Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals

A. Theodore Markettos*, Colin Rothwell*, Brett F. Gutstein*†,
Allison Pearce*, Peter G. Neumann‡, Simon W. Moore*, Robert N. M. Watson*
* University of Cambridge, Department of Computer Science and Technology † Rice University ‡ SRI International
www.thunderclap.io theo.markettos@cl.cam.ac.uk

***Abstract*—Direct Memory Access (DMA) attacks have been known for many years: DMA-enabled I/O peripherals have complete access to the state of a computer and can fully compromise it including reading and writing all of system memory. With the popularity of Thunderbolt 3 over USB Type-C and smart internal devices, opportunities for these attacks to be performed casually with only seconds of physical access to a computer have greatly broadened. In response, commodity hardware and operating-system (OS) vendors have incorporated support for Input-Ouptut Memory Management Units (IOMMUs), which impose memory protection on DMA, and are widely believed to protect against DMA attacks. We investigate the state-of-the-art in IOMMU protection across OSes using a novel *I/O-security research platform*, and find that current protections fall short when faced with a functional network peripheral that uses its complex interactions with the OS for ill intent. We describe vulnerabilities in macOS, FreeBSD, and Linux, which notionally utilize IOMMUs to protect against DMA attackers. Windows uses the IOMMU only in limited cases. and it remains vulnerable. Using Thunderclap, an open-source FPGA research platform that we built, we explore new classes of OS vulnerability arising from inadequate use of the IOMMU. The complex vulnerability space for IOMMU-exposed shared memory available to DMA-enabled peripherals allows attackers to extract private data (sniffing cleartext VPN traffic) and hijack kernel control flow (launching a root shell) in seconds using devices such as USB-C projectors or power adapters. We have worked closely with OS vendors to remedy these vulnerability classes, and they have now shipped substantial feature improvements and mitigations as a result of our work.**

I. Introduction

Modern computers are a complex distributed system of interlocking hardware/software components, even inside the case. Direct Memory Access (DMA) allows programmable peripheral devices – storage adapters, network adapters, USB controllers, GPUs, and other accelerators – to access system memory in order to improve performance. Historically, DMA has been available only within the physical case of a computer – e.g., PCI Express (PCIe) or on-chip interconnect. More recently, DMA has been available via connections for external devices – Firewire, and latterly Thunderbolt 2, and USB-C with Thunderbolt 3. Adoption has been driven by rising I/O performance requirements, such as flash storage and multi-gigabit networking, and the trend towards smaller laptops, with fewer ports and externally pluggable peripherals.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23194
www.ndss-symposium.org

Fig. 1: Thunderbolt dock with FPGA implant, an implementation of our I/O-security research platform

DMA introduces an intimate security relationship between the general-purpose CPU, its memory, and peripheral devices (which themselves frequently contain processors): it allows peripherals the ability to read or overwrite key operating-system (OS) internal data structures in kernel memory, placing the peripheral within the OS's Trusted Computing Base (TCB). The deployment of Firewire in the early 2000s led to the emergence of *DMA attacks* in which external devices, as well as other Firewire-enabled computers, were used to extract data from, or gain privilege on, target systems [7], [12], [13], [16]. Both the performance and vulnerability of DMA allowed for highly effective "drive-by" attacks extracting confidential memory contents or compromising system integrity.

Contemporary hardware and OS vendors are aware of these threats and employ an Input-Output Memory Management Unit (IOMMU) to limit access by DMA-enabled peripherals to system memory. MacOS, Linux, and FreeBSD, for example, can be configured to open up only limited portions of kernel memory to DMA, in order to prevent malicious devices from extracting encryption keys or modifying kernel data structures. The principle of this approach is similar to that of the *Memory Management Unit (MMU)* used for memory protection on general-purpose CPUs since the 1960s: the physical address space is virtualized to produce a number of *I/O virtual address (IOVA) spaces* through which DMA access from peripherals is transformed and limited (Figure 2). Just as the OS imposes virtual address spaces on processes to isolate them from kernel memory and one another, the OS constrains PCIe devices to performing DMA via specific I/O virtual address spaces that contain only mappings for memory

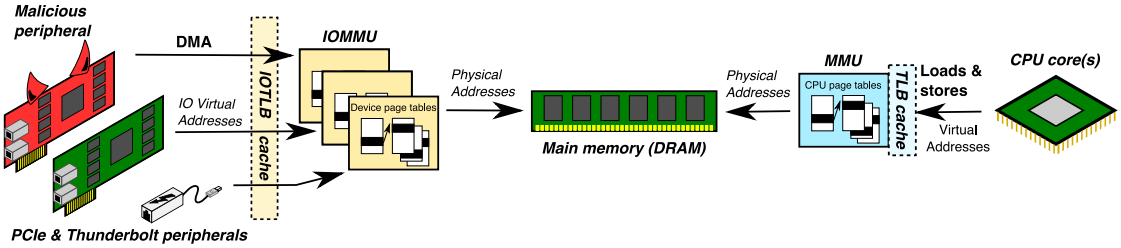


Fig. 2: The IOMMU translates I/O virtual to physical addresses and applies access control, similar to how the MMU translates virtual addresses from processes.

intentionally exposed by the OS or corresponding device driver – e.g., to allow packets to be read or written by a network card. This approach recognizes that devices may be untrustworthy, and allows such devices to be considered outside of the OS TCB. IOMMUs are widely believed to be effective in limiting DMA attacks.

In this paper, we explore IOMMU protection strategies employed by a number of widely used operating systems, and reveal a substantially more nuanced state of affairs. An essential insight is that, while IOMMUs allow peripheral devices to be constrained, the DMA interface between device drivers and peripherals is a porous and complex attack surface that malicious actors can manipulate to influence software behavior and trigger vulnerabilities. The comparison with MMU-based OS protection is apt: just as the system-call interface is one of the most critical security interfaces in an OS, used to constrain untrustworthy software requesting system services that access data provided by the attacker and lead to complex (and potentially vulnerable) kernel behavior, IOMMU-based protection is just as much about communication as it is isolating malicious peripherals from sensitive memory. And, as with MMUs, tradeoffs in IOMMU use necessarily exist – especially with respect to performance, where, just as with MMUs, TLB resources are limited, and page-table walks trigger additional memory traffic and memory-access latency, which constrain the acceptable vocabulary of this interface even after years of optimization [3], [9], [40]–[43], [53].

However, there are important differences from the system-call interface. The software side of peripheral DMA interfaces is not implemented by carefully hardened kernel system-call code, tested by decades of malicious attacks and fuzzing, but by thousands of device drivers that have been designed around historic mutual trust, hardware convenience, and performance maximization. Unlike most system-call interfaces, many key data structures shared between the kernel and peripherals are via shared memory – eg., descriptor rings – rather than register passing and selected copy avoidance. Prior work has suggested that shared-memory interfaces are particularly vulnerable to race conditions and other unsafe interactions [1], [25], [65].

To date, DMA attacks have focused on either systems unprotected by IOMMUs, or the narrow class of race conditions in which the IOMMU is left disabled or improperly configured during early boot – e.g., on hardware reset, in firmware, or during kernel startup [22], [24], [27], [46], [47], [59], [68]. To explore the more interesting classes of vulnerabilities and corresponding exploit techniques in the steady state of contemporary IOMMU-aware OSes, we have developed a novel hardware platform, Thunderclap, which is an FPGA-based PCIe-enabled device suitable for use with internal PCIe slots, external Thunderbolt 2 ports, and external USB-C ports with Thunderbolt 3. We have designed several physical embodiments of the FPGA-based platform including malicious docking stations (Figure 1), USB-C chargers, and projectors – all devices that end users are comfortable casually connecting to notebook computer systems, and that they can reasonably expect to borrow without compromising their personal data.

Rather than simply issuing loads and stores to sensitive kernel memory, we have developed a *peripheral device emulation platform*, allowing us to engage with more complex OS and device-driver behaviors by emulating full I/O devices that have DMA access. We utilize a CPU on the FPGA to implement a full software model of an arbitrary peripheral device, which allows us to choose the device driver we interact with, and to explore subtleties of interaction with shared-memory structures such as network card descriptor rings. Implementing our new adversarial model, we are able to interact deeply with OS functions such as memory allocation and free, IOMMU mapping creation and revocation, and so on. Thunderclap allows us to explore rich device, OS, and device-driver specific behaviors in this essential but largely unexplored vulnerability space.

The results are catastrophic, revealing endemic vulnerability in the presence of a more sophisticated attacker despite explicit use of the IOMMU to limit I/O attacks. We describe a range of new vulnerability classes, but also how conventional exploit techniques used in software-based attacks, and the mitigations used to limit them, differ in the context of DMA-based attacks in the presence of an IOMMU. Adversarial techniques differ substantially in this new space; for example, attackers can trigger new vulnerable behaviors – such as holding IOMMU windows open awaiting a low-probability shared-memory race. We find that mitigation techniques intended to limit userspace attackers via the system-call interface, such as KASLR, are applicable but require careful re-application in the DMA context due to historic assumptions – for example, the common practice of leaking kernel pointers to peripheral devices is no longer acceptable. Two years of interactions with major OS and device vendors have led to significant security updates, and explicit recognition that *OS IOMMU bypass vulnerabilities* are within vendor threat models. We are able to achieve IOMMU bypass within seconds of connecting on vulnerable macOS, FreeBSD, and Linux systems across a range of hardware vendors. Apple, Microsoft and Intel have issued security updates to partially address these concerns. In this paper, we:

- Provide background on I/O, DMA, and IOMMUs.
- Present our methodology, including threat model and I/O-security research platform with peripheral device models.
- Survey a range of current general-purpose operating systems for vulnerabilities, demonstrating escalating complexity as the sophistication of IOMMU use grows.
- Consider the suitability of existing access-control techniques.
- Discuss how these problems may be mitigated, including considering performance constraints.
- Conclude with a consideration of related work, vulnerability disclosures to date, and future areas of work.

II. Background

In this section we introduce a number of key technologies, the landscape of existing attacks, and modern defenses. We describe how systems are currently structured and the vulnerabilities that an expanded threat model exposes.

A. Interface classification

We can classify devices into two broad categories. The first uses a protocol-based approach that may be described as message-passing, where memory is not accessed directly. This covers protocols such as native USB and SATA.

The second uses a shared-memory approach. We classify shared-memory interfaces into several categories: inside-the-case inter-chip communications, soldered or modular, typically interconnected with PCI Express; external 'pluggable' devices, typically via Thunderbolt; system-on-chip (SoC) devices, typically via on-chip interconnect such as AXI; memory shared between computers in a clustering arrangement (remote DMA or RDMA). Today, most peripherals of any scale, e.g. network cards and GPUs, perform DMA, allowing them to access shared system memory. Our experimental work focuses on PCI Express and Thunderbolt for practical reasons, but would equally apply to on-chip devices. We outline some of these technologies in subsequent sections.

B. Peripheral technologies

The PCI Express (PCIe) interconnect [51] is the backbone of laptop, desktop, and server computers. Peripheral devices such as those for networking and storage attach (directly or indirectly) to PCIe. Each side (device and CPU/main memory) can issue memory requests of the other. PCIe cards are mostly used within the computer's physical enclosure.

Thunderbolt [28] is Intel's proprietary external cabling system that combines hotpluggable PCIe and video. It is popular for connecting docking stations and 'dongles' to laptops, which add external PCIe devices (graphics, storage, networking) and/or additional video outputs. Thunderbolt 1 & 2 use the mini-Displayport connector and are most prevalent on Apple laptops and desktops.

USB Type-C [64] is a multipurpose connector standard. By default it carries USB, but devices may add a microcontroller that can negotiate the port into an 'alternate mode', to carry a different protocol. These include video (Displayport/MHL/HDMI), analog audio, and Thunderbolt 3. Type-C also provides Power Delivery modes and is a popular means of charging devices.

Thunderbolt 3 [30] is behaviorally similar to the previous Thunderbolt 1 and 2 but with additional speed modes, and is conveyed over the Type-C connector. Thunderbolt 3 is increasingly widespread on middle- and high-end laptops.

The IOMMU (Input-Output Memory Management Unit) sits between main memory and PCIe devices (including those externalized via Thunderbolt), applying address translation and protection against requests from devices. Originally designed for virtualization – dedicating peripherals to different virtual machines – it has since been repurposed to protect non-virtualized machines against malicious peripheral DMA.

C. DMA attacks

The threat from peripherals first came to light with the spread of Firewire, an early competitor to USB, used by vendors such as Apple and Sony. Unlike USB, Firewire provided DMA to external peripherals. This improved performance by reducing host-controller-directed memory copying, allowing peripherals to directly address host memory.

Initial DMA attacks used this Firewire feature to read physical memory of a computer and then apply standard *forensic memory techniques* to compromise the system, e.g., to steal passwords or reveal disk encryption keys [7], [12], [13], [16]. With advances in technology, such attacks were updated to use PCI, Cardbus, PCI Express and Thunderbolt – both externally and internally [6], [19]–[21], [31], [56], [58].

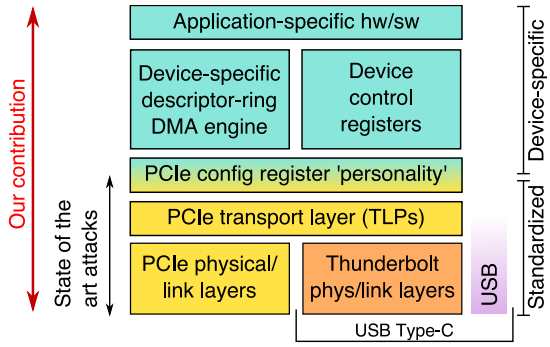


Fig. 3: Stack of a typical network or storage device. Lower layers are standardized, while the DMA and application layers vary among devices. Implementing all layers in a software model allows us to explore vulnerabilities throughout the stack.

In light of these attacks, operating systems had to improve their use of system protections. A key landmark was macOS 10.8.2 in 2012, the first time the IOMMU was enabled by default for protection against malicious peripherals. As a result, a large swath of DMA attacks were blocked, resulting in a refrain from attack authors that the IOMMU solved the DMA attack problem:

> "IOMMU ... does appear to provide protection against simple DMA attacks effectively" [32]
>
> "DMA does not work! what to do?" [20]

(Further discussion of the literature is given in Section X).

D. IOMMU primer

Since the IOMMU is the primary place where protection is implemented in the input/output (I/O) system, we outline its operation here. Implementations, namely Intel VT-d [29], AMD-Vi [2], and Arm's System MMU [4], are broadly similar, with minor differences.

The IOMMU implements a similar protection model for devices as the MMU (memory management unit) does for processes. Both involve the translation of addresses in memory read and write operations, as well as access control.

In brief, the MMU translates *virtual addresses* (used by a program or *process* on the CPU) to *physical addresses* (used by the underlying hardware memory). It uses multiple levels of *page tables*, each translating a smaller region of memory (address space). Different tables are switched in and out each time the processor switches to running a different process. Each unit of translation (or 'page') may have different read, write, or execute permissions, allowing one program to have its memory protected from another. Because a full table lookup is slow, the Translation Lookaside Buffer (TLB) is a cache of recently used translations.

The IOMMU mirrors the MMU operation, although for accesses from peripheral devices. I/O devices generate their own memory read and write transactions. Translations, this time from *I/O virtual addresses* (IOVAs) to physical addresses, are performed using the same table structure. However, because multiple I/O devices may make accesses at the same time, we can have a separate table for each device, comparable to the use of a separate MMU page tables for each OS process. PCIe allows 2^{16} devices, so there are potentially 2^{16} sets of tables

– although in practice a system may have only a few dozen devices present. For performance, there exists an Input/Output TLB (IOTLB) to cache recent translations. The arrangements of MMU and IOMMU are shown in Figure 2.

OS IOMMU bypass vulnerabilities arise when a malicious attacker is able to manipulate OS, device-driver, or application behavior to bypass intended protections, allowing undesired attacks on memory integrity or confidentiality. *Spatial vulnerabilities* occur when the 4KiB page (or superpage) granularity of translations allows undesired access beyond the intended physical memory range, or when permissions are set more broadly than necessary – for example, if only a sub-page-size region of memory is intended to be exposed by DMA. *Temporal vulnerabilities* occur when IOMMU mappings are open longer than necessary, allowing undesired accesses when memory may have been reused for other purposes – e.g., if IOTLB invalidations are performed asynchronously to allow DMA across memory reuse. Previous work [37], [42], [53] hypothesized the IOMMU suffers from such vulnerabilities but without describing any exploits.

III. Methodology

This section describes our threat model, including our aims and the practical opportunities open to attackers. It describes the features of our I/O-security research platform and our test environment. It describes how our platform allows us to fully explore the breadth of what can be achieved, with different scenarios where devices are used. In subsequent sections we use it to survey operating systems and then focus on them in detail.

A. Threat model

We consider *malicious peripherals*, i.e., hardware devices that may be attached to a computer system for ill intent, or an existing peripheral that may be compromised to the same ends (e.g., via malicious firmware update).

We focus on peripherals that can read and write system memory, directly or indirectly, via PCIe, Thunderbolt, or on-chip interconnect. Message-passing peripherals such as USB or SATA devices are not covered by our work and have different threat models (e.g., [49], [57]). However, the *host controllers* (i.e., the bridges from these protocols to memory transactions – usually PCIe) – are in scope. An attack from a USB-only peripheral (not via Type-C and Thunderbolt) would first require a host controller exploit: it may be feasible, but we do not explore this. Attacks may be external, via addition of a pluggable device, or internal, via compromised firmware of an existing device.

Attackers can present themselves as whatever kind of devices they wish by selecting their PCIe device ID, allowing them to select the vulnerable device driver of their choice. Peripherals may be external or internal; hot-pluggable, modular, soldered-down, or on-chip. Physical form factor has limited impact on the exploits a device can undertake, but it has a substantial bearing on user expectations. The form factor can be used to mislead users, shaping a device to look like one object but act like another. Users may not understand how much access they are granting to a device that they plug in.

Our platform fits into a number of physical forms to misdirect user expectations, including a **docking station**, **charger**, and **projector**. These are items users might borrow or connect to without considering security implications. Additional scenarios might be:

Compromised dongle. An existing Thunderbolt Ethernet or Wi-Fi dongle that contains a standard PCIe chip has its firmware compromised (e.g., via [32] or [11]). The dongle is fully functional, yet with additional trojan functionality to exfiltrate data. The dongle is left in a meeting or hotel room for an unsuspecting visitor.

Supply-chain attack. The firmware of a PCIe network card has exfiltration functionality added in the factory or in the supply chain, before it is installed in a customer server. Alternatively, a bad firmware update is applied in the field.

While we consider primarily PCI Express and popular laptop, desktop, and server operating systems, our work generalizes to any device where memory is exposed to peripherals. Our focus is on use of the IOMMU for host protection. Hypervisors using it to delegate peripherals to virtual machines, and to protect from breaking out of VMs using IO devices, present a related but distinct problem. The same IOMMU is used in a very different way, with divergent threat models. This also applies to OSs such as Qubes or Bromium that launch a separate virtual machine for each task, using the IOMMU to delegate peripherals to one specific VM.

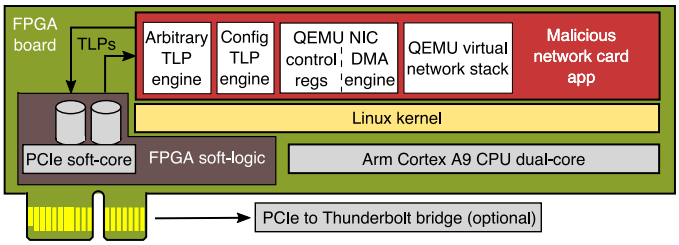


Fig. 4: Implementation of fully-functional network card using a QEMU device model running on FPGA

B. The Thunderclap I/O-security research platform

To investigate deep interactions with peripherals, we required a more intricate research platform than previously published. The full peripheral stack is depicted in Figure 3, including parts that are standardized and those that are device-specific. Because the device-specific layers are those that have most interaction with software, we needed an implementation that extends substantially beyond the work of previous researchers, who implemented only the standardized layers. To do this, we needed flexibility to probe both hardware and software, and so we built our research system using a complex software stack running on an FPGA, as depicted in Figure 4.

For the hardware, we used the Arm Cortex A9 CPU on an Intel Arria 10 FPGA to run a software-defined device model extracted from the QEMU full-system emulator. Additionally FPGA soft-logic allows it to generate arbitrary PCIe packets.

1) Baseline to reproduce the state of the art: The PCIe transport layer imposes the semantics of memory on top of the underlying layers that provide reliable end-to-end delivery of packets. These *transport layer packets* (TLPs) may be reads and writes classified as either memory, configuration, legacy-I/O, or other messages (such as power control). The replies to read requests are *completion* packets, which contain either the requested data or an error code.

Our PCIe hardware delivers raw PCIe TLPs to programs on the FPGA CPU through simple queues, which enables our adversarial application to send and receive arbitrary packets. This allowed us to build a baseline platform to test vulnerabilities to generic PCIe devices, similar to prior literature. The baseline software can perform DMA by generating arbitrary memory read and write packets and interpreting their results, including returned data and indications of errors.

CPU	Motherboard	Firmware enables IOMMU by default	Connection	Operating systems tested
Intel i7-7700HQ	Dell XPS 15 9560	✓	Thunderbolt 3	Ubuntu, Windows 10 Pro
Intel i5-6360U	Macbook Pro late-2016	✓	Thunderbolt 3	MacOS, Windows 10 Home/Pro
Intel i5-4278U	Mac Mini late-2014	✓	Thunderbolt 2	MacOS
Intel i5-4570	iMac 27" late-2013	✓	Thunderbolt 2	MacOS
Intel i5-4670	Asus Z87 Deluxe Dual		PCIe	FreeBSD, Ubuntu
Intel i7-4770	Asus Q87M-E		PCIe	Windows 10 Enterprise
Intel i7-930	Intel DX58SE	✓	PCIe	FreeBSD, Ubuntu/RHEL/Fedora
Intel Xeon E5-2670 SR0KX	Intel S2600CP2		PCIe	FreeBSD, Ubuntu
AMD Ryzen 1600X	MSI X370 Gaming Plus	'Auto'	PCIe	Ubuntu

TABLE I: Victim machines for experiments

2) Implementing a full device model: To go further, we wished to emulate a functional device that would cause the driver to be activated on the victim OS and expose data via the IOMMU. We used a software model of an Intel 82574L Gigabit Ethernet Controller from the QEMU full system emulator [8]. This device has drivers for each operating system we investigated. We extracted the QEMU `e1000e` device model and ran it on the FPGA CPU. An extremely cut-down version of QEMU's main loop has to be run in order to keep the simulated model operating. Incoming PCIe packets are translated into QEMU function calls in the device model. Similarly, QEMU's simulated 'DMA' is translated into real PCIe DMA transactions. This enabled our emulated device to generate the same memory reads, writes and interrupts that a real device would generate. QEMU's internal network stack allows our fake network interface card (NIC) to generate plausible traffic such as DNS and DHCP – we are not only a malicious device, but one that functions correctly as far as the operating system is concerned. The complex software dependencies required by QEMU mandated the full POSIX environment provided by Linux, in contrast with a real PCIe peripheral that usually runs a much more minimal embedded software stack.

With a functional emulated device, which worked despite the latency of software-generated packets, we could then add a malicious payload. We added an adversarial component which was made aware of the state of the emulated NIC and generated additional malicious DMA traffic and additional PCIe state.

3) Platform form factors: Thunderclap runs on FPGA evaluation boards, including the Arria 10 SoC Development Kit. Noting that user expectations are molded by the physical shape a platform takes, we have designed (although not fully engineered) a number of embodiments of the platform into form factors users might expect:

- A Thunderbolt docking station, where the Arm drives the docking station I/O (ethernet, USB, etc)
- A USB-C projector, which has an internal FPGA as well as a Thunderbolt bridge to extract video
- A USB-C charger, to charge a laptop as well as provide a malicious Thunderbolt FPGA

C. Test environment

We attached the FPGA to a number of laptop, desktop, and server systems to test different exploit paths, over a wide range of operating systems. Full details of the hardware/software combinations are shown in Table I.

We focus on the Intel and AMD IOMMUs in our study. In the mobile space, ARM's System MMU (SMMU) applies broadly the same concepts, and a natural extension of our work would consider use of the SMMU. However, we note that our study already covers the kernels used in Android (Linux) and iOS (XNU, common with macOS). The most interesting attacks on these platforms (malicious firmware in radio basebands, cameras or network devices) would require more reverse engineering to implement, since their software environment is proprietary. Additionally, these platforms do not offer PCIe or Thunderbolt to external devices as laptops do, so we could not reuse our existing research hardware.

D. Vulnerability space

Our aim is to investigate the shared-memory vulnerability space. In doing so, we examine how it is exploited through increasingly complex interactions with the operating system and device drivers.

To illuminate this, we illustrate how kernel and device-driver vulnerabilities allow us to extract private data (for instance, plaintext VPN network traffic), change kernel behavior (for example, change control flow by manipulating code pointers, allowing construction of malicious programs from snippets of pre-existing executable code by means of Return Oriented Programming (ROP) techniques), and circumvent memory protections. We then review our vendor interactions, the effectiveness of available mitigations and potential future directions.

IV. Operating-system survey

To understand how the IOMMU is used in different OSs, we performed an analysis of documentation, source code, and (where necessary) IOMMU page tables of running machines. A summary is given in Table II, which lists the OS versions we used for subsequent experiments.

We found that many systems did not even turn the IOMMU on: either it was disabled in the firmware, or the operating system required obscure configuration to enable the IOMMU. MacOS was the only OS to enable the IOMMU by default. It is notable that even RedHat Enterprise Linux 7.1 (which is Common Criteria EAL4+ certified [55]) did not enable the IOMMU by default. On those systems without default enablement, we set the necessary configuration to enable the IOMMU for device protection.

When the IOMMU is enabled, there are two broad modes that are used. *Shared mappings* have a single IOMMU page table that is used by all devices. *Per-device mappings* implement a different page table for each PCIe device. We discuss the implications of these design choices in following sections. Subsequently, Section IX considers why the IOMMU is used the way it is.

V. Attacks without OS interaction

The most basic vulnerabilities may be explored with our baseline platform, which is able to generate arbitrary PCIe packets from software. Generating independent memory transactions replicates prior work in that it has no interactions with the kernel or any device drivers; it naïvely explores what data it can access at a hardware level. In principle any PCIe device with DMA capability could perform similar attacks, though a particular product (as [20]) may have its own limitations.

A simple approach is memory probing, looking for accessible memory regions. As an FPGA able to make PCIe memory

Operating system	Build/ kernel	Can use IOMMU	Default enabled	IOMMU page mappings		Vulnerability				
				Shared	Per-device	Data leakage	Kernel pointer	Shared-allocator	Spatio-temporal	ATS
Windows 7						✓	✓	✓	n/a	
Windows 8.1	9200					✓	✓	✓	n/a	
Win 10 Home/Pro 1709	16299					✓	✓	✓	n/a	
Win 10 Enterprise 1607	14393	✓		✓		✓	✓	✓	n/a	?
Win 10 Enterprise 1703	15063	✓		✓		✓	✓	✓	n/a	?
MacOS 10.10-10.13		✓	✓	✓		✓	$< 10.12.4$[1]		n/a	
Linux: Ubuntu 16.04	4.8/10	✓			✓	✓		✓	✓	✓
Linux: Fedora 25	4.8	✓			✓	✓		✓	✓	✓
Linux: RHEL 7.1	3.10	✓			✓	✓		✓	✓	✓
FreeBSD 11	11	✓			✓	✓	✓		✓	
PC-**BSD**/TrueOS 10.3	10.3	✓			✓	✓	✓		✓	

[1] Fixed after our disclosure.

TABLE II: Operating system survey, describing applicability of our vulnerabilities to different platforms

requests, we are able to scan I/O virtual addresses. When reading, the PCIe transport layer returns either Unsupported Request (indicating we were not allowed to read) or Successful Completion (our read was permitted).

It turns out that most operating systems use relatively low I/O virtual addresses, so scanning the first few gigabytes of memory is sufficient. We call accessible regions *windows*, i.e., groups of pages that the OS has intentionally or unintentionally exposed to our device. On discovering a page is accessible, we can also look inside and possibly change the contents. We used this as a starting point for attacks against IOMMU and non-IOMMU systems.

A. Microsoft Windows

Of the operating systems that we studied, Windows uses the weakest form of IOMMU protection. We were able to compromise it using the baseline platform with no device model, with minimal effort. These attacks are perhaps uninteresting in terms of characterizing the attack surface, but their consequences are grave.

Attack story 1: Windows 7, 8.1, 10 Home/Pro
Most versions of Windows do not use the IOMMU and so are entirely unprotected from DMA attacks. This includes versions prior to Windows 10, and Windows 10 Home and Professional editions. We verified that all memory is exposed to peripheral devices, and so an attacker has full access to read and modify it. A user of such a machine is entirely unprotected from malicious devices. For example they can search for and replace parts of the Windows code with their own, or read secret data from memory.

Attack story 2: Windows 10 Enterprise
The only version of Windows to support the IOMMU is Windows 10 Enterprise, which uses it just for its optional 'Virtualization-Based Security' (VBS) feature.

VBS runs the primary 'root' Windows system inside a Hyper-V virtual machine, running a second container alongside. The container's minikernel is intended to protect private data such as encryption keys, constraining access from the root OS. VBS can implement Device Guard (DG), which prevents execution of malicious code; and Credential Guard (CG), which prevents secret data being read. DG and CG are not enabled by default, and the enablement process is sufficiently complex that it would likely be usable only in a controlled corporate environment. For example, DG and CG will not enable without UEFI and Secure Boot being enabled.

The IOMMU is intended to prevent devices bypassing the hypervisor's protections. Extended Page Tables (EPT) in the MMU are used to remap guest physical addresses to host physical addresses to isolate the virtual machines; this applies only to the CPU, so the IOMMU is used to prevent devices attached to the root OS from accessing the secure container.

Build 14393 specifically allowed us to attach a debugger to Hyper-V with VBS on, allowing us to examine the IOMMU page tables. We found all devices share a single I/O page map, meaning that memory exposed to one device is exposed to all. Furthermore, the vast majority of physical pages are mapped 1:1 into I/O virtual address space and read/write – whereas VBS may protect the container and hypervisor, the root OS is unprotected. We verified this with the FPGA: by scanning through physical memory, almost all memory pages were accessible to the attack device.

Disclosure We first contacted Microsoft in 2016, and have been in ongoing discussions. In 2018, they accepted that DMA attacks are within their threat model and announced Windows 10 Kernel DMA Protection [44], where the IOMMU is enabled in firmware [69] and Windows uses it for protection against Thunderbolt devices (only). This sits in the PCIe memory allocator and only opens IOMMU windows for memory allocated to devices, protecting the majority of the Windows system memory. It applies only to devices shipped with version 1803 and not to earlier systems upgraded to 1803 unless the vendor ships a firmware update. It also requires changes to third-party drivers to support DMA remapping.

Related work Our explorations augment growing interest in DMA attacks on Windows [6], [15], [20]–[22], [61]. They are included here as a demonstration of an OS that makes poor use of the IOMMU to defend against DMA attacks and the use of our platform to reproduce state of the art attacks.

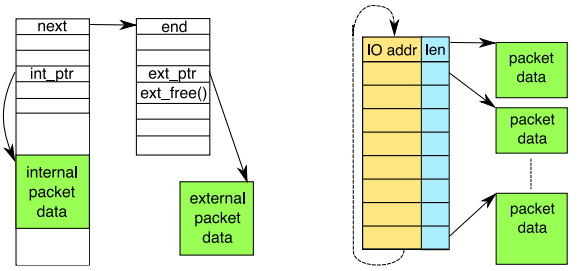
Other OS MacOS, and FreeBSD and Linux when enabled, use the IOMMU such that they do not give the attacker such full access to system state.

VI. Richer device-driver interaction

An attacker who behaves as a real device has greater power. Presenting as a real peripheral, it interacts with the device driver and operating system in a way that a baseline DMA platform does not. In this and subsequent sections, we consider what it means to be a full device attacking OSs with defenses against malicious devices.

To understand more complex vulnerabilities, it is useful to be aware of the architecture of a modern peripheral such as a network interface card (NIC), which we implement as our device model. Conceptually, a network card can be viewed as a bidirectional pipe: packets come in, packets go out. Several architectural constructs make it efficient for both hardware and software, which are common across vendors and across operating systems. To understand the problems of device security, it is important to understand how a NIC functions and how it interfaces to software.

The first problem is that packets are generated and consumed in several stages. For instance, an outbound TCP packet might have its payload generated by an application. The TCP layer prepends



a OS buffer chains b NIC ring buffer

Fig. 5: Common OS and network-card data structures

a TCP header. Then an IP header is prepended. Other layers such as IPsec or VLAN tagging may add additional headers. Finally an Ethernet header is prepended and a CRC may be appended. This full packet is dispatched to the network hardware for transmission.

In a naïve implementation, each prepend would involve copying the packet to free up space for the new header. However, performance dictates slow memory copying should be avoided. Instead, a common design pattern stores the packet in a linked list of memory blocks, allowing addition and removal of headers just by changing pointers. For efficiency, pools of these blocks are statically or semi-statically allocated, and typically there are two types. The first is the *inline buffer*, where the linked-list data structure contains a small fixed buffer that is enough for header fields, acknowledgements, and other small packets. The second uses an *external cluster*, where the linked list points to a larger standalone memory block – typically used for large payloads. This pattern exists in macOS and FreeBSD as the *mbuf*, Linux as the *skbuff* and Windows as the *NET_BUFFER_LIST*. Figure 5a gives an illustration.

The next abstraction concerns the hardware/software interface. Allowing the NIC to read and write packet data directly from memory is more efficient than using the CPU. To avoid hard-coding OS-specific data structures into silicon, the NIC uses *scatter-gather lists*, lists of addresses and lengths that the NIC should read to transmit a packet and where to write received packets. The NIC driver translates between OS-specific data structures like *mbuf*s and scatter-gather lists.

Since the drivers on the CPU and the NIC are processing concurrently, the *ring buffer* data structure is typically used. This provides a circular buffer of address/length tuples, a circular scatter-gather list (Figure 5b). When sending, the CPU writes pointers into one end of the buffer and the NIC consumes from the other end. By moving the read and write pointers, the driver indicates to the NIC there is new work, and the NIC confirms when the work is done. The driver's task is to allocate memory, to keep the ring buffer full of any outbound work and dispatch incoming packets to the network stack.

Finally, it is worth bearing in mind the performance requirements. A 40Gbps NIC as used in a datacenter node can transfer up to 60 million packets per second at line rate, depending on packet size. Since each packet may have several scatter-gather entries, $O(10^8)$ individual DMA operations per second might be needed for each direction.

A. IOMMU usage by network devices

When an IOMMU is in operation, the most obvious way to use it for protection requires some changes to ring buffer usage. First, packet data must be allocated from a pool of physical memory that allows exposure to devices (some memory may be inaccessible due to hardware limitations). Second, before a data block is placed in the ring buffer for transmission, a *window* must be opened for it to be accessible by the device. This involves creating a mapping for the block in the IOMMU page table. Third, the address written into the ring buffer is now the I/O virtual address of the mapping, rather than the physical address. Finally, when the device is finished with the data, the operating system should close the window again, revoking the mapping from the IOMMU page table and IOTLB.

While this is the obvious usage model, various OSs deviate from it, as explained in subsequent sections.

B. IOMMU usage by other devices

While we have focused on NICs, other devices have similar structures. NVMe flash storage is based around a similar ring buffer for data blocks. The XHCI host controller interface for USB uses scatter/gather rings, and the AHCI interface for SATA uses in-memory pointer tables to indicate command and data regions for transfer. In each case the pattern of following pointers in host memory looks similar, though the semantics of the payloads transferred is different. In Section IX-B we map spatial utilization, including SATA and GPUs.

C. Our platform as a NIC

Our implementation emulates an Intel 82574L NIC, which provides it with full visibility of the ring buffer and its data payloads. We added a variety of adversarial functions to the basic NIC to examine vulnerability to different exploit techniques. Because we run a software model, it is relatively easy to make substantial changes to its behavior.

At a basic level, an attacker can exploit the self-descriptive nature of PCIe: that devices advertise features and those advertisements are trusted. For example, the attacker can set up configuration registers as needed to bypass whitelisting.

More deeply, our NIC has visibility of the ring buffer structure and so is notified of the locations of packet data in the I/O virtual address space. In theory, the NIC is notified only about data buffers; thus, we can exploit the fact that IOMMU mappings have a minimum granularity of 4KiB pages, and thus can look for data on the same or nearby pages. Additionally, we can change the way the NIC uses the ring buffer – e.g., failing to indicate we are finished with data buffers, preventing their IOMMU entries being invalidated.

At a higher level, we can also generate spurious network traffic against the host (perhaps announcing a new default route to divert connections to our NIC, ARP poisoning, HTTPS man-in-the-middle and so on). However, we consider this a separate class of attacks that does not require being a DMA-enabled peripheral and has been covered in other work, for instance over USB [49].We therefore consider them out of scope, but note that powerful attacks may be possible with a combination of exploits: HTTPS man-in-the-middle with additional knockout of the browser certificate check functions could be very potent, for example.

In following sections, we demonstrate the additional power that awareness and interaction with the device model gives to the attacker.

D. Shared IOMMU mappings on macOS

MacOS was the first system to deploy default use of the IOMMU, since early Firewire and Thunderbolt attacks focused on Macs. Our investigations reveal macOS uses *shared mappings*, a single IOMMU page map that is shared among all devices.

Therefore memory that is exposed to one device is exposed to all. This means one device can snoop on memory intended for another – examples might be a malicious peripheral keylogging via the USB controller or reading the framebuffer. For example, the framebuffer is always exposed on systems with discrete graphics.

In fact, macOS's *mbuf*s are a special case – *mbuf*s are allocated during early boot and remain exposed to all devices at all times. Therefore every device has full visibility of network traffic continuously. This is weaker than even shared mappings, where most other peripherals' buffers are protected once they are de-allocated from the device. It appears this is due to the network stack being derived from FreeBSD, which exists as a semi-separate codebase within the macOS (XNU) kernel – but also avoids any IOMMU mapping and unmapping expense when transmitting and receiving packets.

Attack story 3: MacOS VPN cleartext data extraction
The first hurdle we encountered was that macOS does not attach drivers to unapproved PCIe devices connected via Thunderbolt. To overcome this, we changed the device and vendor IDs reported by Thunderclap to be an Apple-approved device (as dictated by a system configuration file). We set our NIC to mimic the version of the 82574L inside Apple's first-generation Mac Pro desktop.

We set up an IPSec VPN connection using the motherboard BCM57765 ethernet controller and sent 'secret' traffic over it. We then plugged in Thunderclap via Thunderbolt, recording all the memory windows that were passed to the platform over the course of the generic set up network traffic that macOS carries out with newly attached network devices (DHCP, IPv6 solicitations, multicast DNS).

In the memory windows were *mbuf*s the motherboard NIC had been asked to send. Reading through these windows, we identified plaintext leaked from the secret connection in other parts of the pages passed to our NIC; this is present due to in-place decryption performed within IOMMU-exposed memory.

Attack story 4: MacOS root shell via kernel pointer exposure
Using a similar technique, we were able to achieve a root shell on macOS. Every *mbuf* is 256 bytes long, and starts with a variable amount of metadata, including a pointer to the network data it carries. For some *mbuf*s, this is allocated externally to the 256-byte region: for others, it is internal. If the data is external to the *mbuf*, a pointer to a custom free function *m_free()* (along with three arguments to call it with) can be included in the *mbuf* metadata.

Due to the 4KiB page granularity, mapping the internal data to the NIC also exposes the metadata. This means that, from the NIC, we have access to a function pointer that the attacker can set to any value, allowing them to change control flow.

In that we appear to be a valid NIC, we are naturally given IOVA pointers to descriptor rings that point to the IOVAs of *mbuf*s and have both read and write permissions. Specifically, we scan the pages to find an *mbuf* with external storage, and modify internal flags and structures to ensure that the custom free-function will be called. We are then able to change the *m_free()* function pointer in the *mbuf* to point to an address of our choosing. We also control timing of the function call, since the NIC indicates it has transmitted an *mbuf* and when the OS should free it.

This is not enough; in order to make kernel code injection attacks harder, macOS employs Kernel Address Space Layout Randomization (KASLR). This adds a randomly-chosen offset (the *slide*) to the address of everything in the kernel. We determined that the slide is a multiple of 2MiB, meaning that low 21 bits of each address in the kernel are the same regardless of the value of the slide. We also found that the AHCI and USB drivers shipped with macOS leak the randomized virtual address of a kernel symbol through pages that they open to all peripherals at boot time. Since the lower 21 bits of the symbol are constant, we can scan I/O address space looking for symbols where these bits match; with a high probability these will reveal the slide.

To demonstrate exploitability of this vulnerability, we caused the CPU to execute the functions `panic` and `KUNCExecute`. The latter allows us to execute programs as arbitrary users, including root, from the kernel. We ran `Terminal.app`, which gave us a root shell on the machine. Additionally, we have found an instruction sequence that allows us set the stack pointer to a value of our choosing. This should be sufficient to allow an attacker to build a ROP attack on the kernel.

A simpler adversarial platform? The nature of shared mappings means we can achieve a similar effect with a basic device with no driver attachment. We can, for instance, scan memory with the FPGA looking for memory windows containing signatures that correspond with the BCM57765 descriptor rings. These allow us to find out the I/O virtual addresses of *mbuf* chains. We performed the whole attack using our non-NIC FPGA and were equally successful, replacing the free pointer in all *mbuf*s we found. Since we had no view of NIC state, this reduced the accuracy of the exploit firing to a few seconds, additionally filling the system log with (ignored) IOMMU page faults.

Disclosure We demonstrated this vulnerability in macOS 10.11.5 and disclosed it to Apple. Apple has since patched the vulnerability in 10.12.4. Kernel function pointers in *mbuf*s are now blinded by XORing with a secret cookie that is held in kernel memory not exposed to the peripheral device. Because the bottom 21 bits are known plaintext, we can generate a valid function pointer to 2MiB of kernel code despite the blinding; however, it is not called because the flags an attacker needs to modify have been moved outside DMA-able memory (though they can still cause the kernel to panic).

Other OS We identified a similar pattern on Windows using the kernel debugger. The `NET_BUFFER` contains some opaque structures, one of which sometimes contains a function pointer into `tcpip.sys`, and a distinctive flags word that makes it discoverable from the DMA device. We overwrote the function pointer in the debugger with a pointer to `KeBugCheckEx` (the 'Blue Screen of Death' function) and succesfully hijacked control flow. These data structures were entirely exposed to our FPGA (both NIC and non-NIC versions). We conclude that, were Windows to improve its use of the IOMMU, it would still be vulnerable to such an attack. Windows applies per-module KASLR, hence we must use `tcpip.sys` for initial ROP gadgets.

E. Per-device mappings on FreeBSD

FreeBSD uses a network stack with the same BSD origins as macOS, with similar *mbuf* structures. However, it provides per-device IOMMU mappings, with a different page map for each device. The baseline platform has no windows opened in its page map and is thus prevented from accessing any memory at all. If the device reports that it has the vendor and device ID of a NIC, but does not exhibit the expected behavior, the driver fails to attach the device and no mappings are opened.

However, memory windows are opened when Thunderclap masquerades as a functional NIC, and it is notified of their locations via the ring buffer.The *mbuf*s exposed contain *free()* function pointers which an attacker can overwrite to hijack control flow.

Attack story 5: FreeBSD kernel privilege via control flow
Despite very different IOMMU configuration, we can adapt attack 4 to work on FreeBSD. We used TrueOS (formerly PC-BSD) 10.3, as it is a desktop FreeBSD distribution that by default performs DHCP against an attached NIC; server FreeBSD requires explicit configuration of new NICs (it is possible broadcast configuration traffic may suffice for unconfigured NICs, though we did not try it). We attached our malicious PCIe NIC to the system. On boot, TrueOS performed DHCP against our NIC; we were handed *mbuf*s whose *free()* function pointers we could overwrite, enabling a call to kernel code of our choice. Since FreeBSD does not do KASLR, kernel pointers are static.

The driver tells the NIC the I/O virtual address of the transmit ring. We search it for the address of an *mbuf* by looking for non-2KiB-aligned addresses. Then we modify an *mbuf*'s flags to appear to the host as having external data and a custom free function. We also must create a reference count inside the body of the *mbuf*, and set a field that works as a pointer to the refcount to point to this value. To do this, we need the kernel's address for the *mbuf*. We derive this by masking the *mbuf*'s pointer to its internal data region.

An exploit using the custom free function is slightly more complicated on FreeBSD than macOS. In contrast to macOS, which gives full control over all three free function parameters, FreeBSD always calls the function with the address of the *mbuf* as the first parameter. However, with a gadget that allows the stack pointer to be set to the value found in the second argument register, there is enough to carry out a ROP attack.

Other OS Were macOS to use per-device mappings, a similar attack would still work subject to defeating KASLR. Linux prevents it since its *skbuff* locates function pointers and data on different pages, and only the data is exposed to our NIC. It is unclear if this is a deliberate design decision or an artifact of the allocator.

VII. Exploring the vulnerability space

Thus far, we described vulnerabilities in operating systems that may be exploited using relatively simple malicious behavior – there was no need for anything more complex.

However, the vulnerability space is much richer. Fundamentally, the malicious device presents to the operating system with a series of claims, about what it is, what resources it needs and how it behaves. Operating systems are entirely credulous of these claims, since they have no other means of distrusting them. As a result, a device driver is attached and further interacts with the device, generally believing everything the device says. As part of this interaction, memory is exposed to the device based on previous claims.

If the device is malicious, how can it manipulate the OS and device driver to abuse the shared-memory interface?

A. Linux

Unlike macOS, Linux has no sharing of IOMMU page mappings between devices – which should, in theory, provide better security. On macOS we exploited a spatial vulnerability, where more data is exposed to the NIC than it needs to operate. On Linux, this is also possible – devices are exposed pages of 4KiB granularity, and data is leaked from other parts of the same page. Can an attacker, as a NIC, force further leakage?

Attack story 6: Poor allocators and kernel NAT dispatch tables
We studied a Fedora 25/kernel 4.8 desktop with a genuine Intel 82574 motherboard NIC that used the `e1000e` driver. A primary function of the driver is to allocate *skbuff*s for incoming packets. These come from a pool, but the data buffers are allocated with a general kernel allocator, based on the maximum packet size. When 2KiB buffers are allocated, the other half of the 4KiB page is a 2KiB allocation from another part of the kernel. Using SystemTap tracing [54] we dumped pages that were exposed to the read queue of the Intel NIC. In these we found much kernel data, for instance the dispatch table for the `nf_nat` Network Address Translation packet rewriting functions. Since the NIC can write this table, a malicious function could be attached which rewrites destination addresses of packets to exfiltrate to a malicious Internet server.

Attack story 7: Spatio-temporal attack – UNIX domain sockets and VPN traffic
Having been given a packet, the NIC should update the ring buffer pointer to indicate the packet was accepted for transmission. Until the NIC updates the pointer, Linux will keep the window open assuming the NIC is still busy. We conceived a spatio-temporal attack where a malicious NIC can thus force the window to stay open and monitor data in other parts of the window.

We modified our FPGA NIC to drop return updates, causing windows to be left open. We then watched as other parts of the pages were reused over time. In the windows we saw syscall kernel stacks (not writable, but enough to break ASLR for kernel code and all data memory); UNIX domain socket traffic (as used by security protocols such as SSH agent authentication) and plaintext VPN traffic.

B. PCIe configuration vulnerabilities

PCIe allows devices to self-describe. A region of memory called *configuration space* contains a description of the device, accessed by special *configuration request* packets.

Configuration space provides information such as vendor, device type and ranges of registers the device provides. Additional features are indicated by *capabilities*, data structures that describe optional functionality such as power control. Firmware and operating systems use this data to understand what devices are present in the system and attach the appropriate software drivers to them.

This not only allows the attacker to select the device driver to target, but also to manipulate configuration of PCIe by the OS's bus management framework to maximize advantage during an attack.

Attack story 8: Full IOMMU bypass with ATS
PCIe can allow devices to carry out IOMMU translations themselves, bypassing the central IOMMU, with the rationale that devices can implement IOTLBs that are tailored to their requirements. This feature is called 'Address Translation Services' (ATS) in the PCIe specification, and 'Device TLBs' by Intel. Memory requests that have been translated by the device have a header bit set, which implies that the IOMMU does not need to apply translation. A device that supports ATS has the ATS capability in its PCIe configuration space, and the feature must be enabled by the OS otherwise such requests are dropped.

Linux's IOMMU subsystem allows any device with this PCIe capability to use ATS. We modified our NIC model to report support for ATS, and confirmed Linux enabled it.

Without the ATS capability, no memory windows were accessible to our device. When the device advertised ATS support, Linux enabled the ATS feature in the PCIe switches. Then we set the 'already translated' bit in each memory packet, and we had unrestricted access to memory. PCIe Access Control Services (ACS) can block such 'pre-translated' requests however this was not

Fig. 6: When a malicious Thunderbolt device is attached, Windows prompts for access without a description of what rights are being requested. Users cannot make an informed decision whether to enable a device.

enabled by default, perhaps due to performance considerations.

Other OS FreeBSD, macOS and (we suspect) Windows do not support ATS, so are not vulnerable to vulnerabilities of this nature. Of our test machines, only our server and Dell laptop supported ATS.

VIII. Device access control

In previous sections, we have explored the vulnerability space of operating system IOMMU protection using the Thunderclap platform. Here we briefly address access control security features of peripheral interconnects. These are largely orthogonal to the broader attack surface we have described but must be subverted by an attacker in practice.

1) PCIe: lacks protections to vet, block, or audit unauthorized devices. An attached device is automatically allowed to send and receive packets. The IOMMU filters only memory traffic. PCIe Access Control Services (ACS) can block other types but only in limited circumstances. Apart from pluggable ExpressCards, most PCIe devices are internal. Compromising the firmware of existing internal devices is thus the primary attack vector against PCIe.

2) Firewire: has no access-control: all connected devices are given rights to generate memory transactions. While mostly obsolete, adaptors from Firewire to Thunderbolt still provide an attack vector.

3) Thunderbolt (TBT): tunnels PCIe and DisplayPort (DP) video, over USB Type-C or miniDisplayPort connectors. Users can be (deliberately) confused whether a port supports TBT or just native USB/video. TBT supports an access control protocol, but it concerns only the TBT-PCIe bridge device and not the PCIe device beyond it. A PCIe device can be replaced without the access-control mechanisms being aware [21] and the system cannot query a PCIe device before enabling DMA.

MacOS applies whitelisting for Thunderbolt devices, keyed on their ID ROM. Apple's requirements are unclear, but generally a device sold as 'Mac compatible' will be on the whitelist. Once whitelisted, a device is free to make PCIe transactions, and all of the vulnerabilities that we describe will apply. Any whitelisted Thunderbolt device could perform our attacks. Many Thunderbolt to PCIe bridges (intended for external GPU enclosures) are whitelisted, so the attacker has a variety of choices. On a Mac, we could switch out the internal PCIe board of our Thunderbolt docking station, and the new PCIe device would be accepted by the OS without any messages or prompts. For example, a Thunderbolt device shaped as a 'charger' is indistinguishable from a USB-C charger.

Windows: Windows currently uses a Thunderbolt prompt as its only defense against DMA attacks: an approved device has full access to all of system memory. UEFI firmware settings allow TBT to operate in several modes – everything allowed; USB/DisplayPort only; always prompt the user; or 'secure mode', which checks a token previously saved on the device. On our Dell laptop, the default firmware setting prompts for each device. In Windows the user is prompted whether to disable PCIe access, allow only once, or accept and remember the setting. The prompt gives the TBT device name, which can be content-free (Figure 6), and does not represent the actual PCIe devices connected to the TBT bridge [21]. For a Thunderbolt dock with a trojan PCIe device inside, the prompt is no different. Additionally, replacing the PCIe board in an approved Thunderbolt device with a malicious one does not cause a subsequent re-authentication [21]. Since users become habituated to prompts, and can be misled by the physical shape of the device, they can be tricked into accepting a malicious device.

Linux will accept TBT devices that are authorized by the firmware and connected at boot. Patches for approval of hotplug devices have been produced by Intel [36] and distributions are beginning to implement user interfaces.

FreeBSD has no TBT hotplug support, although it will accept devices that are connected at boot and authorized by the boot firmware, as they present as generic PCIe.

IX. Discussion and mitigations

Our work with vendors (see Section XI) has caused them to ship mitigations to the specific attacks we have described. However, many of the vulnerabilities we uncovered concern the behavior of fundamental operating system components such as kernel memory allocators, IOMMU-controlling subsystems, and peripheral device drivers. In this section we consider how vulnerabilities in operating system IOMMU protection may be mitigated and why doing so is not necessarily straightforward.

A. New adversary models

Fundamentally, a change in threat model of operating systems and device drivers is needed. Peripherals can no longer be considered trustworthy and should be removed from system TCBs. OS developers (and attackers) have long understood that, despite using an MMU to separate kernel and user process, the system-call interface is a rich attack surface on the kernel. The interface offers breadth and depth of interaction with complex kernel subsystems, any of which might suffer a security vulnerability yielding ring-0 privilege. Our work shows that, with a shift in adversary model, I/O peripheral DMA interfaces offer an equally rich attack surface despite use of the IOMMU: the complex performance-sensitive feature-rich concurrent shared-memory interfaces used by peripherals can influence the behavior of numerous kernel-resident device drivers and subsystems.

Some of our discoveries reflect simple and easily corrected implementation mistakes. For example, I/O buffers should not be allocated from the same pool as kernel jump tables. Doing so would enable a malicious peripheral to gain arbitrary privileged code execution trivially by exploiting spatial or temporal vulnerabilities. More fundamentally, however, current OS designs expose millions of lines of ring-0 device-driver code to adversaries which they were never designed to protect against. Even with careful review, drivers are unlikely to resist attacks – and, as with system calls, the attacker is given their choice of code to attack: PCIe devices can declare the device ID of the weakest driver. Mitigations against system-call exploits, such as KASLR,

can be useful. However, they are no help if undermined elsewhere – as for KASLR when device drivers leak kernel pointers.

B. Performance

Previous work describes two main performance problems with the IOMMU. These are IOTLB pressure (caused by a large number of in-flight mappings fighting for limited IOTLB space), and the slow speed of IOTLB invalidations [3], [9], [48]. As a result, there is considerable cost to turning on the IOMMU by default, which may explain why Linux and FreeBSD do not do so.

These performance concerns complicate mitigating vulnerabilities in operating system IOMMU protection. Specifically, providing better protection may involve creating more IOMMU mappings to isolate individual data objects (to address spatial vulnerabilities) and invalidating the IOTLB more frequently or in performance-critical code paths (to address temporal vulnerabilities). Both of these techniques increase IOTLB pressure and reduce overall system performance, which makes adopting them less straightforward.

Some prior work describes improvements to the way the IOMMU is used, which claims to ameliorate this cost [3], [40]–[43], [53], although not addressing all of our vulnerabilities. Markuze [42] proposed *shadow buffering*, an IOMMU driver design that addresses both the spatial and temporal vulnerabilities. It involves copying data to and from a region of memory that is always accessible to a peripheral on map and unmap calls. While shadow buffering provides protection from known DMA attacks, its performance cost prevents it from providing a complete solution in practice. They later improve with DAMN [43], a scheme which implements a special allocator for pre-IOMMU-exposed packets. Since these allocators are dedicated to a particular NIC, it reduces invalidation and IOTLB churn, although all received packets must still be copied out of these buffers (increasing cache pressure) and inter-NIC traffic (for example, when routing) would require double-copying.

Address utilization study. Much of the focus of prior work has been on network stacks. We investigated whether similar work would apply equally to other device classes. In particular, much of the work focuses on either copying or pre-allocation of IOMMU-exposed memory.

We ran a number of benchmarks to study physical address usage across different device classes. The goal was to better understand whether work improving network stacks will generalise to other devices.

We tested Windows 10 Enterprise 1703 and Ubuntu 16.04 (kernel 4.14) with the IOMMU disabled, in each case recording the 'natural' physical addresses devices use for DMA. We did this by interposing a PCIe analyzer between the device's PCIe card and a slot on a Supermicro C9X299-RPGF motherboard, with an Intel i9-7940X 14-core CPU and 16GiB of RAM. 10 Gigabit Ethernet (Intel X520-DA), AHCI host controller for SATA storage (ASM1061) and GPU (AMD RX460 2GiB) cards were tested. Due to limitations of our PETracer ML analyzer, devices were restricted to PCIe Gen1 x4 (10Gbps total bandwidth), and trace recordings limited by the analyzer's buffer (2GiB of packets after filtering). These constraints reduce the speed of operation and the length of recording time, but do not affect the general distribution of measurements, which are not performance-sensitive.

We ran workloads designed to test behavior of full applications, rather than microbenchmarks, mostly based on the Phoronix Test Suite [35]. Due to OSes use of zero-copy techniques, microbenchmarks that replicate behavior of applications in a simplified way – with a smaller memory utilization – may not be representative.

Figure 7 shows address heatmaps for network, storage and graphics workloads that are representative of our dataset. They plot addresses on a 12th-order Hilbert curve (after [45]) where any contiguous address range appears as a block. For readability, each pixel aggregates 256KiB of address space.

Figures 7a, 7b show the memory reach of networking is fairly small, since packet buffers are often reused. This was repeated across other benchmarks testing both small packet roundtrips and bulk transfers which fully utilized the NIC.

Figure 7c is typical of storage behavior, in which much larger blocks are transferred – even, as in this case, when accessing small files. GPU workloads (figs. 7d, 7e) show a much broader reach, but in small blocks surrounded by memory untouched by the GPU. Figure 7e depicts a workload that will not fit in GPU memory, leading to it spilling to system memory across PCIe and yet broader memory footprint.

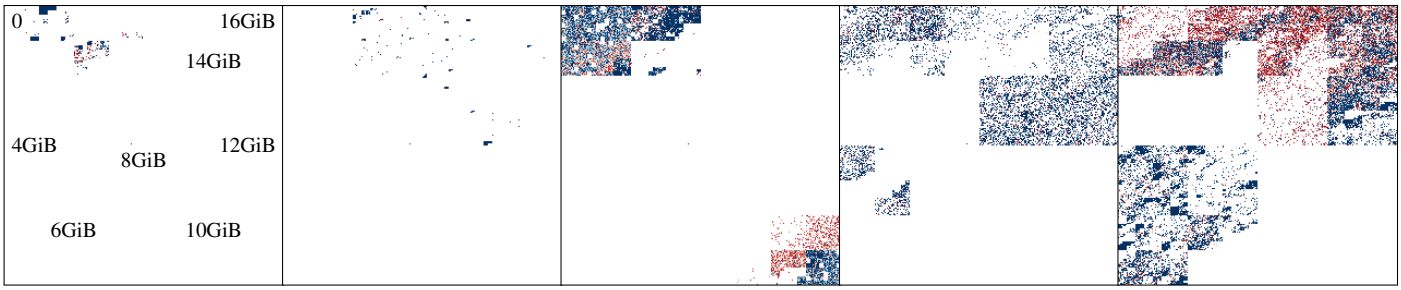
From these we can see that storage and graphics have much richer, more complex patterns than networking. A difference can be seen between purely communication devices, and those where a dataset is truly shared between CPU and device, as with the GPU. In particular, given small IOTLB sizes (64 in [48]), setting up thousands or millions of mappings for pages for a GPU is likely to cause heavy IOTLB pressure. Superpages, 2MiB or 1GiB of contiguous physical address space per entry, may help to reduce the IOTLB footprint. However, memory utilization in compute workloads (such as 3D models) may be difficult to reorganize to suit the IOMMU, or that may have an unacceptable performance cost for the application. Other work, for GPUs and other accelerators, indicates the performance problem is so bad that others [23], [26] have proposed redesigning or removing the IOMMU completely. Thus, the IOMMU performance problem is still present.

C. Mitigations and feasibility

To completely protect against malicious peripherals, changes in system design must be made to reflect their untrustworthy nature and removal from the TCB. Specifically, we recommend (1) exposing to peripherals only the minimum amount of data required for them to function correctly, with the most restrictive access permissions possible; (2) eliminating temporal vulnerabilities by ensuring IOMMU mappings are completely destroyed before IOMMU-mapped memory is processed, or reused by the host; (3) enforcing per-device I/O virtual address spaces to limit the ability of one malicious device to compromise the function of non-malicious devices.

There is a trade-off space involving software techniques for using IOMMU hardware: more safe use disrupts current software practices (e.g., colocating I/O data and kernel metadata or passing kernel pointers to devices) and performance (requiring additional memory copying/zeroing, greater memory fragmentation, or synchronous IOTLB invalidation). Adding expensive operations to performance-critical code paths can significantly decrease I/O throughput, which represents a barrier to implementing better protections. Concern about code changes should also not be underemphasized: seemingly Apple deemed restructuring their network stack to avoid leaking kernel pointers too invasive, instead encrypting them – contrasting with the preferable Linux design choice of placing them in a separate unmapped page.

These performance and implementation costs make complete protection from malicious peripheral devices challenging to achieve in practice. There is currently no general-purpose solution that provides protection from malicious peripheral



a Linux 10G ethernet, PostgreSQL pgbench, 100 remote clients

b Windows 10G ethernet, PostgreSQL pgbench, 100 remote clients

c Linux AHCI SATA, kernel compile, 28 threads

d Windows 10 GPU, Final Fantasy XV DirectX 1080p, tutorial mode

e Windows 10 GPU, Superposition DirectX 4K benchmark

Fig. 7: Physical address heat maps for different I/O workloads. Storage and graphics workloads are much richer than networking which often reuses smaller regions of memory buffers. Colors: white=unused, blue=lightly used, orange/red=heavily used. Each pixel aggregates 256KiB of address space, contiguous addresses form a block. (a) annotates the address layout of the 16GiB address space depicted.

devices without significant performance degradation, and many partial mitigations are not implemented in commodity operating systems, perhaps because of their impact on existing codebases. Below we discuss some potential mitigations, briefly outlining their effectiveness and implementation costs.

Device-specific I/O virtual address spaces can be implemented with small changes to an IOMMU driver, mitigating the ability of one malicious device to compromise the function of non-malicious devices (although not in themselves preventing PCIe ID forgery on DMA writes), and do not significantly affect system performance.

Allocator hygiene is a basic improvement: I/O data structures and sensitive kernel structures should not be allocated from the same pools. This fixes obvious spatial vulnerabilities, but it does not prevent I/O data structures from containing sensitive fields that a malicious device could exploit and does not segregate I/O data belonging to different devices or DMA transactions. This change would cause a relatively minor impact on existing codebases and performance, requiring the creation of new allocator pools and modification of I/O subsystems to use them. *Spatial segregation* extends allocator hygiene so that devices have access only to the minimum amount of data required for them to function correctly. I/O data is isolated from all other data (including kernel data and control fields in existing I/O data structures) and from I/O data belonging to different devices or DMA transactions. I/O control structures such as descriptor rings necessary for the device to function are similarly isolated. This change would eliminate spatial vulnerabilities but come at a significant implementation and performance cost. I/O data structures (e.g., network buffers) would need to be redesigned to separate I/O data from other fields, and the corresponding subsystem and device drivers would require non-trivial code changes to use the new structures and to isolate their data correctly. Isolating data in distinct pages would reduce performance by increasing memory usage, the number of IOMMU mappings, and hence IOTLB pressure.

Synchronous IOTLB invalidation would protect against temporal vulnerabilities by ensuring IOTLB invalidations are completed before the IOMMU driver reports them as such. Adopting synchronous IOTLB invalidation requires only trivial changes to the IOMMU driver, but can reduce throughput by as much as 80% for high-performance I/O workloads [42]. Alternatively, memory for I/O data could simply not be reused or reused only once a corresponding asynchronous IOTLB invalidation has completed, which would require changes to I/O memory allocators and reduce the efficiency of memory usage but perhaps have a less significant effect on I/O throughput. However, this change for reuse would not necessarily ensure that IOMMU mappings are completely destroyed before their underlying memory is processed by the host.

Buffer pre-allocation (as in [42], [43]) involves allocating memory for device I/O from a special pool that is perpetually exposed via the IOMMU. Some pre-allocation strategies involve copying data to and from the pool on IOMMU map and unmap calls, and some use memory from the pool for device I/O directly. Copying techniques mitigate spatial and temporal vulnerabilities when applied correctly, because they can provide protection at arbitrary granularity and copy specific fields from existing data structures. They require modifications only to existing I/O memory allocators, but they introduce negative cache effects and a significant performance overhead. Conversely, schemes that use memory from pools directly require modification to I/O allocators, I/O data structures, I/O subsystems (since the subsystems must determine before allocation time with which device to associate I/O data). They also introduce a new type of vulnerability because devices could modify pool memory that is used to store I/O data after it has passed system security checks (e.g., modifying the source IP address of a network packet after it has passed a firewall filter).

Byte-granularity or non-paged IOMMUs could prevent spatial vulnerabilities but would need a new hardware paradigm. Such range-based IOMMUs would require new drivers and have different properties related to translation lookup and IOTLB caching and invalidation.

Memory encryption would allow devices to have arbitrarily-sized memory regions only they can interpret. This would need hardware changes and transforms the problem into one of key distribution. Different components (NIC, PCIe switch, driver, network stack) have complex relationships, and safely distributing and revoking keys is a hard problem. AMD's memory encryption [33] does not handle this problem, allowing DMA only when a single system-wide key is used.

X. Related work

We divide prior work into the categories of peripheral memory access attacks that do not work against systems with basic IOMMU protections, carried out by previous attack platforms and by compromising firmware on existing devices; attacks against IOMMU-enabled systems that mostly exploit vulnerabilities in IOMMU configuration; and other uses of the IOMMU. In some cases we build on this work; in other cases prior work is orthogonal.

DMA attacks may be divided into those that involve attaching a hardware/software platform to a victim system, and those that compromise firmware of existing devices. The goals may be similar, but the route is quite different.

DMA attacks and attack platforms. DMA attacks were of concern even on 1960s machines [14]. More recently, DMA attacks have been performed against modern systems, using vectors such as Firewire, PCI and PCIe [6], [7], [12], [13], [16], [31], [58]. A number of these have spawned generic DMA attack platforms such as SLOTSCREAMER [19], PCILeech [20], [21] and Inception [39]. These platforms can attack many operating systems over various hardware interfaces. They steal sensitive data like encryption keys, violate kernel security policies, and even take complete control of a target machine. However, all of these attack platforms depend on unrestricted memory access to scan for sensitive structures or modify a specific location in memory. They do not work against systems with basic IOMMU protections. Additional reverse engineering efforts focused on Thunderbolt [56] and demonstrated DMA attacks via Thunderbolt 2 [20] and Thunderbolt 3 [21], but found them blocked by the IOMMU.

Compromised device firmware. Other work [59], [60], [62], [63] replaced the firmware of existing devices: allowing basic DMA attacks, but no more potent than those above. Vulnerabilities of NICs [17], [18] and Wi-Fi chips [5] allowed arbitrary code injection via crafted packets. DMA attacks were thus possible remotely. Much of this work cites IOMMU use as an effective mitigation.

Subverting the IOMMU. Most prior work on bypassing the IOMMU to carry out DMA attacks focused on exploiting architectural or boot-time configuration weaknesses, rather than OS-controlled IOMMU protections.

Lone Sang [37] hypothesized a number of IOMMU attack vectors without demonstrating them, and hence did not study OS behavior. They suggest modifying IOMMU page table structures, ACPI tables, and configuration registers. However, if an IOMMU is configured to correctly protect memory, peripheral devices are unable to modify these structures. The authors also suggested ATS support might allow bypass of the IOMMU. The only attack demonstrated in this paper is PCIe ID spoofing. Here, a malicious device spoofs the bus-device-function (BDF) ID of a legitimate NIC to inject malicious packets over DMA and poison the ARP cache of a victim machine. This is a PCIe weakness rather than an IOMMU one. Additionally, any NIC, even one without DMA access, can poison the ARP cache without needing to use this PCIe attack. Lone Sang later built a PCIe fuzzer that found the same vulnerability [38]. (We accidentally verified this weakness due to a bug when developing our platform, in our case spoofing interrupts.) They also implemented a keylogger that can read keyboard input and send keystrokes through peer-to-peer PCIe legacy IO requests. These are different from DMA and the IOMMU does not control this channel.

Other attacks have bypassed the IOMMU by racing the IOMMU setup at system boot. Many devices enable DMA at boot time, before the OS is launched and IOMMU enabled. Wojtczuk [68] modified ACPI tables during boot so the OS believed no IOMMU was present. ThunderGate [59] contains a firmware image that deploys a malicious PCI Option ROM containing code the system will execute – a result duplicated by others [50]. A weaponization of this technique is the Thunderstrike bootkit [27] and a similar attack was used by the CIA [24], and to attack Windows VBS [22]. Morgan [46], [47] enabled DMA by rewriting the IOMMU page tables while they were being created, before the IOMMU was fully enabled. Apple and recently Microsoft and partners have blocked such boot-time vulnerabilities.

Subsequent to our disclosure to vendors and to others in the community, some further IOMMU attacks have been published. Beniamini [10], [11] compromised the firmware of a Broadcom PCIe Wi-Fi chip using an over-the-air exploit. They used this device for DMA attacking the main OS, finding the ARM IOMMU was not used on his Android platforms. On iOS a custom IOMMU is used: they managed to exploit OS ring-buffer handling code to modify IOMMU mappings. Kupfer [34]'s masters thesis also reproduced some attacks similar to ours.

Other uses of the IOMMU. Additional work has been published on the security and performance considerations of IOMMU use by hypervisors to keep guest operating systems isolated, while giving them direct access to peripheral devices [52], [66], [67]. Hypervisor-related IOMMU work is both complementary and orthogonal to our work. Other work addresses IOMMU security but has a substantially different threat model [70].

XI. Disclosure

We disclosed these vulnerabilities to OS vendors starting in 2016 and have collaborated on mitigations over two years.

To mitigate our control-flow attacks, macOS 10.12.4 introduced a new code-pointer blinding feature, used when *mbuf* pointers are exposed via the IOMMU. This technique limits the effectiveness of attackers in injecting kernel pointers, but leaves open a number of data fields, including data pointers, that could leave the system exposed to further vulnerabilities.

After ongoing dialogue, Microsoft announced Kernel DMA Protection to enable IOMMU support in devices shipped with Windows 10 1803 (but not earlier firmware). They confirmed the vulnerabilities in this paper remain a concern, in particular, spatial vulnerabilities caused by page-sharing given that I/O memory is allocated from a general pool; they stated they will investigate this for subsequent releases. Critically, documentation for device-driver authors does not yet explain how to program robustly in the presence of a DMA-capable attacker. However, enabling the IOMMU will bring Windows into line with other platforms.

Linux's kernel security team considers our attacks within their threat model. They stated that, because Linux is used in many different environments, the problems are difficult to solve in the general case. For now, device authentication schemes remain the primary defense. Citing our disclosure, Intel's work in kernel 4.21 enables the IOMMU for Thunderbolt ports and disables ATS.

The FreeBSD Project indicated that malicious peripherals are not currently within their threat model for security response, although they were concerned about these attacks, and requested a copy of the paper for further review.

In conversation with one vendor of widely used notebook computers that do not currently include Thunderbolt 3, it was clear that IOMMU-bypass attacks via Thunderbolt were both within their threat model and would compromise intended security protections of their system. They stated that they would want to understand how to address these attacks before adding Thunderbolt to new product lines.

We are continuing our outreach to further vendors to establish whether these attacks are within their accepted threat models, and what mitigations they may see as appropriate. The widespread deployment of USB Type-C with Thunderbolt 3 increases the relevance of this work across a range of mobile devices, where physical access by attackers, as well as the promiscuous use of adapters, dongles, and chargers will be a growing concern.

XII. Future work

The vulnerability space available when the IOMMU is enabled is much richer than might have been expected. This presages a wide range of attacks. Most obviously, OSs unintentionally expose data structures with a rich semantic content to peripherals. Mining these structures for further vulnerabilities and exploit techniques is likely to be a profitable field.

Additionally, device behaviors are complex, and the IOMMU's exposure depends on those device behaviors. Further vulnerabilities are likely when a device opens new windows, extends existing windows, or keeps windows open longer.

A natural extension of our work would consider mobile and system-on-chip platforms. A system-on-chip comprises a complex mesh of parts with access to memory, only some of them labeled as processors. A control compromise in one part (e.g., an LTE radio, audio controller, or vision processing) affects safe operation of another (e.g., engine management or navigation of a vehicle). The IOMMU is supposed to keep these apart. In this space the OS, driver, and device stack are quite different, yet have faced little scrutiny.

XIII. Conclusion

We have demonstrated that the vulnerability space exposed to malicious peripherals can be broad and nuanced. Commodity operating systems have largely recognized the threat, and use the IOMMU to protect against DMA attacks.

However, it is not enough to simply isolate device memory, as implemented by MacOS in 2012 and now Windows 10 1803. Devices interact deeply with the device driver, and with other parts of the operating system. Like user processes using the system-call interface, device implementations communicating over the IOMMU-kernel shared-memory interface can stimulate complex vulnerable behavior. Moreover, malicious devices can mold themselves to target vulnerable interfaces, choosing the weakest software to attack.

The threat models of operating-system vendors have failed to take this into account. It is not sufficient to simply enable basic IOMMU protections in the PCIe bus framework and consider the job finished. Our findings show there is no defense in depth: the layers beyond, such as communication stacks and memory allocators, are not hardened against malicious devices. Close engagement with multiple OS vendors has led to marked improvement in IOMMU security and vulnerability mitigation through now-deployed software updates.

In a world where computers are smaller and more devices are externally pluggable (especially with the ubiquity of USB-C), malicious peripherals can be powerful adversaries.

Acknowledgment

The authors would like to thank Herbert Bos, Chris Dalton, Matt Evans, Antonio Galvan, Cristiano Giuffrida, Mark Hayter, Xeno Kovah, Greg Kroah-Hartman, Markus Kuhn, Ben Laurie, Steven Murdoch, Chris Riggs, Timothy Roscoe, and Benjamin Serebrin for their feedback and suggestions. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 ("CTSRD") and HR0011-18-C-0016 ("ECATS"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This work was also supported by EPSRC EP/R012458/1 ("IOSEC"). We also acknowledge Arm Limited and Google Inc. for their support.

References

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security analysis and enhancements of computer operating systems," NIST, Tech. Rep. NBSIR 76-1041, Apr. 1976.

[2] Advanced Micro Devices, Inc., "AMD I/O virtualization technology: (IOMMU) specification," Feb. 2015. [Online]. Available: http://support.amd.com/TechDocs/48882_IOMMU.pdf

[3] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "IOMMU: Strategies for mitigating the IOTLB bottleneck," in *6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, Jun. 2010.

[4] ARM Limited, "ARM System Memory Management Unit architecture specification," Jun. 2016.

[5] N. Artenstein, "Broadpwn: Remotely compromising Android and iOS via a bug in Broadcom's Wi-Fi chipsets," Jul. 2017. [Online]. Available: https://blog.exodusintel.com/2017/07/26/broadpwn/

[6] D. Aumaitre and C. Devine, "Subverting Windows 7 x64 kernel with DMA attacks," *HITBSecConf Amsterdam*, 2010.

[7] M. Becher, M. Dornseif, and C. N. Klein, "FireWire: all your memory are belong to us," *Proceedings of CanSecWest*, 2005.

[8] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–46.

[9] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn, "The price of safety: Evaluating IOMMU performance," in *OLS '07: The 2007 Ottawa Linux Symposium*, July 2007, pp. 9–20.

[10] G. Beniamini, "Over the air – vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices," Oct. 2017. [Online]. Available: https://googleprojectzero.blogspot.co.uk/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html

[11] ——, "Over the air: Exploiting Broadcoms Wi-Fi stack," Apr. 2017. [Online]. Available: https://googleprojectzero.blogspot.co.uk/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html

[12] A. Boileau, "Hit by a bus: Physical access attacks with Firewire," in *Ruxcon 2006*, May 2006.

[13] R. Breuk and A. Spruyt, "Integrating DMA attacks in exploitation frameworks," University of Amsterdam, Tech. Rep., 2012. [Online]. Available: http://www.delaat.net/rp/2011-2012/p14/report.pdf

[14] D. D. Clark, "An input/output architecture for virtual memory computer systems," Ph.D. dissertation, Massachusetts Inst. of Technology, 1974.

[15] J.-C. Delaunay, "Practical DMA attack on Windows 10," May 2018. [Online]. Available: https://www.synacktiv.com/posts/pentest/practical-dma-attack-on-windows-10.html

[16] M. Dornseif, "0wn3d by an iPod: Firewire/1394 Issues," in *Proceedings of PacSec Applied Security Conference 2004*, 2004. [Online]. Available: https://pacsec.jp/psj04/psj04-dornseif-e.ppt

[17] L. Duflot, Y.-A. Perez, and B. Morin, "What if you can't trust your network card?" in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 378–397.

[18] L. Duflot, Y.-A. Perez, G. Valadon, and O. Levillain, "Can you still trust your network card?" *CanSecWest/core10*, pp. 24–26, 2010.

[19] J. Fitzpatrick and M. Crabill, "Stupid PCIe tricks, featuring the NSA Playset," in *Proceedings of DEFCON 22*, 2014.

[20] U. Frisk, "Direct memory attack the kernel," in *Proceedings of DEFCON'24*, Las Vegas, USA, Aug. 2016.

[21] ——, "DMA attacking over USB-C and Thunderbolt 3," Oct. 2016. [Online]. Available: http://blog.frizk.net/2016/10/dma-attacking-over-usb-c-and.html

[22] ——, "Public FPGA based DMA attacking," in *34c3*, 2017.

[23] H.-C. Fu, P.-H. Wang, and C.-L. Yang, "Active forwarding: Eliminate IOMMU address translation for accelerator-rich architectures," in *55th Annual Design Automation Conference*, 2018.

[24] S. Gallagher, "New WikiLeaks dump: The CIA built Thunderbolt exploit, implants to target Macs," Mar. 2017. [Online]. Available: https://arstechnica.com/security/2017/03/new-wikileaks-dump-the-cia-built-thunderbolt-exploit-implants-to-target-macs/

[25] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *NDSS 2003*.

[26] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address

translation for accelerator-centric architectures," in *High Performance Computer Architecture (HPCA)*, Feb 2017.

[27] T. Hudson and L. Rudolph, "Thunderstrike: EFI firmware bootkits for Apple MacBooks," in *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, 2015, p. 15.

[28] Intel Corporation, "Thunderbolt technology: Technology brief," 2012.

[29] "Intel®Virtualization Technology for Directed I/O Architecture Specification," Intel Corporation, 2014.

[30] Intel Corporation, "Thunderbolt™ 3 – the USB-C that does it all," May 2015. [Online]. Available: https://thunderbolttechnology.net/blog/thunderbolt-3-usb-c-does-it-all

[31] A. Ionescu, "Getting physical with USB Type-C," in *Recon Brussels*, 2017. [Online]. Available: http://alex-ionescu.com/publications/Recon/recon2017-bru.pdf

[32] S. S. John, "Thundergate." [Online]. Available: http://thundergate.io/

[33] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," Apr. 2016. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[34] G. Kupfer, "IOMMU-resistant DMA attacks," Master's thesis, Technion - Israel Institute of Technology, May 2018.

[35] M. Larabel and M. Tippett, "Phoronix Test Suite v8.0.0." [Online]. Available: https://www.phoronix-test-suite.com/

[36] Linux kernel development community, "The Linux kernel user's and administrator's guide: Thunderbolt." [Online]. Available: https://www.kernel.org/doc/html/v4.13/admin-guide/thunderbolt.html

[37] F. Lone Sang, E. Lacombe, V. Nicomette, and Y. Deswarte, "Exploiting an I/OMMU vulnerability," in *5th International Conference on Malicious and Unwanted Software (MALWARE), 2010*.

[38] F. Lone Sang, V. Nicomette, and Y. Deswarte, "A tool to analyze potential I/O attacks against PCs," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 60–66, Mar 2014.

[39] C. Maartmann-Moe, "Inception." [Online]. Available: https://github.com/carmaa/inception

[40] M. Malka, N. Amit, M. Ben-Yehuda, and D. Tsafrir, "rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[41] M. Malka, N. Amit, and D. Tsafrir, "Efficient intra-operating system protection against harmful DMAs," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[42] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[43] A. Markuze, I. Smolyar, A. Morrison, and D. Tsafrir, "DAMN: Overhead-free IOMMU protection for networking," in *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[44] Microsoft, "Kernel DMA protection for Thunderbolt 3," Oct. 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/security/information-protection/kernel-dma-protection-for-thunderbolt

[45] R. Monroe, "Map of the Internet," 2006. [Online]. Available: https://xkcd.com/195/

[46] B. Morgan, É. Alata, V. Nicomette, and M. Kaâniche, "Bypassing IOMMU protection against I/O attacks," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, Oct 2016, pp. 145–150.

[47] B. Morgan, É. Alata, V. Nicomette, and M. Kaâniche, "IOMMU protection against I/O attacks: a vulnerability and a proof of concept," *J. Brazilian Computer Society*, vol. 24, no. 1, p. 2, Jan 2018.

[48] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *SIGCOMM 2018*, Aug. 2018.

[49] K. Nohl, S. Krißler, and J. Lell, "BadUSB – on accessories that turn evil," in *BlackHat U.S. 2014*.

[50] D. Oleksiuk, "Dmytro's rogue PCI-E device," Apr. 2017. [Online]. Available: https://firmwaresecurity.com/2017/04/07/dmytros-rogue-pci-e-device/

[51] PCI-SIG, "PCI Express base specification revision 3.0," Nov. 2010.

[52] G. Pék, A. Lanzi *et al.*, "On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment," in *9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS '14)*, 2014.

[53] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU scalably," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Jul. 2015.

[54] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen, "Locating system problems using dynamic instrumentation," in *2005 Ottawa Linux Symposium*, 2005, pp. 49–64.

[55] Red Hat Inc., "Red Hat achieves Common Criteria Security certification for Red Hat Enterprise Linux 7," Oct. 2016. [Online]. Available: https://www.redhat.com/en/about/press-releases/red-hat-achieves-common-criteria-security-certification-red-hat-enterprise-linux-7

[56] R. Sevinsky, "Funderbolt: Adventures in Thunderbolt DMA attacks," in *BlackHat USA*, 2013.

[57] O. Shwartz, A. Cohen, A. Shabtai, and Y. Oren, "Shattered trust: When replacement smartphone components attack," in *WOOT'17*, 2017.

[58] snare and rzn, "Thunderbolts and Lightning – very, very frightening," in *Proceedings of SyScan Singapore 2014*, Apr. 2014.

[59] S. St. John, "Thunderbolt: Exposure and mitigation," Fall 2013. [Online]. Available: http://www.thundergate.io

[60] P. Stewin and I. Bystrov, "Understanding DMA malware," in *9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, 2013.

[61] A. Trikalinou and D. Lake, "Taking DMA attacks to the next level," in *BlackHat USA*, Jul. 2017.

[62] A. Triulzi, "Project Maux Mk. II, I 0wn the NIC, now I want a shell," in *Proceedings of PacSec 2008*, 2008.

[63] ——, "The Jedi Packet Trick takes over the Deathstar," in *Central Area Networking and Security (CANSEC 2010)*, Mar. 2010.

[64] USB Implementers Forum, "USB Type-C cable and connector specification," Jul. 2017.

[65] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *Proceedings of the First USENIX Workshop on Offensive Technologies*, ser. WOOT '07, 2007.

[66] P. Willmann, S. Rixner, and A. L. Cox, "Protection strategies for direct access to virtualized I/O devices," in *USENIX 2008 Annual Technical Conference*, 2008.

[67] R. Wojtczuk and J. Rutkowska, "Following the White Rabbit: Software attacks against Intel VT-d technology," 2011. [Online]. Available: http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf

[68] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Another way to circumvent Intel Trusted Execution Technology," 2009.

[69] J. Yao, V. J. Zimmer, and S. Zeng, "A tour beyond BIOS: Using IOMMU for DMA protection in UEFI firmware," 2017. [Online]. Available: https://firmware.intel.com/sites/default/files/Intel_WhitePaper_Using_IOMMU_for_DMA_Protection_in_UEFI.pdf

[70] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein, "Understanding the security of discrete GPUs," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10, 2017.