and efficient solution for isolating MSIs, which does not require controlling any MSI-capable devices other than the DE. Our solution leverages the Interrupt Remapping features in the IOMMU [2, 34]. With Intel VT-D Interrupt Remapping, MSI messages are embedded with a specified handle [34]. 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 DE, the hypervisor HV modifies the DE'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 PE). The HV 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 suppress or mis-transform MSI signals. Thus, the HV 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 trusted-path hypervisor configures the corresponding interrupt remapping table entry to only accept MSI messages with the DE's device identifier. As shown in Figure 4, spoofed interrupts generated by manipulated devices do not have the interrupt identifier of the DE, 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 6.1.

Throughout the trusted-path session, the HV protects all the above registers and tables using the mechanisms described in Sections 5.1 - 5.3.

5.4.3 Isolating Inter-Processor Interrupts

During a trusted-path session, CPUs that run the OS can leverage Inter-Processor Interrupts (IPIs) to forward unserviced hardware interrupts to the trusted-path program endpoint (Figure 4). The OS issues those IPIs by writing special data to the Interrupt Command Register (ICR) of the LAPIC. The data carries information including the interrupt type, vector number, and delivery destination.

Challenges. Because the DE's interrupts are exclusively routed to the associated program endpoint, and that program endpoint only handles interrupts from its device endpoints, any unserviced device interrupts forwarded from other CPUs to the PE should be treated as spoofed interrupts.

Defending against these spoofed interrupts requires care. First, when delivering interrupts to the CPU, the LAPIC does not distinguish between IPIs and interrupts directly from devices. IPIs with the DE's interrupt vector number are always delivered to the PE. Second, the IOMMU is not used for intercepting IPI messages, and is often not on the path between two LAPICs. Third, memory protection mechanisms (Section 5.2) of the hypervisor HV do not work. When access to the ICR is trapped into the HV via memory access violations, the contents of the memory write are not reported to the HV [3, 32]. The HV cannot determine what IPIs are sent without knowing the value written to the ICR. Note that the HV cannot blindly block all IPIs, since some of them are for important system management purposes such as cache coherency.

Solution. To prevent spoofed IPIs, our trusted-path hypervisor employs a mechanism to control the LAPICs by enabling the LAPIC x2APIC mode. In x2APIC mode, LAPIC registers are accessed via Model Specific Registers (MSR) access instructions, which are privileged instructions that can be intercepted by the hypervisor. Fortunately, an MSR access violation does report to the hypervisor the value being written that triggered the violation. Therefore, during the trusted-path session, the hypervisor HV intercepts all data writes to the ICRs of all other CPUs that run the OS, and blocks *only* the data writes that trigger spoofed IPIs to the PE. This interception remains active until the trusted-path is torn down. In Section 6.4, we also propose some architecture modifications that would help simplify our protection mechanism here.

6 I/O Architectural Suggestions

We make suggestions for changes to the commodity x86 I/O architecture that would significantly simplify the design of our trusted-path solution.

6.1 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 trusted-path device endpoint [48].

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 HV identifies all devices behind the same PCI/PCI-to-PCIe bridges that connect the DE by enumerating the PCI configuration space. Before executing the PE, these devices are put into a quiescent state (e.g., sleep, or a pending state). The HV can verify the devices' quiescent state by reading device-specific status registers before approving the execution of the PE. During the PE's execution, the HV prevents the compromised OS from waking the pending devices by interposing on the relevant I/O ports and memory ranges (Sections 5.1 and 5.2).

However, quiescing all devices sharing the same PCI/PCI-to-PCIe bridge with a trusted-path DE reduces I/O performance. During the execution of the trusted path PE, an OS cannot communicate with any of those devices. To eliminate this uncomfortable trade-off between trusted-path security and performance, we suggest several potential architectural changes. First, motherboard manufacturers can configure a system that supports trusted path by assigning 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.

6.2 Unmonitored Peer-Device Communication

Manipulated PCI/PCIe and USB 2.0+ devices may establish peer-to-peer connections with a trusted-path device endpoint, bypassing *all* isolation mechanisms implemented by the hypervisor [49, 59, 60]. PCI/PCIe peer-to-peer communication

complies with the PCI/PCIe specifications [10, 54], and thus cannot be denied by the device itself. The hypervisor's 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) [4]. 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 trusted-path hypervisor configures the ACS on all corresponding bridges to prevent any peer-to-peer communication between the DE and other devices. The hypervisor also protects the ACS configuration using the mechanisms described in Section 5. 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 [61] is easier, because the communication only succeeds when both communicating devices enable OTG and comply with OTG protocols. Thus, the HV or PE can explicitly configure the DE to disable USB OTG.

6.3 MMIO Memory Access Control

As mentioned in Section 5.2, there is no central controller/chipset hardware that can explicitly control access to the mapping between devices and their MMIO memory regions, without involving the on-CPU software. As one good example, the IOMMU provides controls on the mapping between devices and their DMA memory regions. We suggest similar protection mechanisms be implemented within the memory management unit (MMU). This would enable trusted code (i.e., the HV) to explicitly assign MMIO memory regions to devices, or to a group of devices, based on a specific access control policy. This I/O architectural change would help simplify our countermeasures against the MMIO mapping attack.

6.4 Memory Virtualization Support

Memory virtualization support in mainstream CPUs (e.g., AMD NPT [3], Intel EPT [32]) only delivers memory address and access type information to the hypervisor when a memory access violation occurs. Thus, the hypervisor cannot easily know what data was being written to the corresponding memory address. This limitation affects the virtualization of all devices that rely on MMIO access. We propose an improvement of the memory virtualization support. The memory violation should be delivered to the hypervisor with a pointer to the contents that were being written to the memory region. This improvement will help to simplify the hypervisor design for virtualizing MMIO-capable devices. It will also help simplify device design, since no special mode for virtualization (e.g., x2APIC mode of LAPIC mentioned in Section 5.4.3) is needed.

7 Program Endpoint (PE) Driver Design

We discuss design options for trusted-path device drivers in PEs, and provide some high-level guidelines for driver implementation under different CPU privilege levels.

Driver Design. To isolate the trusted-path device driver from the OS, we need to modify the commodity device driver to eliminate any dependencies on the commodity OS or OS kernel. Such driver porting efforts are manageable for the following four reasons. First, previous research on device driver implementation in user space [12, 40] and device driver isolation [18, 26, 58] shows how to extract the drivers from the OS kernel in a manner that reduces reliance on the OS for I/O services. Second, research on driver code characteristics shows that most of the commodity driver code is for housekeeping purposes, such as resource allocation, clean-up, and error handling, with only a small portion of the code dealing with the actual device I/O [26]. The modifications for implementing the DE device driver inside a PE might not propagate to the entire codebase of the commodity driver. Third, our trusted-path hypervisor already implements critical I/O services needed by the DE drivers (such as interrupt-controller configuration, device I/O port and MMIO settings, and DMA memory management), which are quite helpful in driver implementation. Fourth, trusted-path device drivers need not be full-featured device drivers as in the commodity OS. Instead, trusted-path device drivers may only need to support a minimal set of functions that meet the I/O needs of the associated program endpoint. For example, to implement secure display, the graphics-controller driver in the PE might only support a subset of all possible graphics-card modes.

Guidelines. The detailed implementation of a trusted-path device driver is closely related to the features of its trusted-path device and the needs of the program endpoint. These differ significantly across trusted-path applications. We introduce general guidelines for designing I/O related portions of the DE driver in our trusted-path system.

To serve the different functional and performance needs of trusted-path applications, DE device drivers can be implemented either in user privilege level (Ring 3), or kernel privilege level (Ring 0). We recommend implementing device drivers in Ring 3 for security-sensitive trusted-path applications, where performance is not the major concern of the application. In the Ring 3 case, the driver itself cannot directly execute I/O operations associated with privileged CPU instructions: e.g., IN/OUT for I/O port access, RDMSR/WRMSR for accessing MSRs, and IRET for returning from interrupt service routines. Our trusted-path system should *not* temporarily elevate the privilege level of the device endpoint drivers and the program endpoint to allow executing these privileged instructions. Instead, the DE driver executes these instructions with the involvement of the HV, via certain hypercall interfaces. In addition, device drivers in Ring 3 are often operating in virtual memory space and lack the view of physical-to-virtual memory

mappings. In cases where the drivers need to perform operations directly on physical memory addresses (e.g., to manipulate devices' MMIO registers), the driver needs the involvement of the HV to provide the corresponding physical addresses. The frequent involvement of the hypervisor introduces a performance penalty to the device driver. Our case study (Section 9) illustrates our experiences in minimizing the involvement of the HV to enable high performance driver operations.

In the Ring 0 case, the device endpoint driver can execute privileged CPU instructions, which may give the driver access to critical system resources such as I/O ports and MSR registers. The key challenge is for the trusted-path hypervisor to confine the capabilities of the DE drivers, so that the drivers can *not* abuse their capabilities to compromise the rest of the system, including the OS, applications, and other devices. In addition, the unsuspecting abuse of privileged instructions for access to system resources may incur a performance penalty by triggering hypervisor involvement or driver misbehavior.

8 User Verification of Trusted-Path State

Our design enables verification of the trusted-path 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 10). First, the standard remote attestation protocol is identical for different trusted-path configurations, and thus a general-purpose verifier suffices to work for all trusted-path 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 PE-to-DE trusted-path state [37]. Moreover, our design can also support multiple trusted-paths on a platform using just one simple device.

Note that all user-verification of the trusted-path 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.[3]

Trusted-Path Verification Protocol. We describe a simple protocol for *user verification* of the trusted-path 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 trusted-path program endpoint requests a TPM Quote containing cryptographic hashes of the code and static initialized data of the hypervisor and the program endpoint that are digitally signed using a TPM-based key. The program endpoint returns the signed quote to the untrusted application, which sends the quote to the hand-held verifier. The verifier checks the validity of the signature and cryptographic hashes taken over the HV and the PE, 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 hypervisor and program endpoint are running on the host platform, and the PE-to-DE trusted path has been established. If the red light comes on, the security properties of the trusted-path are not guaranteed.

Supporting Multiple Trusted Paths. If activation of multiple trusted paths to different program endpoints is desired from the same hand-held device, we envision that a single trusted path to a trusted shell [30] can first be executed on the target platform. This trusted shell, together with the trusted-path and the underlying hypervisor, can be verified by the user as explained above. All other isolated PEs can then be registered via the underlying hypervisor using trusted shell commands. A user can also identify, select, invoke, manage, monitor, and tear down any desired PE via the trusted shell. Because the trusted path for input to and output from the trusted shell has already been verified by the user using a hand-held device, there is no need to verify any subsequent trusted paths.

9 Case Study: A Simple User-Oriented Trusted Path

We implement a user-oriented trusted-path and evaluate its performance to illustrate the feasibility of our trusted-path design. This trusted path application protects a user's keyboard input sent to an application, and the output from the application to the computer's display, against attacks launched from the compromised OS or 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.

9.1 Hypervisor Implementation

We implement our hypervisor by extending a multi-core version of TrustVisor [43]. Our extension includes configuration access protection, device I/O ports, MMIO and DMA memory protection, and interrupt redirection and protections. Our DEs are a PS/2 keyboard and a VGA-capable integrated graphics controller. Specifically, the HV protects the device config-

[3] To obtain malware-independent verification of the trusted-path state we must detect the effect of the *Cuckoo attack* [46], 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.

Table 1: **A comparison of hypervisor codebases.**

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

uration space (Section 5.3), and then securely enumerates all devices. The HV also sets up the IOAPIC and LAPIC to deliver the keyboard interrupt to the CPU that runs the PE (Section 5.4), and protects the IOAPIC and LAPIC configuration using the mechanisms described in Sections 5.1 and 5.2. In addition, the HV 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 5.4).

Small TCB. We use the sloccount[4] program to count the number of lines of source code in TrustVisor and our hypervisor HV. As shown in Table 1, our implementation of HV 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 5.3), 450 lines are for the interrupt protection mechanisms in Section 5.4, and 300 lines are for the I/O port and memory protection mechanisms in Sections 5.1 and 5.2. Our software TCB for the hypervisor (not including the source code for debug purposes) is about 15,500 lines of code in total.

9.2 Program Endpoint Implementation

The PE 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 PE runs in CPU Ring 3. This unprivileged setting allows for more efficient isolation mechanisms between the PE and the rest of the system. That is, instead of trapping every port access from the PE (Section 5.1), the HV simply configures the OS's I/O permission bitmap in the Task State Segment to confine the PE's access to only the DE's I/O ports.

However, running the PE in Ring 3 makes DE driver porting more difficult. First, some sensitive I/O instructions (e.g., IN, OUT) and some critical device-driver instructions (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 PE 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 HV for the above operations, while still maintaining the isolation of the PE 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 PE reserves a region in its virtual memory space and then makes a hypercall to the HV. The HV re-maps the reserved PE memory pages to the VGA display memory region. After this hypercall, the PE 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 DE interrupt handler finishes execution and returns control back to the PE, the interrupt handler should run a return-from-interrupt (IRET) instruction. The HV provides a hypercall that emulates this IRET instruction.

[4] http://www.dwheeler.com/sloccount/

Table 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%

9.3 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 PE implementation can achieve good performance by minimizing invocations of the HV.

Trusted-path Setup and Tear-down. To measure the HV overhead for trusted-path establishment, we compared the time required for the creation of a PE's isolated environment and a trusted path between a PE and a DE with the time required to create only a PE's isolated environment using TrustVisor. As shown in Table 2, TrustVisor took about 1.752 milliseconds to create the isolated environment, while our HV 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.

We also measured the HV overhead incurred in trusted-path tear-down after the PE completes all of its operations. In 10,000 trials, TrustVisor tore down the isolated environment of the PE in approximately 0.436 milliseconds, while HV 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.

Table 3: **DE 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%

Device Driver Performance. We measure the HV 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 3 illustrate that our optimized implementation of user-level DE drivers can achieve good performance by minimizing the frequency of HV invocations for operations that require system-level privileges. Our optimized PE implementation took only 18 microseconds to perform INB and 19 microseconds for OUTB. In contrast, invoking the HV and performing a same operation would take around 40 microseconds. PE writes to VGA display memory take approximately 15 microseconds, but it would take more than 39 microseconds to invoke the HV to perform the same operation. This implies that a context switch between the trusted-path program endpoint and the hypervisor takes roughly 23 microseconds.

10 Related Work

We first compare our trusted-path system with closely related work; i.e., the Uni-directional Trusted Path (UTP) [22] system, the DriverGuard system [11], and hypervisors with a structured root domain [14, 47]. Then we review other trusted-path proposals. None of the related work achieves all of our trusted-path isolation and TCB reduction properties (Table 4), for the following three reasons. (1) Most proposals rely on large TCBs and sacrifice assurance. (2) Some proposals employ cryptographic channels and require special devices and key management functions. These solutions are often impractical and have fundamental usability issues. (3) Device virtualization-based solutions often fail to provide device isolation and/or program endpoint isolation.

10.1 Closely Related Work

The UTP system [22] 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 [11] protects the confidentiality of the I/O flows between commodity peripheral devices and some Privileged Code Blocks (PCBs) in device driver code. Our system protects *both* the confidentiality and integrity of the I/O data. 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.

Hypervisors with structured root domains can assign different device drivers to separate virtual machines (VMs) and securely associate them with application VMs [14, 47]. These hypervisors isolate the I/O ports and the memory address-space of a device driver domain from other domains. However, their device driver VM isolation mechanisms are incomplete. A single malicious VM driver may still exploit the device-isolation inadequacies of commodity I/O hardware discussed above (e.g., MMIO mapping attack, MSI spoofing, IPI spoofing, and DMA request ambiguity) to compromise other device VMs. Our solutions defend against *all* of these attacks. Moreover, a program endpoint in an application VM typically communicates with the device VM via a guest OS [14, 47], and the device driver inside a device VM is also not fully isolated from the OS in that device VM [47]. This greatly swells their trusted-path TCBs (Table 4).

10.2 Other Trusted-Path Proposals

Large TCB Requirements. Trusted path on the DirectX system [38] and the Trusted Input Proxy system [9] 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 [29] 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 [50]. This design trusts a full virtual machine monitor, and only protects data integrity. Similarly, Gilbert et al. propose a trustworthy mobile sensing architecture [27] 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).

Cryptographic Channels and Special Devices. Saroiu et al. [50] propose another sensor reading protection system based on the assumption that the reading is digitally signed by a TPM on the sensor (c.f. [17]). The Zone Trusted Information Channel (ZTIC) is a dedicated device with a display, buttons and cryptographic primitives [39, 64]. 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 [45].

Solutions using cryptographic channels and special devices

Table 4: **A comparison of trusted-path components in different architectures.** "Dom0" denotes the monolithic root domain in the "split-driver" model [8]. "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 with dom0**	**Hypervisor with device pass-through**	**Hypervisor with structured dom0**	**Our Solution**
TP Program Endpoint	app	VM(OS+app)	VM(OS+app)	VM(OS+app)	app
TP Device Driver	OS/hypervisor	dom0	OS	VM per device	app
Other TP Components		hypervisor	hypervisor+dom0	hypervisor+Dev VMs	small hypervisor
TCB Size (SLoC)	>10M	>1.2M	>1.2M	>1.2M	≈16K

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?

Device Virtualization. Hypervisors that are based on the "split-driver" model [8] move device management from the hypervisor to a root domain, *dom0*, which is frequently large and unstructured [14]. 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 DE of a trusted path via the *untrusted* guest OS upon which it runs.

Hypervisors with device pass-through support [42] (e.g., Xen, KVM) or para-passthrough support (e.g., BitVisor [56]) 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.

11 Applicability to New I/O Architectures

QuickPath/HyperTransport. Intel's QuickPath Architecture [33] 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 communication-path isolation and device isolation problems for the I/O devices of a trusted path any more than the commodity x86-based I/O architecture. Although we present a trusted-path design for the latter, our design is easily adapted for the former. The changes are only in the composition of the communication path: 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 trusted-path design is equally applicable to other similar I/O architectures, including AMD's HyperTransport [31].

ARM. Recent advances to ARM's TrustZone security extensions [1] and virtualization support [62] make the application of our trusted-path design to ARM-based I/O architectures possible [63]. ARM's TrustZone Security Extensions [1] 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 [62], 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 trusted-path system to the ARM architecture, and supporting a wide range of applications on mobile and embedded platforms, is future work.

12 Conclusion

Building a general-purpose trusted path mechanism for commodity computers with a significant level of assurance requires substantial systems engineering, which has not been completely achieved by prior work. Specifically, it requires (1) effective countermeasures against I/O attacks enabled by inadequate I/O architectures and potentially compromised operating systems; and (2) small trusted codebases that can be integrated with commodity operating systems. The design presented in this paper shows that, in principle, trusted path can be achieved on commodity computers, and suggests that simple I/O architecture changes would simplify trusted-path design considerably.

Acknowledgement

We are grateful to the reviewers, and Kevin Butler in particular, for their insightful suggestions. We also want to thank Adrian Perrig and Amit Vasudevan for stimulating conversations on trusted path.

This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the US Army Research Office, and by the National Science Foundation (NSF) under grants CNS083142 and CNS105224. The views and conclusions contained in this document are solely those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

References

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

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

[3] AMD. AMD 64 Architecture Programmer's Manual: Volume 2: System Programming. Pub. no. 24593 rev. 3.20, 2011.

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

[5] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: a study of VM/370 integrity. *IBM System Journal*, 15(1):102–116, 1976.

[6] 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.

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

[8] 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.

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

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

[11] Y. Cheng, X. Ding, and R. H. Deng. DriverGuard: A fine-grained protection on I/O flows. In *Proc. European Symposium on Research in Computer Security*, 2011.

[12] D. Clark. *An Input/Output Architecture for Virtual Memory Computer Systems*. PhD thesis, MIT, 1974.

[13] D. D. Clark and M. S. Blumenthal. The end-to-end argument and application design: the role of trust. *Federal Communications Law Journal*, 63(2):357–390, 2011.

[14] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: Security and functionality in a commodity hypervisor. In *Proc. ACM Symposium on Operating Systems Principles*, 2011.

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

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

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

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

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

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

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

[22] 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.

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

[24] 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.

[25] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Proc. IEEE Symposium on Security and Privacy*, 2010.

[26] 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.

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

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

[29] 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.

[30] 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.

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

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

[33] Intel. Intel's QuickPath architecture: A new system architecture for unleashing the performance of future generations of Intel

multi-core microprocessors, 2008.

[34] Intel. Intel virtualization technology for directed I/O architecture specification. Intel Pub. no. D51397-005 rev. 1.3, 2011.

[35] 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.

[36] 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.

[37] B. Lampson. Usable security: How to get it. *Communications of the ACM*, 52(11):25–27, 2009.

[38] 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.

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

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

[41] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *Proc. ACM Conference on Computer and Communications Security*, 2011.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

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

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

[52] 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.

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

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

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

[56] 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.

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

[58] 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.

[59] A. Triulzi. Project Maux Mk.II - "I own the NIC, now I want a shell!". In *PacSec/core*, 2008.

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

[61] 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.

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

[63] 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? Technical Report CMU-CyLab-11-023, Carnegie Mellon University, 2011.

[64] 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.

[65] 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.