Towards Efficient and Effective IOMMU-based Protection from DMA Attacks

by

Brett F. Gutstein

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Alan L. Cox, Chair
Professor of Computer Science and Electrical & Computer Engineering

Keith Cooper
L. John & Ann H. Doerr Professor of Computational Engineering; Professor of Computer Science and Electrical & Computer Engineering

Scott Rixner
Professor of Computer Science and Electrical & Computer Engineering

Houston, Texas
May, 2018
ABSTRACT

Towards Efficient and Effective IOMMU-based Protection from DMA Attacks

by

Brett F. Gutstein

Malicious actors can carry out direct memory access (DMA) attacks to compromise computer systems. In such attacks, peripheral devices abuse their ability to read and write physical memory independently of the CPU to violate the confidentiality or integrity of a system’s data. Relatively recently, commodity architectures have incorporated the I/O memory management unit (IOMMU), which allows the CPU to govern peripheral device memory access.

This thesis demonstrates that IOMMU usage in existing operating systems does not protect against DMA attacks effectively and comes with a prohibitively high performance cost. It introduces Thunderclap, a novel DMA attack platform used to carry out new attacks that completely compromise FreeBSD, macOS, Linux, and Windows, even with their current IOMMU-based protections enabled. It then presents and evaluates strategies for IOMMU usage that make strides towards efficient and effective protection from DMA attacks.
Acknowledgments

Thanks first and foremost to my advisor, Alan Cox, for being so supportive, knowledgeable, kind, and inspiring over the past couple of years. I became interested in research because of his influence, and if I have been able to enjoy any success in that arena, it is because of his guidance. Thanks to Keith Cooper for his valuable feedback and for supporting my education, both directly and indirectly, for the past decade. Thanks to Scott Rixner for his keen precision and for always making things interesting.

Thanks to my advisor at the University of Cambridge, Robert Watson, as well as the rest of the team I worked with during my time there: Theo Markettos, Colin Rothwell, Allison Pearce, Simon Moore, and Peter Neumann. I learned a lot from them, and they made much of the work in this thesis possible.

Thanks to my mom, my grandparents, my dad, Jill, Ari, Emily, and all of the other family and friends who have supported me always and during the past year in particular. Special thanks to Philip Taffet, who has kept the fire high with me for many years.
Contents

Abstract ii
List of Illustrations vii
List of Tables viii

1 Introduction 1
1.1 Contributions ............................................. 4
1.2 Organization ............................................. 6

2 Background 7
2.1 IOMMU design ........................................... 7
2.2 The DMA KPI ............................................. 9
2.3 The IOMMU driver ........................................ 10
   2.3.1 IOVA allocation .................................... 11
   2.3.2 Page table management ............................ 12
   2.3.3 IOTLB invalidation ................................. 13
2.4 Pluggable interconnects ................................. 13
2.5 IOMMU usage model ................................... 15

3 Related Work 18
3.1 Peripheral memory access attacks ...................... 19
   3.1.1 DMA attack platforms .............................. 19
   3.1.2 Compromised device firmware ...................... 21
   3.1.3 Subverting the IOMMU ............................. 23
3.2 IOMMU usage and architecture ....................... 25
3.2.1 Improvements in software ........................................ 25
3.2.2 Hardware designs .................................................. 27
3.3 Memory encryption ..................................................... 28

4 OS-level IOMMU Protection Vulnerabilities ......................... 30
  4.1 OS-level vulnerabilities ............................................ 30
  4.2 Attacker model ....................................................... 34
  4.3 The Thunderclap attack platform .................................. 35
  4.4 Example attacks ..................................................... 38
    4.4.1 FreeBSD ....................................................... 39
    4.4.2 macOS ......................................................... 45
    4.4.3 Linux .......................................................... 49
    4.4.4 Windows ....................................................... 51
  4.5 Discussion .......................................................... 53

5 Improvements in IOMMU Usage ........................................ 55
  5.1 A novel IOMMU driver design ...................................... 55
    5.1.1 Vmem .......................................................... 56
    5.1.2 Page table management ......................................... 57
    5.1.3 Per-core caching ............................................... 59
    5.1.4 Page table walks ................................................ 61
    5.1.5 Discussion and evaluation ...................................... 62
  5.2 DMA KPI changes .................................................... 73
    5.2.1 Asynchronous unmap ............................................ 73
    5.2.2 Sendfile optimization .......................................... 76
  5.3 Memory management techniques .................................... 78
    5.3.1 Shadow buffers ................................................ 79
    5.3.2 Device memory allocators ..................................... 82
  5.4 Discussion .......................................................... 83
6 Conclusions 85

6.1 Future work ................................. 87

6.1.1 Further improvements in IOMMU usage .... 87

6.1.2 IOMMU hardware designs ............... 87

Bibliography 89
## Illustrations

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Schematic diagram of the IOMMU and MMU in current architectures.</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Diagram showing how IOVAs and PCIe BDF numbers are used in IOMMU translation.</td>
<td>8</td>
</tr>
<tr>
<td>4.1 Example of the protection granularity IOMMU usage vulnerability.</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Timeline of events for the non-strict IOTLB invalidation IOMMU usage vulnerability.</td>
<td>33</td>
</tr>
<tr>
<td>4.3 Schematic diagram of the Thunderclap stack.</td>
<td>36</td>
</tr>
<tr>
<td>4.4 Diagram of transmit and receive descriptor rings.</td>
<td>40</td>
</tr>
<tr>
<td>5.1 An example vmem arena hierarchy.</td>
<td>60</td>
</tr>
</tbody>
</table>
## Tables

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Single-core netperf TCP_STREAM performance results.</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>Multicore netperf TCP_STREAM results.</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>Netperf TCP_RR results.</td>
<td>72</td>
</tr>
<tr>
<td>5.4</td>
<td>Single-core receive-side asynchronous unmapping evaluation.</td>
<td>76</td>
</tr>
<tr>
<td>5.5</td>
<td>Shadow buffering cache effect evaluation.</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Malicious actors can carry out direct memory access (DMA) attacks from peripheral devices to compromise computer systems. I/O memory management unit (IOMMU) hardware allows the CPU to govern peripheral memory access and can protect against DMA attacks in theory. However, this thesis demonstrates that IOMMU usage in existing operating systems does not protect against sophisticated DMA attackers and comes with a prohibitive performance cost. The overarching goal of this work is to make strides towards a system that provides better protection from DMA attacks using existing IOMMU hardware at a low enough performance cost that it can be enabled by default in commodity operating systems.

Early peripheral devices, such as network cards, graphics cards, and nonvolatile storage controllers, gave computers a relatively simple interface to the outside world. Processors could read and write registers on peripheral hardware controllers to send and receive data from external sources. In particular, processors had to transfer data between peripheral devices and main memory because the controllers could not access memory directly.

As peripheral devices became faster and more sophisticated, processor-bound data transfer became a significant bottleneck. It limited the performance of high-throughput peripherals and also consumed significant CPU time. In response, commodity computer architectures adopted direct memory access (DMA), a feature that allows peripheral devices to access system physical memory independently of the processor.
DMA greatly improved I/O performance and CPU availability but also changed the trust model for peripheral devices. Before DMA, a malicious or misbehaving peripheral could, at worst, present incorrect data to the processor. However, because DMA allows peripheral devices to read and write any physical memory location, DMA-enabled peripherals essentially have complete control over the system and must be trusted not to act maliciously. When a peripheral device abuses DMA to violate the confidentiality or integrity of information stored in memory, it is said to have carried out a DMA attack. DMA attacks from purpose-built malicious peripheral devices or legitimate devices with compromised firmware have been widely used by hackers and the intelligence community to unlock password-protected systems, steal encryption keys, and dump the contents of a system’s memory [5, 7, 10, 16–20, 22, 50].

Relatively recently, commodity architectures have adopted system-level I/O memory management units, such as Intel’s VT-d [28] and AMD’s AMD-Vi [2]. IOMMUs impose CPU-controlled address-space virtualization on peripheral devices similarly to how MMUs impose it on processes. They initially saw widespread adoption to ensure guest operating system isolation and provide pass-through access to peripheral devices in virtualized systems. In such a pass-through scheme, a peripheral device’s DMA transactions are remapped by the same translation rules as a guest operating system’s physical addresses. The guest can then program the device’s DMA transactions without making a hypervisor call to determine appropriate host physical addresses.

In theory, IOMMUs could be used to fully protect any computer system, virtualized or not, from DMA attacks by restricting peripheral device access to memory. Indeed, as we will see in Chapters 2 and 3 an IOMMU usage model that reflects the state of IOMMU-based protection in existing operating systems is sufficient to protect against all published DMA attacks known at the time of writing. However, Chapter 4 shows
that IOMMU-based protection from DMA attacks in existing operating systems still suffers from a number of issues in practice. Security vulnerabilities allow new and sophisticated DMA attacks to compromise target systems, and current techniques for using the IOMMU involve performance costs that greatly reduce I/O throughput and processor availability when driving high-performance peripherals like 40Gbit/s NICs and NVMe drives [9, 34, 45]. As a result, many existing systems have made design choices that further sacrifice IOMMU-based security for performance. Even so, the current two highest-performing designs cause up to a 25 and 30 percent reduction in throughput and up to a 60 and 80 percent increase in CPU utilization when driving a 40Gbit/s NIC compared to the no-IOMMU case [37]. Because of this performance cost, most commodity operating systems disable their IOMMU support by default. macOS is the only known commodity operating system to enable its IOMMU support by default.

The rise of pluggable hardware interconnects has greatly increased the risk of DMA attacks. DMA transactions are carried out over the system interconnect (usually called the system bus), canonically PCIe [42] in current systems. In the past, the system bus was only exposed to peripheral devices plugged in to the motherboard inside of a machine’s case. Interconnects like Thunderbolt [29], which is increasing in popularity and adoption, expose PCIe to devices that can be hotplugged in to a port on the outside of a machine even while that machine is running. The newest version of Thunderbolt also features power, video, and PCIe connections over a single port. Section 2.4 demonstrates how these characteristics make it easier for malicious actors to gain access to the system bus and carry out DMA attacks.

The main points of this thesis are as follows. It demonstrates that current IOMMU usage does not provide sufficient protection from DMA attacks by introducing Thun-
derclap, a novel DMA attack platform used to carry out new attacks that completely compromise FreeBSD, macOS, Linux, and Windows, even with their current IOMMU-based protections enabled. It further demonstrates that the performance cost of IOMMU use is prohibitive and evaluates novel strategies for using the IOMMU that make strides towards efficient and effective protection from DMA attacks.

1.1 Contributions

This thesis first formulates a model for how existing operating systems utilize the IOMMU to protect against DMA attacks. It demonstrates that this model is sufficient to protect against all known DMA attack strategies published in the literature. It also defines a realistic model for the capabilities of DMA attackers that reflects recent trends in hardware interconnects, synthesis, and emulation.

This thesis then introduces Thunderclap, a novel FPGA-based DMA attack platform that instantiates this attacker model. It can emulate a wide range of devices and appear to the target machine as a legitimate and innocuous peripheral. However, it simultaneously runs malicious attack code that can generate arbitrary DMA transactions. The Thunderclap platform was developed in collaboration with A. Theodore Markettos and Colin Rothwell [47] from the University of Cambridge, taking into account previous insights by Allison Pearce [43] from the University of Cambridge. I developed novel attack strategies using Thunderclap to exploit previously unexploited vulnerabilities in operating system IOMMU usage. This thesis demonstrates employing these new strategies to completely compromise FreeBSD, macOS, Linux, and Windows, even with their IOMMU protections enabled.

Next, this thesis introduces and evaluates a novel IOMMU driver design that improves the performance of functions that grant and revoke memory access to and
from peripheral devices. In existing systems, these functions are called before and after each DMA transaction for devices like high-performance network cards. These network cards can operate on the order of millions of transactions per second, so improvements to relevant IOMMU driver functions have a significant impact on both I/O and CPU performance. Existing IOMMU driver designs separate the three main tasks of address allocation, page table management, and cache invalidation performed by the driver into distinct subsystems. This thesis develops a novel address allocator design that implicitly manages page table structure, effectively combining those two subsystems. The combined design offers a number of performance benefits relative to existing systems.

This thesis then presents novel kernel programming interface (KPI) functions by which peripheral device drivers can invoke IOMMU driver operations. Existing systems offer synchronous interfaces exposing expensive operations that must often be called in performance-critical threads such as interrupt handlers to ensure protection from DMA attacks. These restrictive interfaces have led existing systems to adopt implementations that reduce the performance cost of IOMMU usage but introduce a security vulnerability. New KPI functions that support asynchrony, clear semantics, and specialized support for important use cases can similarly reduce the performance cost of IOMMU usage while eliminating this vulnerability.

Finally, this thesis evaluates both existing and new strategies for managing memory that is exposed to peripheral devices. Shadow buffering is a technique proposed by Markuze et al. in 2016 that involves maintaining a pool of perpetually accessible memory for each peripheral device in the system [37]. With shadow buffering, when a device is granted access to a certain buffer in memory, that buffer is copied into the device’s pool. When the device’s access is revoked, the buffer is copied from the
pool back to its original location. Shadow buffering addresses security vulnerabilities present in existing systems, but it still comes with a significant performance cost. This thesis evaluates shadow buffering and identifies negative effects on cache performance that have not previously been demonstrated. This thesis then describes an alternative strategy of maintaining per-device pools of accessible memory that directly satisfy allocation requests, which can also address security vulnerabilities but at a potentially lower performance cost.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 presents relevant background information about the IOMMU and IOMMU usage in existing operating systems. Chapter 3 discusses related work. Chapter 4 presents previously unexploited operating system-level vulnerabilities in IOMMU usage. It also describes the Thunderclap platform and demonstrates how Thunderclap was used to exploit the above vulnerabilities and completely compromise existing operating systems via DMA attacks. Chapter 5 describes and evaluates strategies for addressing security vulnerabilities in and reducing the performance cost of IOMMU protections. Chapter 6 concludes the thesis and discusses future work.
Chapter 2

Background

This chapter outlines the design of IOMMU hardware and how IOMMUs are used in existing operating systems. It defines a model for operating system IOMMU usage and discusses how hotpluggable hardware interconnects increase the threat of DMA attacks against real systems. Both this chapter and the rest of this thesis use Intel’s VT-d \[28\] as the basis for discussion and evaluation because it is the most widely studied IOMMU. AMD-Vi is similar \[2\], but other IOMMUs may have different designs.

2.1 IOMMU design

As shown in Figure \[2.1\] the IOMMU imposes address space virtualization on peripheral devices like the MMU does on processes. In particular, the IOMMU sits between peripheral devices and main memory and intercepts DMA transactions. It translates the I/O virtual addresses (IOVAs) contained in DMA requests and checks read and write permissions, allowing transactions to proceed only if the appropriate translation entries are present and the permission checks succeed.

VT-d \[28\] supports a standard four-level page table structure for translating a 48-bit virtual address to a 4KB physical page, as shown in Figure \[2.2\]. Its page table also supports standard permissions flags and 2MB and 1GB superpages. There is an additional two-level radix tree structure, which this thesis will refer to as the domain table, that maps devices to the four-level page tables that govern their respective
Figure 2.1: Schematic diagram of the IOMMU and MMU in current architectures.

Figure 2.2: Diagram showing how IOVAs and PCIe BDF numbers are used in IOMMU translation.
virtual address spaces. Each DMA request is required to present the IOMMU with a 16-bit value identifying the device that originated the request. This source identifier is determined in an implementation-specific manner depending on the system interconnect and is used to traverse the domain table. In this way VT-d supports isolating each peripheral device in its own virtual address space.

Existing commodity systems canonically use PCIe [42] as the system interconnect. PCIe is a point-to-point packet-switched interconnect with a layered hardware protocol that features a physical layer, a data link layer, and a transaction layer. In a system with PCIe, when a device makes a DMA read or write request, that request is sent encapsulated in PCIe transaction layer packets (TLPs). These TLPs contain a 16-bit bus/device/function (BDF) number that uniquely identifies the originating device. This BDF number is used as the IOMMU source identifier, as shown in Figure 2.2.

The IOMMU also features an I/O translation lookaside buffer (IOTLB) that caches translation information like the MMU’s TLB. In particular, the IOTLB stores mappings between source identifier-IOVA pairs and their corresponding physical addresses. This caching greatly speeds up the IOVA translation process but introduces other performance concerns. In particular, invalidating an entry in the IOTLB is an extremely expensive operation that has been reported to take about 2100 cycles to complete for VT-d [35]. Details like the size and number of IOTLBs present in an IOMMU are left as implementation-specific in the VT-d specification.

2.2 The DMA KPI

In most existing systems, peripheral device drivers make use of a kernel programming interface (KPI) that this thesis will refer to as the DMA KPI to grant and revoke memory access to and from their respective devices. DMA KPIs tend to consist of two
abstract functions. In particular, the \texttt{map} function grants a peripheral device access to a specified region of memory with specified permissions, and the \texttt{unmap} function revokes access to a specified region of memory from a peripheral. Generally, the IOMMU is used in one concrete implementation of these abstract functions. Other concrete implementations could involve no-ops (to be used when the IOMMU is disabled) or memory-copy operations (to be used when the IOMMU is disabled and legacy 32-bit PCIe devices need to access memory above the 4GB boundary). The concrete implementation used by an operating system is typically chosen at boot-time by an administrator-specified option.

It should be noted that most systems do not enable their IOMMU support by default because of its significant performance impact. FreeBSD, Linux, and Windows all implement some form of support for IOMMUs, but they do not enable it out of the box. macOS is the only popular commodity operating system that does make use of an IOMMU, Intel’s VT-d, by default.

### 2.3 The IOMMU driver

Functions defined by the IOMMU driver are used in the IOMMU-related concrete implementation of the DMA KPI. The IOMMU driver is responsible for managing the translation structures for each peripheral device’s virtual address space, which consists of three main tasks: allocating IOVAs, managing the IOMMU page tables, and invalidating IOTLB entries when appropriate. On a \texttt{map} call, the IOMMU driver must allocate a range of IOVAs for the passed-in buffer then add the corresponding translations to the IOMMU page table for the appropriate device before returning the IOVA at the start of the range. On an \texttt{unmap} call, the driver must (1) remove the relevant translations from the device’s IOMMU page table, (2) invalidate the
corresponding IOTLB entries, and (3) free the buffer’s IOVA range before returning.
Note that these steps must be performed in the order listed above to ensure correctness.
Because the IOMMU-related implementations of \texttt{map} and \texttt{unmap} involve expensive
operations and are often called in throughput-critical code paths millions of times per
second, they have a significant effect on system performance.

Existing IOMMU drivers split the three main tasks described above into distinct
subsystems. This thesis now describes these subsystems in detail, including their
implementation details in macOS [4], Linux [59], and FreeBSD [8].

2.3.1 IOVA allocation

IOVA allocation is not strictly necessary for the IOMMU to provide protection from
DMA attacks. The \texttt{map} operation could simply use the passed-in buffer’s physical
address as its IOVA and bypass allocation completely, as suggested in [45]. However,
IOVA allocation provides a number of important benefits that have led all major
operating systems to adopt it. It packs page table mappings together, which results
in a significantly smaller page table than one whose mappings use sparsely distributed
physical addresses. It makes non-physically contiguous buffers virtually contiguous to
peripheral devices, reducing the number of DMA transactions required to perform
I/O. It allows mappings with differing permissions to be made to the same physical
buffer. And it also allows legacy PCI devices that cannot address memory over the
4GB boundary to interact with all of system memory.

To perform IOVA allocation, Linux and FreeBSD use red-black tree allocators that
keep track of allocated ranges. These allocators perform linear scans of the trees to
find appropriate free ranges to satisfy allocation requests and logarithmic-time tree
update operations. Linux’s allocator is equipped with a heuristic that usually finds
appropriate free ranges in constant time, bypassing the linear scan, and its allocator packs IOVA allocations as close together as possible to minimize page table memory usage [45]. Linux is also the only existing system to feature a per-core caching layer on top of its IOVA allocator, which remedies the significant performance cost of using allocators protected by a global lock on multicore machines. macOS uses a buddy allocator for most IOVA allocations and a red-black tree for allocations that are too large for the buddy allocator.

### 2.3.2 Page table management

The page table management subsystem is responsible for (1) updating entries in the page table as buffers are mapped and unmapped and (2) dynamically adding and removing page table pages, which are the pages of memory that make up the page table’s structure, as needed. To add an entry, it must first traverse the page table and add any page table pages necessary to support the entry’s translation. After removing an entry, it should check for and remove any page table pages that are no longer necessary to minimize the page table’s memory footprint.

The main difference among existing page table management implementations is multicore support. FreeBSD and macOS perform page table modifications under a global lock, which greatly reduces performance on multicore machines. Linux uses atomic operations to add page table structure and takes advantage of the fact that allocated IOVA ranges are pairwise disjoint to update page table entries without synchronization. However, reclaiming page table pages that are no longer needed requires complex synchronization without a global lock, so Linux is not able to reclaim unused page table memory [45].
2.3.3 IOTLB invalidation

IOTLB invalidation is necessary as part of the unmap operation to ensure that stale translations do not remain in the cache. As noted above, it is an extremely expensive operation performed by the IOMMU hardware that has been reported to take about 2100 cycles to complete [35]. The cost of IOTLB invalidation has led to the development of IOTLB invalidation schemes with non-strict unmapping semantics. With non-strict semantics, unmap operations remove an entry from the page table and initiate an IOTLB invalidation but do not wait for it to complete before returning. The corresponding IOVA range is freed asynchronously once the invalidation has completed. Batched invalidation is a scheme with non-strict semantics that batches multiple unmap operations and performs a single global IOTLB invalidation for all of them. While batching greatly improves performance, all non-strict IOTLB invalidation schemes negatively impact security, as we will see in Section 4.1. On the other hand, a strict unmapping semantics requires that the unmap operation wait for IOTLB invalidation to complete before freeing the corresponding IOVA range and returning. Currently, no design for a high-performance IOTLB invalidation subsystem with strict unmapping semantics has been published.

FreeBSD, Linux, and macOS all use non-strict IOTLB invalidation by default when their IOMMU support is enabled. FreeBSD and Linux allow the administrator to specify that strict IOTLB invalidation be used via a kernel setting.

2.4 Pluggable interconnects

Pluggable hardware interconnects expose the system bus, canonically PCIe [42], to peripheral devices via ports on the outside of a machine’s case. The work in this
thesis focuses on Thunderbolt \cite{29}, an interconnect that is increasingly being adopted in modern machines, but others such as FireWire \cite{1} have been used by machines in the past. Access to the system bus is necessary to make DMA requests and was historically only available to peripheral devices plugged in to the motherboard on the inside of a machine’s case. Thunderbolt increases the risk of DMA attacks by granting system bus access to external devices. With operating system support, Thunderbolt also features hotplugging, which grants system bus access to external devices that are plugged in even while the system is running.

It may not always be feasible for a malicious actor to gain access to the inside of a target machine’s case. However, it is much more likely that they could access an external Thunderbolt port on an unattended laptop or workstation. In such a scenario, a few seconds of access are enough for the attacker to install a malicious payload or dump the contents of memory via a DMA attack. Further complicating the situation is that the newest version of Thunderbolt, Thunderbolt 3, offers PCIe access, video output, and power delivery over the USB-C port. Trojan charging stations, projectors, and bridges or dongles that use the USB-C port could exploit Thunderbolt PCIe access to carry out DMA attacks covertly while performing their advertised functions. Thunderbolt thus adds a rich attack surface that makes the threat of a DMA attack much more significant in practice.

Thunderbolt offers access control that is supported to varying degrees by different operating systems. It features a number of security levels: none, which allows all Thunderbolt devices to connect to the system bus freely; user, which allows the user to decide whether to allow a device to connect based on a prompt; secure, which extends user mode by making devices store a random key assigned to them when they are first allowed and present it during subsequent interactions (to prevent spoofing
the device ID of already-allowed devices); and dponly, which allows devices to connect and tunnel DisplayPort but rejects all access to the system bus. The security level is set in the BIOS, which generally defaults to the user level. The operating system is responsible for allowing users to make access control decisions and for reporting those decisions to the Thunderbolt driver.

Thunderbolt access control may successfully defeat an attacker that tries to hotplug a malicious device into a machine left unattended. However, it would not protect against the Trojan device attacks discussed above, since in those scenarios the user allows access to the system bus for a malicious device that they believe is innocuous. For example, a user may insert and approve a Thunderbolt dongle connected to a projector at a conference venue without knowing that the dongle also tunnels PCIe to a DMA attack device. Thunderbolt access control, even when supported correctly and employed by knowledgable users, does not completely prevent attackers from gaining access to the system bus.

Details about hotplugging and access control support across operating systems are presented in Chapter 4.

2.5 IOMMU usage model

We now formulate a model for IOMMU usage in existing operating systems. Such a model will be useful in reasoning about operating system-level IOMMU usage vulnerabilities and the relative power of DMA attack strategies. In particular, we will see in Chapter 3 that this IOMMU usage model protects against all previous DMA attacks published at the time of writing. However, systems that instantiate this usage model are not completely free from vulnerabilities. Chapter 4 demonstrates exploiting previously unexploited vulnerabilities that can compromise such systems.
The properties of our IOMMU usage model are as follows: (1) The IOMMU is enabled before DMA during boot, (2) devices cannot disable the IOMMU during boot, (3) system bus features that allow IOMMU bypass are not enabled, and (4) device memory access is granted and revoked purposefully by device drivers. Properties (1) through (3) capture the idea that the operating system uses the IOMMU in a way that is free of configuration errors. If property (1) did not hold, devices would have unrestricted access to physical memory, including regions that control IOMMU translations, before the IOMMU is enabled. Property (2) prevents devices from, for example, presenting malicious PCIe Option ROMs during boot that disable IOMMU protections. Property (3) ensures that features that allow peripheral devices to bypass the IOMMU, such as PCIe Address Translation Services, are not enabled. Property (4) is less specific and captures the idea that devices can only access memory that is exposed to them by device drivers in the course of normal operation. For example, it precludes using the IOMMU in a way that perpetually exposes all of physical memory to all devices. It ensures that devices can only access pages of memory that have been exposed to them by their drivers for the purpose of communication.

This model represents an idealized version of IOMMU usage in existing systems. In practice, such systems may have configuration errors that violate properties (1), (2), or (3). However, fixing these errors is a relatively straightforward matter of engineering and managing configuration settings. We will see in Chapter 4 that existing systems may also expose memory to devices with unnecessary access permissions or fail to isolate devices in their own I/O virtual address spaces. These characteristics do not violate property (4), nor do they change which types of DMA attack strategies are effective and ineffective against the systems. In other words, previously published DMA attacks are still ineffective against such systems, and the DMA attacks described in this
thesis are still effective. However, giving devices unnecessary access permissions and failing to isolate devices in their own I/O virtual address spaces make it significantly easier for malicious peripherals to carry out DMA attacks in practice and thus should be avoided.
Chapter 3

Related Work

Work related to this thesis has focused on peripheral DMA attacks, IOMMU-based memory protection, and memory encryption. Compared to the work of this thesis, previous memory access attacks have not directly bypassed OS-imposed IOMMU access control and are defeated by systems that instantiate our IOMMU usage model. Furthermore, existing IOMMU-based protection schemes either trade off security for performance or incur unacceptable performance costs. Memory encryption is a method of protection from malicious peripherals that does not rely on the IOMMU. While potentially promising, it has not been shown to provide general-purpose protection from DMA attacks in commodity systems.

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 \cite{39, 44, 57, 58}. The work presented in this thesis focuses on protecting from malicious peripherals at the operating system level. Hypervisor-related IOMMU work is complementary yet orthogonal and will not be discussed in detail. Work that discusses security against attackers with capabilities beyond those of the model presented in Section 4.2, such as the capability to patch or control an OS device driver \cite{60}, will also not be discussed.
3.1 Peripheral memory access attacks

A significant amount of work exploring peripheral memory access attacks has been published. One line of investigation involves the construction of specialized platforms that carry out simple DMA attacks over various hardware interfaces. These DMA attacks involve attack devices that do not emulate the functionality of a legitimate peripheral. They would not work against systems that instantiate our IOMMU usage model because no device drivers would attach and expose regions of memory to them. Another line of investigation involves compromising the firmware of existing peripheral devices to carry out similarly simple attacks. These attacks rely on scanning all of physical memory for sensitive data and thus would not work against our IOMMU usage model either. Some previous studies have also described subverting IOMMUs to carry out DMA attacks, but they rely on configuration errors and do not actually exploit vulnerabilities in operating system-level IOMMU use.

3.1.1 DMA attack platforms

A number of DMA attack platforms have been developed to exploit unrestricted memory access in systems that do not use IOMMUs. These platforms can attack many operating systems over various hardware interfaces. They can steal sensitive data like encryption keys, violate any kernel security policies, and take complete control of a target machine. However, they do not work against systems that use the IOMMU in accordance with our model.

Dornseif et al. first abused the FireWire interface’s memory access functionality to carry out DMA attacks in 2004 [16] and 2005 [7]. Their attack device was an iPod running Linux that could send arbitrary FireWire memory access requests. They demonstrated that the device could connect via FireWire to a victim running macOS,
FreeBSD, or Linux and extract live memory dumps or modify the system state. Boileau extended this scheme in 2006 to work against Windows XP by having the attack device spoof a trusted FireWire Configuration Status Register [10]. He also demonstrated that an attacker could recover disk encryption passphrases from the victim’s memory. Breuk and Spruyt discussed integrating DMA over FireWire as an attack vector for the Metasploit framework [12]. Inception is a popular open-source software library that allows an attack machine with a FireWire interface to carry out DMA attacks against a connected victim machine [33]. It supports unlocking, implanting Metasploit payloads, and memory dumping, but none of these features works against a system that uses an IOMMU. In fact, all of the attacks discussed above use platforms that depend on unrestricted memory scans and do not emulate legitimate peripheral devices.

Aumaitre and Devine developed a DMA attack platform targeting the CardBus (PCMCIA) interface in 2010 [5]. The platform combined a MIPS processor and PCI IP core on an FPGA to send arbitrary PCI memory access requests via an attached CardBus port. The designers demonstrated its ability to unlock and execute arbitrary code on a machine running a 64-bit Windows 7 kernel. While more architecturally sophisticated than previous attack devices, this platform did not pose a greater threat. Its access was still limited to memory below the 4GB boundary by the 32-bit CardBus interface. It also did not emulate a legitimate peripheral and relied on memory scans to carry out its exploits. Thus it would not work against a system that instantiates our IOMMU usage model.

Sevinsky reverse engineered a Thunderbolt disk and Ethernet adapter in 2013 [50]. He discovered that the Ethernet adapter could be taken apart to expose wires that transmit arbitrary PCIe traffic through its Thunderbolt interface. He further suggested that these wires could be connected to a PCIe attack platform to carry out DMA attacks.
over Thunderbolt but did not demonstrate such a setup. However, his suggestion hinted at the fact that DMA attack platforms can be disguised as legitimate pluggable peripherals.

FitzPatrick and Crabill presented SLOTSCREAMER, an attack device based on a PCIe to USB adapter that allowed a host connected via USB to control PCIe packets sent to a victim machine, in 2014 [19]. They highlighted the feasibility of real-world DMA attacks by demonstrating their device through a PCIe to Thunderbolt bridge and cabling that disguised it as a projector. While significantly cheaper and architecturally simpler than previous FPGA-based attack devices, SLOTSCREAMER required an external machine (USB host) to function. And though the platform supported PCIe, it was still limited to 32-bit addresses for DMA. In 2016, Frisk augmented the same PCIe to USB adapter with improved firmware and software to increase its memory access throughput [20]. He demonstrated the device against Linux and Windows 10 with a PCIe to ExpressCard bridge and against macOS with a Thunderbolt 2 bridge. However, since macOS had started to enable the Intel VT-d IOMMU by default, the attacks against macOS only worked after the IOMMU was manually disabled. Frisk later attempted to attack an Intel machine with his device over a Thunderbolt 3 bridge [21]. He ran into Thunderbolt 3 access controls that prevented the setup from working, but he did not attempt to subvert them.

3.1.2 Compromised device firmware

Some previous work has also discussed modifying or exploiting firmware on existing peripheral devices to turn them malicious and compromise a target system. This strategy does not require the construction and use of specialized attack platforms, but it is not necessarily more effective than the attack platform strategy. In many
cases, it still requires physical access to a system’s peripheral devices, and previously published DMA attacks carried out by compromised devices are also defeated by our IOMMU usage model.

Triulzi presented a proof of concept in 2008 that involved modifying the firmware of a PCI-connected NIC and GPU in the same system [54]. The NIC would function normally but forward packets containing a magic number to the GPU over the PCI bus. The GPU would then interpret commands contained in the packets to perform DMA attacks. However, the proof of concept relied on manual modification of existing firmware, and no DMA attacks were demonstrated. In 2010, Triulzi extended the proof of concept to allow a malicious NIC to infect other NICs on the system via a remote firmware update feature [55]. Again, no DMA attacks were demonstrated. Though an IOMMU cannot police traffic between peripheral devices to prevent the kind of communication discussed above, it can protect the integrity of memory so that compromised devices are not able to violate system security policies.

ThunderGate is a suite of tools for modifying firmware on certain Ethernet controllers, and it focuses particularly on the controller present in Apple Thunderbolt Gigabit Ethernet adapters [51]. It contains a firmware image that enables the controllers to carry out DMA attacks that our IOMMU usage model protects against.

Stewin and Bystrov discussed modifying the firmware of Intel’s Management Engine processor to create a DMA attack-based keylogger called DAGGER [52]. It would also be thwarted by our IOMMU usage model.

Duflot et al. discovered a vulnerability in the firmware of certain Broadcom NICs that allowed a remote attacker to execute arbitrary code on the NICs by sending specific network packets [18]. An attacker exploiting this vulnerability could remotely carry out DMA attacks or rewrite device firmware to be malicious. This work escalated
the threat of DMA attacks by proving that they can be carried out without physical access to a target machine. Duflot et al. later published a paper discussing detection methods for such attacks [17]. Both works cited IOMMU use as an effective mitigation strategy.

### 3.1.3 Subverting the IOMMU

Prior work on subverting the IOMMU to carry out DMA attacks has focused on exploiting boot time or configuration vulnerabilities rather than OS-level IOMMU protections. These attacks work only against systems that do not instantiate our IOMMU usage model.

In 2010, Sang et al. suggested a number of attack vectors that could potentially compromise VT-d protections [49]. The list of vectors included modifying IOMMU page table structures, ACPI tables, and configuration registers in memory via DMA. However, if an IOMMU is enabled correctly from boot time, peripheral devices are never able to modify the structures listed above. The authors also suggested that devices could use device IOTLB support to bypass the IOMMU. Device IOTLB support is a name given to features of system buses that allow peripheral devices to supply their own IO virtual address translations, bypassing the IOMMU by design for performance reasons. Obviously, a malicious peripheral could use this feature to bypass the IOMMU. Device IOTLB support violates property (3) of our IOMMU usage model and should be disabled by all systems that value security. The final and only demonstrated attack vector presented in this paper is BDF number spoofing. In the proof of concept exploit, a malicious device spoofs the BDF number of a legitimate NIC to inject malicious network packets via DMA and poison the ARP cache of a victim machine. However, if an IOMMU is configured to correctly protect sensitive
structures in memory, BDF number spoofing does not escalate the privileges of a malicious peripheral. In other words, the worst thing such a peripheral could do is present incorrect data to the system, which it could do even without BDF number spoofing. Application-level attacks such as ARP cache poisoning should be protected against at the application level.

Sang et al. later used an FPGA platform to implement a keylogger peripheral device that can read keyboard input and send keystrokes through a system’s I/O address space, which is unrelated to the concept of I/O virtual address spaces discussed in this thesis [48]. This attack works even against systems that use the IOMMU effectively because IOMMUs do not control access to the I/O address space. However, it is not a DMA attack since it does not involve peripheral access to main memory. Different mitigation strategies to protect from attacks of this nature should be explored.

Other attacks have bypassed the IOMMU by exploiting configuration problems during system boot. Morgan et al. discovered that in certain systems, DMA was enabled for peripheral devices before the IOMMU was properly configured during the boot process [40]. Using DMA, they were able to modify IOMMU page tables in memory during boot to allow unrestricted peripheral memory access even after the IOMMU was enabled. This attack is a demonstration of the one Sang et al. posited in 2010. The ThunderGate suite also contains a firmware image that compromises VT-d during boot by supplying a malicious PCI option ROM [51]. Hudson demonstrated similar option ROM attacks in 2014 and 2015 [25, 26]. These attacks rely on violations of our IOMMU usage model properties.
3.2 IOMMU usage and architecture

Some prior work has also focused on using the IOMMU to protect memory. Advancements have been made in software that determines how the operating system exposes memory to devices via the IOMMU, with proposed designs offering varying levels of performance and security. New IOMMU hardware designs have also been suggested to improve performance and simplify the tasks that an IOMMU driver has to perform.

3.2.1 Improvements in software

In 2007, Ben Yehuda et al. acknowledged the performance cost of using IOMMUs and presented a number of software optimizations that sacrifice security for performance [9]. One suggestion involved allocating IOVAs using a next-fit policy and only invalidating the IOTLB once the entire address space has been used and the allocator rolls over. Others included pre-allocating memory to be exposed to peripherals, mapping and unmapping memory from peripherals’ address spaces in large batches, and never unmapping memory. While most of these suggestions entail undesirable security properties, they were motivated by restrictions of early IOMMU hardware such as the inability to invalidate specific IOTLB entries.

Tomonori [53] and Cascardo [13] made improvements to IOMMU drivers that increased the single-threaded performance of IOVA allocators. At the time of their work, no scalable IOVA allocators had yet been proposed, and most IOMMU driver subsystems used global locks to serialize access. Malka further improved Linux’s single-threaded IOVA allocator by addressing a pathology in its allocation heuristic that caused it to take linear rather than constant time for usage patterns common among high performance peripheral devices [34]. However, even with this improvement, a 40Gbit/s NIC achieved just over 3Gbit/s in a streaming TCP benchmark under a
strict IOTLB invalidation scheme.

Peleg et al. published a multicore-scalable design for the Linux IOMMU driver in 2015 [45]. In particular, they added a per-core caching layer over the existing Linux IOVA allocator and designed a scheme for scaling batched IOTLB invalidation to multiple cores. Page table management in the driver already scaled well. These changes improved multicore IOMMU driver performance, which is sometimes critical in driving high-throughput peripherals. However, a design that relies on batched IOTLB invalidation cannot be used in existing systems to provide effective protection from malicious peripheral devices, as we will see in Section 4.1. When configured with a strict IOTLB invalidation scheme, their system achieved up to 80% less throughput than the no-IOMMU case while completely saturating all 16 CPUs. It also incurred up to 12x the CPU cost. This design also does not protect against one of the vulnerabilities introduced in Chapter 4.

In 2016, Markuze et al. proposed a technique called shadow buffering as an alternative to existing IOMMU usage schemes [37]. With shadow buffering, the operations related to IOVA, page table, and IOTLB management usually performed during map and unmap calls are replaced with memory copy operations. In particular, each peripheral device has permanent access to a pool of so-called shadow buffers in memory. On a map call, data to be exposed to the peripheral are copied into the pool. On an unmap call, the data are copied out of the pool. Shadow buffering enables effective protection from DMA attacks, and replacing expensive IOMMU operations from map and unmap with copy operations delivers I/O performance and CPU utilization on par with other existing strategies. However, this performance is not sufficient for general-purpose IOMMU use and also comes at the cost of negative cache effects. Shadow buffering is presented and evaluated in detail in Section 5.3.1.
Kumar and Maass developed LATR, a system that allows MMU operations involving costly inter-processor TLB shootdowns to be performed asynchronously. While not strictly related to the IOMMU, their system involves ideas similar to those presented about asynchronous IOTLB invalidation in Section 5.2.1. However, the use cases, implementations, and concerns about correctness and security for MMU and IOMMU operations are different. LATR also does not propose augmenting software interfaces to expose asynchrony directly.

### 3.2.2 Hardware designs

This thesis is focused on improving the use of existing IOMMU hardware, but some related prior work has proposed novel IOMMU hardware designs. Amit et al. suggested hardware enhancements as well as accompanying software driver modifications designed to increase the IOTLB hit rate. While increasing the IOTLB hit rate is desirable, IOTLB invalidations represent a much more significant issue in using the IOMMU with high-performance peripheral devices.

Malka suggested an IOMMU design to support high-performance peripheral devices efficiently. In particular, the proposed design features a flat, as opposed to hierarchical, IOMMU page table augmented by an IOTLB that only contains one entry. For NICs and some NVMe drives, which interact with memory in a sequential and predictable manner, this design ostensibly eliminates the need for the IOMMU driver to explicitly perform IOTLB invalidation, since each distinct memory access would evict the previous one’s IOTLB entry. However, this design would not support multiple devices or even single devices with parallel access patterns well. It would also reduce IOMMU performance in other scenarios, such as ensuring guest isolation in a virtualized system. Relying on a subsequent memory access to invalidate the IOTLB
may also result in stale IOTLB entries in certain scenarios.

Basu et al. patented an IOMMU design that also attempts to obviate the need for IOTLB invalidation [6]. The patent describes an IOMMU with self-invalidated mappings that automatically remove their page table entries and invalidate the corresponding IOTLB entries without any processor interaction based on certain removal rules. For example, these rules could specify self-invalidation after a certain number of accesses. Such a design, if implemented, could greatly reduce the performance cost of IOMMU use overall without compromising performance in important use cases or introducing invalidation semantics that are difficult to reason about.

### 3.3 Memory encryption

Full memory encryption could, in theory, be used to protect systems against DMA attacks without relying on the IOMMU. In such a scheme, the entire contents of a system’s memory would be encrypted and only accessible in plaintext form to actors with the appropriate secret keys. Peripheral devices would not be able to read sensitive data stored in memory, and authentication codes could be used to detect unauthorized writes.

However, memory encryption comes with a set of performance and usability issues. A recent survey by Henson and Taylor discusses a number of ways to achieve full memory encryption, including with software alone, with specialized hardware devices, and as an integrated part of commodity microarchitectures [24]. They conclude that while some specialized industrial products currently work well, general purpose memory encryption is not mature enough and does not perform well enough for widespread adoption. Furthermore, using memory encryption to protect against DMA attacks would require storing encryption keys outside of DRAM, managing a large number of
keys and complex encryption schemes to grant devices access to particular regions of memory, and transmitting those keys to devices securely. A system addressing these issues has not been discussed in the literature.

Despite these issues, memory encryption has been investigated in the context of protecting against DMA attacks. Colp et al. devised a system called Sentry for memory encryption on ARM smartphones that involves storing secret keys off of DRAM in a chip close to the processor [14]. With Sentry, memory is encrypted when the phone is locked, so a malicious peripheral attacker could not read any sensitive data. However, memory gets decrypted when the phone is unlocked, so the system does not have feature parity with IOMMU protections.

Additionally, AMD and Intel have recently introduced general-purpose memory encryption technologies. AMD’s Secure Memory Encryption (SME) is an architectural feature that allows the contents of memory to be transparently encrypted with AES [30]. With this feature enabled, setting a certain bit in the physical address of a page in its page table entry causes that page to be automatically encrypted and decrypted when it is written and read by the processor. However, devices can access encrypted memory if they set the appropriate bit in the addresses they use for DMA, so this technology does not protect against malicious peripherals. AMD’s Secure Encrypted Virtualization (SEV) encrypts the memory belonging to guest operating systems in a virtualized environment and does not allow DMA access to that memory [30]. Intel’s Memory Encryption Engine (MEE), introduced as part of its Software Guard Extensions, supports encrypting data stored in protected regions of DRAM [23]. These technologies may enable a memory encryption-based scheme for protection from malicious peripherals in the future.
Chapter 4

OS-level IOMMU Protection Vulnerabilities

This chapter presents previously unexploited vulnerabilities in the way that existing operating systems use the IOMMU to provide protection from DMA attacks. It shows that the vulnerabilities can affect systems that instantiate our IOMMU usage model and shows how they can be exploited in practice. This chapter also describes a novel peripheral attacker model that more accurately reflects the current capabilities of real adversaries than previous models. It then introduces Thunderclap, a new FPGA-based DMA attack platform that is significantly more powerful than previous such platforms. It demonstrates how Thunderclap was used to completely compromise FreeBSD, macOS, Linux, and Windows, even with their existing IOMMU protections enabled.

4.1 OS-level vulnerabilities

Even when the IOMMU is correctly configured and used by the operating system in line with our IOMMU usage model, it may provide not complete protection against malicious peripheral devices. We now consider two vulnerabilities that are present in existing operating systems. One vulnerability is related to the granularity of IOMMU page table protections, and the other is related to the widespread adoption of non-strict IOTLB invalidation semantics. These vulnerabilities have been posited in previous work but not investigated and exploited in real systems.
VT-d supports a hierarchical page table structure with 4KB pages, and other major IOMMUs are similar. However, regions of memory that do not completely occupy 4KB pages are routinely exposed to peripheral devices. In existing systems, mapping such a region leaks whatever other data might be co-located on the page because mappings are made at page granularity, as shown in Figure 4.1. This potentially sensitive leaked information could be arbitrary data structures allocated by kernel memory allocators. It could also be a sensitive region of a data structure for which only a particular non-sensitive region was meant to be exposed to the device. Both of these scenarios are demonstrated in real systems below. Depending on the permissions of the affected mapping, a peripheral could exploit this vulnerability to violate system confidentiality by reading sensitive data and to violate system integrity by modifying kernel data and control structures. A system with this vulnerability does not violate the properties of our IOMMU usage model, since memory is only exposed to devices purposefully via their drivers. The operating system simply exposes memory at page granularity, which leaks potentially sensitive data in existing systems.

Another vulnerability stems from non-strict IOTLB invalidation semantics in IOMMU drivers. Existing commodity operating systems, including FreeBSD, macOS, and Linux, feature some form of non-strict IOTLB invalidation as their default or only mode of operation. With these non-strict semantics, the IOMMU driver unmap operation simply initiates an IOTLB invalidation before returning. Additional processing related to the unmap call, for example to complete the IOTLB invalidation and free IOVA ranges as appropriate, occurs in an asynchronous context. Non-strict semantics allows for optimizations, such as batching IOTLB invalidations, that greatly improve performance. However, it also creates a window of time during which the operating system believes that a certain physical page is no longer accessible to a
Figure 4.1: Example of the protection granularity IOMMU usage vulnerability.
device while it is actually still accessible via a stale IOTLB entry. In particular, existing systems perform non-strict (asynchronous) IOTLB invalidation behind the synchronous `map` and `unmap` interface. Once control flow returns from an `unmap` call, the calling driver expects that its device can no longer access the unmapped memory. Commonly, the unmapped memory is immediately freed and reused to hold other data or to perform some other function entirely. Under non-strict invalidation, the device can actually access this reused memory until the asynchronous IOTLB invalidation completes, which can take on the order of tens of milliseconds for batched invalidation schemes. By taking advantage of this vulnerability, a malicious peripheral could violate system confidentiality or integrity by accessing data on a reused physical page via a stale IOTLB entry. This vulnerability is illustrated by Figure 4.2, which depicts a timeline of the above events. The shaded region indicates the time period during which a malicious device could take advantage of a stale IOTLB entry. A system with this vulnerability also does not violate the properties of our IOMMU usage model. Memory is only exposed to devices purposefully via their drivers, but sensitive data can be leaked due to the timing of the exposure.
4.2 Attacker model

This thesis considers a novel peripheral attacker model that is more powerful than previous models and more accurately reflects the capabilities of real attackers. The new model assumes that an attacker controls one or more malicious peripherals that have access to the system bus via ports on the outside of a machine’s case. These malicious peripherals appear to the target system as regular devices and even correctly emulate the functionality of those regular devices. However, they have the ability to simultaneously make arbitrary DMA requests in an attempt to compromise the system. The attacker model does not assume access to code running on the target system, including IOMMU driver or peripheral device driver implementations.

The novel attacker model is more powerful than previous models because it makes fewer assumptions and works effectively against more systems. Previous attacker models have involved attack devices that cannot emulate legitimate peripherals or rely on scanning all of physical memory to carry out attacks. These attack models are ineffective against systems that conform to our IOMMU usage model. Other attack models have assumed access to privileged code running on a target machine, e.g. IOMMU driver or peripheral driver implementations, which the novel attacker model does not assume.

Because attack devices emulate legitimate peripherals under the new model, device drivers attach and expose regions of memory to them. The attack devices can then send arbitrary DMA requests to probe and exploit the two OS-level vulnerabilities described above. These vulnerabilities are currently ubiquitous in commodity systems and provide an attack surface large enough for a malicious device to compromise the machine completely, including by gaining arbitrary privileged code execution, as described below.
This novel model accurately reflects the capabilities of current attackers. The widespread availability of FPGA technology has made it relatively easy for individuals to develop complex hardware. System emulators like QEMU are able to emulate the functionality of many existing peripheral devices. Finally, trends in hardware interconnects like the rise of Thunderbolt 3 over USB-C have made pluggable access to system buses a reality for malicious actors.

4.3 The Thunderclap attack platform

Thunderclap is a novel DMA attack platform that instantiates the above attacker model. It is built on an FPGA and makes use of synthesized hardware elements as well as a softcore processor running a complex software stack to emulate devices and run arbitrary attack code. Thunderclap greatly exceeds the flexibility and capabilities of previous DMA attack platforms by being able to masquerade as legitimate peripheral devices and simultaneously make malicious DMA requests. This section outlines the design of the Thunderclap platform, which was built by my colleagues A. Theodore Markettos and Colin Rothwell at the University of Cambridge. My contributions to this work involved developing novel exploits against real operating systems and implementing them on Thunderclap, as discussed in the next section. I further helped to write a software library that enables code running on Thunderclap to make DMA requests.

The design of Thunderclap is outlined in Figure 4.3. The entire stack is based on an FPGA platform. A Terasic DE5-NET FPGA board containing an Intel/Altera Stratix V FPGA was used for development, but the platform should also fit on a significantly cheaper Cyclone V board. An SD card reader was added to the board to support an operating system’s root filesystem. A 64-bit BERI softcore CPU
and an interface to the board’s hard PCIe core were synthesized on the FPGA as well as standard logic to control memory, storage, and UARTs. The synthesized PCIe interface allows programs running on the CPU to send and receive raw transaction layer packets (TLPs) through simple queues. Finally, a PCIe to Thunderbolt 3 bridge was attached to the board’s PCIe slot. A different bridge or dongle could be attached to this slot to allow Thunderclap to target other hardware interconnects.

Thunderclap runs a complicated software stack to support device emulation and arbitrary DMA request generation. The BERI CPU core runs FreeBSD 11, which provides an environment to support the QEMU full system emulator [46]. QEMU has the ability to emulate a number of peripheral devices in software, and Thunderclap uses these software device models to control how it interacts with the target machine over PCIe. In particular, Thunderclap runs a version of QEMU with three main modifications: (1) the event loop that controls device simulation was simplified significantly to run on the relatively underpowered 100MHz BERI processor, (2) the
functions previously called by the device model to simulate sending and receiving PCIe TLPs, including those that simulate making DMA transactions and sending interrupts, were replaced with ones that send and receive TLPs to and from the target machine, and (3) the main event loop was modified to run arbitrary user-supplied attack code that has access to the emulated device’s state as well as the PCIe-related functions.

The attacks presented in the following section run Thunderclap with QEMU’s device model for the Intel 82574L Gigabit Ethernet Controller. This NIC was chosen because all investigated operating systems have driver support for it. After being plugged in to a target machine, Thunderclap is probed over PCIe by the machine to determine what kind of device it is. It delivers that probe to QEMU, which sends the emulated NIC’s response back to the machine. Once the machine’s NIC driver attaches, Thunderclap again uses QEMU to respond to configuration commands and then to emulate the functionality of the NIC. Meanwhile, it also runs the malicious payload to make DMA requests that compromise the machine.

The relatively high latency of generating PCIe TLP responses in software rather than hardware did not prevent Thunderclap from interacting with the investigated systems. Also, because modern PCIe devices typically send interrupts to the CPU by writing to a particular memory location, Thunderclap is able to emulate interrupt behavior without any special consideration. QEMU additionally provides a number of ways to connect to a real network from an emulated NIC. In particular, QEMU could make use of the FreeBSD network stack and the networking hardware present on Thunderclap to let its emulated NIC send and receive real network traffic. However, this feature was unnecessary in practice. Interactions between the emulated NIC and some basic QEMU-internal network services, such as a virtual DHCP server, were
enough to convince the investigated systems that the emulated NIC was a legitimate peripheral.

### 4.4 Example attacks

This section presents examples of how Thunderclap can be used in practice to compromise target systems. In particular, it presents an attack that targets the FreeBSD network stack as present in both FreeBSD and macOS. By emulating the Intel 82574L Gigabit Ethernet Controller, Thunderclap gains access to sensitive kernel data structures from the emulated NIC’s driver. This access gives Thunderclap the ability to obtain privileged code execution on both operating systems. A similar attack allows Thunderclap to read and overwrite sensitive kernel data structures on a Linux system. In some cases, the examined operating systems do not correctly instantiate our IOMMU usage model or fail to implement other non-IOMMU-related vulnerability mitigation techniques. For example, IOMMU support in Windows is not designed to protect against DMA attacks, but it is investigated anyway. The example attacks take advantage of these failures to demonstrate Thunderclap’s full potential to compromise real systems. However, the vulnerabilities underlying these exploits could be exploited even if the examined operating systems did instantiate our IOMMU usage model.

As a whole, these examples should be considered a small part of a wide range of possible attacks spanning many different emulated devices and adversarial techniques. I developed the novel attack against the FreeBSD network stack, which was used to compromise both FreeBSD and macOS. The ideas behind my attack were later applied by my colleagues to investigate Linux and Windows systems.
4.4.1 FreeBSD

FreeBSD isolates each connected peripheral device in its own initially empty I/O virtual address space. As a result, Thunderclap can only access memory that is explicitly exposed to it by the driver of the device it is emulating. This subsection discusses how current NICs and CPUs communicate via main memory in systems that use an IOMMU, which is largely similar across various operating systems and types of NICs. It then describes the network stack data structures that are exposed to NICs by device drivers on FreeBSD in particular. Finally, it demonstrates how Thunderclap can take advantage of that exposure to execute arbitrary privileged code on a target FreeBSD machine.

Descriptor rings

NICs and CPUs mainly communicate through circular arrays known as descriptor rings that are accessed by both the NIC hardware and the software driver. Each element in a descriptor ring is known as a descriptor. The exact structure of descriptors varies among NICs, but they generally contain a pointer to a memory location, an amount of memory to read or write, and some other metadata and status information. Descriptor rings are allocated and initialized by the NIC driver when it attaches, and the driver then programs the NIC with the location of the rings in memory. Modern NICs use distinct descriptor rings to manage transmitting and receiving packets, and these rings are always exposed to the NIC. Both types of descriptor ring are illustrated in Figure 4.4.

The transmit descriptor ring works roughly as follows. The NIC hardware maintains an index into the ring indicating its current position in transmitting packets, and the driver maintains an index into the ring indicating where new packets to transmit will
be inserted. The NIC’s index can be thought of as the head of the descriptor ring, and the driver’s index as the tail. When the driver receives a packet to transmit from the network stack, it exposes the memory containing that packet to the NIC then inserts one or more descriptors that contain the IOVAs and lengths of buffers that make up the packet at the tail of the descriptor ring. Next, the driver writes to a doorbell register on the NIC to indicate that there is outstanding work. The NIC will then process valid descriptors from the head to the tail of the descriptor ring by reading the specified packet information from memory via DMA and transmitting the packet. Finally, it sends an interrupt to the CPU, and the driver responds by removing processed descriptors from the ring, unmapping the corresponding memory from the NICs I/O virtual address space, then freeing the memory.

The receive descriptor ring works slightly differently from the transmit ring. The
NIC’s index indicates where received packets will be inserted and can be thought of as the tail of the ring. The driver’s index indicates where the CPU will start processing received packets and can be thought of as the head. When no outstanding received packets are present in the ring, all of the descriptors contain the IOVAs and lengths of empty network buffers that are all exposed to the NIC. When the NIC receives a packet, it writes the received information to one or more of these empty buffers via DMA then sends an interrupt to the CPU. The driver responds by unmapping the memory corresponding to the received packet from the NIC’s I/O virtual address space, passing the received packet up the network stack for processing, then replenishing the used descriptors with empty network buffers that are exposed to the NIC.

Though descriptor rings are always exposed to NICs, they do not contain any sensitive information. However, the network buffers that are exposed to the NIC via ephemeral mappings, i.e. mappings that are created then quickly destroyed, do present a significant attack surface.

Mbufs

FreeBSD uses the mbuf data structure to store network packets. Each 256-byte mbuf consists of a variable-sized mbuf header that contains information about the type and size of its stored packet as well as flags describing that packet. Each mbuf also has a small internal buffer for data but can be configured to reference an external region of memory for storage. A single packet may span multiple mbufs that are linked together in an mbuf chain. This linked-list design obviates the need to allocate a new buffer and copy data each time packet headers are added or removed as the packet traverses the network stack.

Most mbufs that contain the body of a packet are configured to use external
memory regions known as clusters for data storage. In current systems, these clusters are typically 2KB and co-located on the same 4KB page as another 2KB cluster. When a particular mbuf cluster is exposed to a malicious NIC via a descriptor ring, that NIC also gains access to the co-located cluster, which might have been allocated for a different device. The malicious NIC could then read or modify potentially sensitive data in that co-located cluster, depending on the mapping permissions.

Mbufs that contain packet headers are configured to use their small internal buffers for data storage. Many such mbufs are allocated on a single 4KB page. When a particular internal buffer is exposed to a malicious NIC via a descriptor ring, not only does the NIC gain access to the internal buffers of co-located mbufs, but it also gains access to mbuf fields that control kernel behavior.

**Kernel control-flow hijacking**

A common feature in modern NICs is header splitting, which configures the NIC to DMA a received packet’s header and body into separate buffers. On FreeBSD with this feature enabled, memory regions containing mbuf structures are exposed to the NIC for writing packet headers. As mentioned above, this exposure allows the NIC to overwrite sensitive mbuf fields. By overwriting these fields in a sophisticated way, a malicious NIC can induce the system to execute arbitrary privileged code.

Mbufs support using custom memory regions (i.e. not clusters) for external data storage. To facilitate this feature, the mbuf data structure contains a pointer to a user-supplied function that is called to free the custom external storage when the mbuf itself is freed. The first parameter to this free function is the kernel virtual address of the mbuf, and the second and third parameters are user-controlled 64-bit fields in the mbuf structure.
A Thunderclap attack that exploits access to mbufs to hijack kernel control flow works as follows. First, the emulated NIC reads the IOVA of an mbuf’s internal buffer for writing a packet header from the receive descriptor ring. The malicious attack code sends DMA writes that modify the flags of that mbuf to falsely indicate that it is using external storage with a custom free function. For the custom free function to be executed without causing a kernel panic, the attack code must also (1) set a flag to indicate that the mbuf is maintaining an internal reference count for the external storage and (2) set that reference count to 1. The attack code can then overwrite the free function and arguments, and the payload will be executed when the modified mbuf is freed. It usually takes no more than a few seconds between payload injection and execution.

With this attack, it is trivial for a malicious peripheral to cause a kernel panic and crash the system. It is also straightforward for such a peripheral to call existing kernel functions with the ability to control the second and third arguments. Using return oriented programming (ROP) attacks, it is possible for a malicious peripheral to induce the kernel to execute arbitrary code. In particular, the second argument field could be modified to point to a region of memory that contains a stack of return addresses for a ROP program, and the free function pointer could be modified to point to a ROP gadget that sets the stack pointer to the value of the second argument register and returns. Because FreeBSD does not implement kernel address space layout randomization (KASLR), a mitigation that randomizes the virtual addresses of kernel symbols across boots, no special consideration must be given to subverting it.
Practical concerns and disclosure

A number of issues must be considered in carrying out this attack against FreeBSD in practice. The first is that FreeBSD does not currently support Thunderbolt hotplugging. However, it does support Thunderbolt devices that are connected at boot time without access control. While this lack of hotplugging support does remove the possibility of undetected attacks on unattended but running laptops or workstations, it does not preclude Trojan attacks in which, for example, a seemingly innocuous Thunderbolt dongle connecting to a projector might simultaneously compromise the attached machine. Furthermore, an attacker could manually reboot an unattended laptop or workstation with the attack device inserted. The reboot would certainly be detected by the owner of the target machine but may not be interpreted as malicious activity.

Another consideration is that stock FreeBSD requires explicit administrator configuration of attached NICs, so that a newly connected NIC will not be used by the system without user input. However, TrueOS (formerly PC-BSD), a desktop-oriented distribution of FreeBSD, does perform DHCP by default on newly attached NICs. Though the above attack was described above to work against the most recent 11.1 release of FreeBSD, it was carried out in practice against a TrueOS release based on FreeBSD 10.3.

Finally, FreeBSD’s DMA KPI does not support mapping regions of memory with read-only or write-only privileges. In particular, every region of memory exposed to a device is both readable and writable by that device. This property makes the above attack easier to carry out and more powerful in practice. Instead of depending on the NIC driver to enable header splitting, the malicious NIC can write to the mbuf structures containing packet headers exposed to it by the transmit ring during
DHCP. Additionally, the malicious NIC can read the kernel virtual address of the mbuf’s internal buffer, write arbitrary data into that buffer, then set the free function arguments to point to that data.

This attack was disclosed to the FreeBSD Project. They indicated that malicious peripheral devices are not currently within the scope of their threat model and do not warrant an immediate security response. However, they expressed concern about this kind of attack and requested a copy of a relevant submitted manuscript for further review.

4.4.2 macOS

macOS’ kernel uses the FreeBSD network stack. As a result, the above attack for FreeBSD applies almost exactly. One interesting difference in macOS is that all three parameters to an mbuf’s custom free function are members of the mbuf struct and can thus be set by an attacker. However, macOS uses kernel address space layout randomization (KASLR) to randomize the address of kernel symbols across boots. KASLR does not prevent the above attack from causing a kernel panic and crashing the system, but it does make it more difficult to carry out targeted attacks based on executable code already present in the kernel, such as calling a specific kernel function or carrying out a ROP-style attack. However, macOS has weaknesses in its IOMMU usage that allow peripheral attackers to subvert KASLR and gain the ability to induce arbitrary privileged code execution.

Breaking KASLR

macOS uses the IOMMU by associating each device with the same shared I/O virtual address space. It also creates mappings with read permissions no matter what
permissions are specified to the DMA KPI functions. Write permissions are only included if they are specified. These weaknesses mean that each device attached to the system can read memory exposed to every other device. When targeting macOS, Thunderclap can snoop on all peripheral traffic, including disk-related data and graphics buffers that drive displays, in addition to potentially sensitive data in network buffers co-located on pages with its own. An experiment revealed that Thunderclap could reliably extract plaintext data from an IPSec VPN connection running over the motherboard’s NIC.

Though macOS overexposes a significant amount of sensitive information to a peripheral attacker, including mbuf control structures, its use of KASLR makes it harder for the attacker to execute specific kernel code. macOS implements KASLR by randomly deriving a slide value each boot and adding that slide to the address of every kernel symbol. In theory, it should be impossible for a malicious peripheral or other kernel attacker to determine this slide and thus modify the mbuf free function pointer to reference any meaningful attack code. In practice, however, the kernel may leak the KASLR slide to attackers, allowing them to subvert the mitigation.

I was able to subvert KASLR protections from Thunderclap in the following way. First, I determined experimentally that the KASLR slide is always a multiple of 2MB, meaning that the lowest 21 bits of a kernel symbol are not changed regardless of the slide value. Then, using a target macOS machine under my control, I recorded the slide value of a particular boot. From Thunderclap, I scanned the entire I/O virtual address space during that boot for eight-byte regions of memory that pointed to any KASLR-slid kernel symbol. This scan was performed by comparing each eight-byte region against the known static address ranges that contain kernel symbols with the known slide applied to those ranges. I discovered that both the USB and AHCI drivers
leak the KASLR-slid address of a symbol, \texttt{gIOBMDPageAllocator}, in mappings that are perpetually exposed to all peripheral devices from boot time. In attacks against macOS, Thunderclap first scans memory for a pointer whose lowest 21 bits match the static address of \texttt{gIOBMDPageAllocator}. With high probability, any pointer it encounters actually does point to \texttt{gIOBMDPageAllocator}, and subtracting its static address from the pointer yields the KASLR slide. This heuristic has never failed in experiments, and scanning the address space usually takes no more than a few seconds because mappings opened at boot time tend to appear in the same general area of the I/O virtual address space across boots.

With KASLR defeated, it is straightforward to take complete control of a target macOS machine by controlling the arguments to arbitrary kernel function calls. For example, the kernel function \texttt{KUNCExecute} executes a specified program as a specified user. Using \texttt{KUNCExecute}, Thunderclap can modify an mbuf to run \texttt{Terminal.app} as root when it is freed, which gives a root shell. Similar attacks can be carried out to bypass lockscreens or take other privileged actions. Like in the FreeBSD case, ROP-style attacks are possible and enable arbitrary privileged code execution.

**Practical concerns and disclosure**

macOS allows Thunderbolt hotplugging by default, but only for Thunderbolt devices that are on a system whitelist containing approved Thunderbolt ID ROMs. The process and requirements for ending up on Apple’s Thunderbolt whitelist are not clear, but devices sold as Mac compatible are generally on the list. macOS also only attaches device drivers to system-approved PCIe devices connected via Thunderbolt. This protection was subverted by programming Thunderclap to emulate the PCIe Device ID and Vendor ID of the 82574L inside Apple’s first-generation Mac Pro desktop.
Whitelisted Thunderbolt devices, including PCIe to Thunderbolt bridges that are connected to malicious PCIe devices spoofing legitimate ones, are free to make PCIe transactions after hotplugging. It may also be possible to spoof Thunderbolt ID ROMs. macOS laptops and desktops are thus at risk of difficult-to-detect hotplug DMA attacks.

Because macOS associates all peripheral devices with a single I/O virtual address space, the attacks described above could be carried out by an unsophisticated DMA attacker that simply spoofs some of the configuration registers and not the functionality of a peripheral approved for PCIe access. In particular, even peripherals to which no regions of memory have been specifically exposed can access memory in the single I/O virtual address space. This observation was validated in practice by successfully compromising macOS from a less powerful version of Thunderclap that emulates certain configuration registers but not the functionality of a legitimate peripheral. The fact that macOS exposes all mbufs to all devices with read and write permissions perpetually from boot time made the above experiment significantly easier to carry out.

The attack presented in this section was demonstrated against macOS 10.11.5 and disclosed to Apple. They considered the attack within the scope of their threat model and patched the vulnerability in macOS 10.12.4. In particular, macOS now XORs function pointers contained in mbufs with a secret cookie that is not exposed to peripheral devices. It has also moved the flags that were modified in the attack outside of device-accessible memory. Additional work should be carried out to determine if an mbuf-based attack is still viable against new versions of macOS. Weaknesses in the system’s IOMMU use could also be used to develop exploits involving different peripheral devices and I/O subsystems.
4.4.3 Linux

My colleagues applied the ideas behind the above attacks to examine how to subvert IOMMU protections on machines running Linux as well. Linux isolates each device in its own I/O virtual address space, and examined Linux peripheral drivers expose regions of memory to devices with appropriate read and write permissions. Linux does not use the mbuf structure to store network packets, but its sk_buff structure performs the same function and is largely similar in form. The main difference between mbufs and sk_buffs is that sk_buffs do not have a small internal region for data storage but can only use an external data region allocated from kernel memory. Descriptors in NIC transmit and receive descriptor rings point to these external regions.

Linux’s attack surface for a malicious NIC is reduced compared to FreeBSD and macOS because the sk_buff lacks an internal data region and device drivers expose regions of memory to peripherals with appropriate access permissions. However, Linux systems are still vulnerable to malicious NICs. In experiments, SystemTap tracing revealed that sensitive kernel control structures were co-located on pages with 2KB external sk_buff regions exposed to the NIC with write access via the receive descriptor ring. For example, the kernel dispatch table for Network Address Translation rewriting functions, which contains a number of kernel function pointers, was observed. This sensitive data exposure suggests that a malicious NIC could easily crash the system by causing a kernel panic.

Sensitive data were also co-located with the external sk_buff regions exposed to the NIC with read access via the transmit descriptor ring. Visible to a malicious NIC were system call kernel stacks, which contained enough information to break KASLR for kernel code and data, and plaintext VPN traffic that was sent over a different NIC. A sophisticated attacker might be able to combine its read and write accesses to carry
out even more powerful exploits against Linux despite its reduced attack surface.

**Address Translation Services**

In the PCIe specification, Address Translation Services (ATS) is a feature that allows devices to perform their own IOMMU translations. Allowing devices to perform their own translations is also referred to as supporting device IOTLBs. If ATS is enabled, DMA requests with a certain bit set will bypass IOMMU translation completely. If ATS is disabled, DMA requests with this bit set are simply discarded.

Linux allows any device that advertises ATS capabilities to use them. My colleagues confirmed that when Thunderclap was modified to advertise that it supported ATS, Linux enabled the ATS feature in the system’s PCIe switches. Setting the translation bypass bit in each DMA request allowed Thunderclap unrestricted access to physical memory, completely bypassing IOMMU protections.

Linux was the only examined system to enable ATS. It should be noted that ATS also requires support from underlying hardware. For example, tested server-class hardware did support ATS, while desktop-class hardware did not. ATS is a feature of the system bus that could allow attackers to completely bypass IOMMU protections, and thus it should never be enabled in systems that want to protect against malicious peripheral devices.

**Practical concerns and disclosure**

Linux recently added support for Thunderbolt hotplugging with access control as described in Section 2.4. Intel-developed userspace command line tools allow users to interact with Thunderbolt devices as they are plugged in and make access control decisions. As discussed above, this access control does provide increased protection
against DMA attacks but does not stop sophisticated attackers from gaining access to the system bus.

Various Linux distributions, including Ubuntu, Red Hat Enterprise Linux, and Fedora, were examined and found to be similar in terms of their IOMMU usage. The particular distribution used for the memory probing experiments discussed above was Fedora 25 with the Linux 4.8 kernel. Vulnerabilities were disclosed to the Linux kernel security team and found to be within their threat model. My colleagues are in ongoing talks with them about these issues.

4.4.4 Windows

Most versions of Windows do not support using the IOMMU at all, leaving them completely vulnerable to all types of DMA attackers. In particular, Windows 7, 8.1, 10 Home, and 10 Pro do not support the IOMMU. Windows 10 Enterprise does use the IOMMU if the optional Virtualization-Based Security (VBS) feature is enabled [38]. With VBS, the Windows operating system is run as a guest of Hyper-V, Microsoft’s commodity hypervisor. A small second guest OS runs alongside Windows and controls a secure region of memory that can store sensitive data such as user credentials as well as Windows-related kernel code integrity information. Even if an attacker completely compromises Windows, the hypervisor ensures that they cannot access sensitive data managed by the other guest OS.

VBS uses the IOMMU to prevent peripheral devices, which are controlled by the Windows guest, from accessing memory owned by the hypervisor and the other guest. It does not use the IOMMU to protect the Windows operating system from DMA attacks. My colleagues determined that all peripheral devices share a single address space and that memory owned by the Windows guest was completely exposed to
all devices. Even unsophisticated DMA attackers that do not emulate a legitimate peripheral can completely compromise the Windows OS. While a scheme like VBS could theoretically protect Windows from malicious peripherals, it would require every piece of sensitive data and all code to be managed and checked by the other secure guest OS, which is impractical. In reality, complete DMA access to the Windows guest in VBS is enough to carry out an extremely wide range of powerful attacks. Additionally, even if VBS were to instantiate our IOMMU usage model to provide protections for the Windows guest, its network stack would likely be vulnerable to attacks similar to those presented above.

**Practical concerns and disclosure**

Similarly to Linux, Windows supports Thunderbolt hotplugging with access control. It has largely the same benefits and drawbacks. An additional drawback of Windows’ Thunderbolt hotplugging support is that GUI dialog boxes requesting user input about access control automatically appear when a device is inserted. These dialog boxes only identify the Thunderbolt device by its self-reported name. Because (1) the dialog boxes are GUI elements that appear automatically rather than command line tools, (2) the boxes do not display adequate information for users to make informed decisions, (3) and uninformed users may be conditioned to respond to dialog boxes affirmatively (allowing rather than restricting behavior) to ensure their system continues to function, Windows systems are at increased risk of hotplug DMA attacks.

The above investigation was carried out on Windows 10 build 14393. This work was disclosed to Microsoft’s Security Response Center, which indicated that it is not currently within the scope of their threat model. They stated that their primary defense against malicious peripheral devices is to disable DMA entirely, and cited an
optional Windows Group Policy that disables DMA from hotplugged devices while a machine’s screen is locked. They alternatively recommended shutting down or hibernating machines when they are unattended. These defenses do not adequately protect against sophisticated peripheral attackers.

4.5 Discussion

This chapter demonstrated how DMA attackers can exploit operating system-level vulnerabilities in IOMMU usage to bypass protections and completely compromise target systems. It presented two vulnerabilities: one stems from operating systems’ failure to account for the 4KB granularity of IOMMU access control, and the other stems from their non-strict invalidation of IOTLB entries. It then introduced a new model for DMA attackers that includes the ability to gain access to the system bus from outside of a machine’s case and the ability to emulate the functionality of legitimate peripheral devices while simultaneously making arbitrary malicious DMA requests. It presented Thunderclap, a novel peripheral attack platform that instantiates the new attacker model. It finally showed how Thunderclap can completely compromise current operating systems to read sensitive data and induce arbitrary privileged code execution.

The attacks presented above can even affect systems that correctly instantiate our IOMMU usage model. However, this work also showed that all of the examined operating systems failed to instantiate the model in ways that made it significantly easier for a malicious peripheral to carry out DMA attacks and even subvert canonical attack mitigations such as KASLR. Operating system vendors and maintainers should make the nontrivial but relatively straightforward effort required to instantiate our IOMMU usage model. Furthermore, they should not rely on Thunderbolt access
control, even if it is governed by user input, to protect against peripheral attackers. Such access control does not protect against malicious devices that a user might allow, such as Trojan Thunderbolt devices that spoof their identity or PCIe attackers connected via a legitimate Thunderbolt bridge or dongle.

The above attacks primarily involved exploiting the IOMMU protection granularity vulnerability in the networking subsystems of different operating systems. This vulnerability was chosen as an illustrative example because it cannot be addressed by setting a kernel option and because exploiting it is relatively deterministic and straightforward. It is important to note that other I/O subsystems could be probed by programming Thunderclap to emulate any device supported by QEMU and that the vulnerability related to non-strict IOTLB invalidation could be explored by programming Thunderclap with a different malicious payload.

To achieve the ultimate goal of effective IOMMU-based protection from DMA attacks, the vulnerabilities presented in this chapter must be addressed. A practical system that aims to make strides towards removing the threat of DMA attacks should instantiate our IOMMU usage model, eliminate the above vulnerabilities, and also take further steps to mitigate attacks, such as isolating devices in their own I/O virtual address spaces. We will see strategies for addressing the above vulnerabilities, as well as strategies for reducing the performance cost of IOMMU usage, in the following chapter.
Chapter 5

Improvements in IOMMU Usage

This chapter presents strategies for improving the performance of operating system IOMMU usage and addressing the vulnerabilities presented in the previous chapter. It starts by introducing and evaluating a novel design for the IOMMU driver that significantly improves the performance of IOMMU usage. It then presents and evaluates new DMA KPI functions that make additional performance improvements and address the non-strict IOTLB invalidation vulnerability. Finally, it shows how novel memory management techniques can further reduce the performance cost of IOMMU usage and address the protection granularity vulnerability.

5.1 A novel IOMMU driver design

As described in Section 2.3, the IOMMU driver is responsible for three main tasks: allocating IOVAs, managing the IOMMU page tables, and invalidating IOTLB entries when appropriate. No matter what KPI or strategy a system uses to facilitate IOMMU protections, the IOMMU driver must perform these tasks to ensure mappings are correctly created and destroyed. As mentioned before, existing IOMMU drivers split their main tasks into distinct subsystems. This section describes the design of a novel IOMMU driver subsystem that combines IOVA allocation and page table management. It does not consider IOTLB invalidation, which is an orthogonal subsystem and must feature strict unmapping semantics for a system to provide effective protection from
DMA attacks.

The combined subsystem presented in this section reduces the amount of work and memory required to manage the IOMMU page tables relative to the current state of the art. To achieve these improvements, the vmem allocator is applied to govern IOVA allocation. Vmem’s import and release functionality allows it to implicitly manage the IOMMU page table as it services IOVA allocation and free requests. Its per-core caching layer allows both IOVA allocation and page table management to scale to multiple cores without significantly increasing synchronization overhead.

5.1.1 Vmem

Vmem is a general-purpose allocator conceived by Bonwick and Adams in 2001. It satisfies allocation requests with integer ranges that represent ranges of arbitrary resources. Vmem can either guarantee constant allocation time or minimize fragmentation, depending on a flag passed in with each allocation request. It also has a powerful interface that allows for the expression of complex allocation scenarios.

Vmem’s interface supports the following actions: creating and destroying vmem arenas that are used to manage resources, allocating and freeing resources from arenas, and configuring arenas to import and release resources from external sources dynamically. Vmem arenas control sets of integers from which allocations are made. Each arena has an initial set and a quantum size that represents the smallest unit of resource allocation possible for that arena. In particular, all ranges allocated by an arena will be quantum-aligned and rounded up to have quantum-multiple sizes.

When an arena cannot satisfy an allocation request, it calls a user-supplied import function to import an additional range of integers into its set. Typically, import functions are defined by the user at arena creation time and simply allocate resources
from other vmem arenas. However, they can be arbitrary functions that import any integer range and perform any additional operations. When an imported range is no longer in use, the arena that imported it calls a user-supplied release function to release that range from its set. Release functions are also typically defined at arena creation time and simply return imported ranges to other vmem arenas. But, like import functions, they can be defined arbitrarily. By equipping arenas with import and release functions that have useful side effects, it is possible to apply vmem in interesting ways.

5.1.2 Page table management

A subsystem that combines IOVA allocation and IOMMU page table management must (1) govern the allocation of IOVAs used to map memory into an I/O virtual address space and (2) guarantee that when it allocates a particular IOVA range, the IOMMU page table pages necessary to support translation for that range are already present in memory.

Vmem could be used as a drop-in replacement for existing IOVA allocators. A single arena of integers representing IOVAs with an initial set containing the entire I/O virtual address space and a quantum size equal to the I/O virtual page size would do the job successfully. The illumos operating system uses vmem in its VT-d driver in this way [15]. However, this IOVA allocation method does not make any guarantees about page table structure and requires that the page table be managed separately.

Vmem can be applied to make guarantees about the page table by equipping arenas with import and release functions that add and remove page table pages. The key insight behind the novel design is as follows. At each level of the page table, a single entry controls translations for a different amount of the total virtual address space.
For example, consider a standard four-level page table with 4KB pages and eight-byte page table entries. At the first (lowest) level, each entry controls the translation for a single 4KB page. At the second level, each entry points to a section of the page table that controls 512 page translations or 2MB of the virtual address space. Similarly, entries at the third level control 1GB, and entries at the fourth (highest) level control 512GB.

The proposed design involves creating an arena hierarchy consisting of one arena for each level of the page table. Each arena has a quantum size equal to the amount of the virtual address space controlled by an entry at its corresponding level. The highest-level arena has an initial set containing the entire virtual address space, and all other arenas have empty initial sets. Each arena except the highest-level one has an import function that (1) allocates a range from the next-highest arena in the hierarchy and (2) adds the page table pages at its corresponding level that control translations for the imported range. Each also has a release function that (1) removes the page table pages at its corresponding level that control translations for the released range and (2) frees the released range back to the next-highest arena in the hierarchy. Note the arenas’ quantum sizes guarantee that an integer number of entries will be modified on each import and release. Also note that the import and release functions may make recursive calls up the arena hierarchy as appropriate.

In an arena hierarchy, allocation requests are satisfied by the lowest-level arena, whose quantum size is equal to the virtual page size. For a range to be allocated by this arena, it must first be imported from all of the higher-level arenas in turn. The import functions guarantee that all of the page table pages required to translate the allocated range will be present in memory before the range is allocated. Thus the arena hierarchy successfully combines IOVA allocation and page table management as
An example arena hierarchy for a four-level page table with 4KB pages and eight-byte page table entries is depicted in Figure 5.1. Allocations are satisfied by the lowest-level arena with quantum size 4KB.

5.1.3 Per-core caching

Each vmem arena is protected by a lock to serialize operations involving that arena. These locks protect arena integrity but quickly become a bottleneck in multicore workloads. A well-known solution to this problem of allocator scalability is adding a layer of per-core caches on top of the allocator [11, 45]. Such caches can be accessed without acquiring the allocator’s lock and allow the allocator to scale without significant synchronization overhead to any number of cores.

Vmem arenas support quantum caching, a scheme that allows resource ranges up to a certain multiple of the quantum size to be stored in per-core caches. The exact multiple can be supplied at arena creation time. With quantum caching enabled, a vmem arena attempts to service both allocation and free requests through the caches.

To solve the scalability bottleneck in the vmem arena hierarchy, we simply enable quantum caching for the lowest-level arena. Since the lowest-level arena handles all allocations and frees, this modification allows the entire hierarchical allocation system to scale to multiple cores.

With quantum caching enabled, the lowest-level arena’s lock will only be acquired when a range is too large for the quantum caches or when a certain CPU core’s quantum cache becomes empty or full. In practice, quantum caches supporting ranges up to eight times the quantum (I/O virtual page) size were sufficient for almost all allocations made by a high performance 40Gbit/s NIC. Quantum caching successfully
Figure 5.1: An example vmem arena hierarchy.
reduces contention on the lowest-level arena’s lock and eliminates the allocation scalability bottleneck in the arena hierarchy design.

5.1.4 Page table walks

No matter the allocator design, once a range of IOVAs has been allocated, the IOMMU page table must be updated to contain that range’s translation. This task requires determining the kernel virtual addresses of the appropriate page table entries and modifying them. In an IOMMU driver with separate IOVA allocation and page table management subsystems, a software walk of the page table must be performed after each allocation to ensure that the page table pages required for translation of the allocated ranges are present in memory and to determine their kernel virtual addresses. In a driver with the vmem hierarchy design, guarantees about page table structure allow these page table walks to be eliminated.

During an import in the vmem arena hierarchy, pages of physical memory are allocated to be added to the page table. In particular, their physical addresses must be inserted into entries at higher levels of the page table so that the IOMMU can traverse it properly. Since the pages’ physical addresses are known at import time, their kernel virtual addresses in the direct-mapped region can be calculated. The direct-mapped region is a part of the kernel virtual address space present in many 64-bit operating systems that contains references to all of a machine’s physical pages. With the kernel virtual address of a page table page known, it is straightforward to calculate the kernel virtual address of a page table entry on that page. For example, given an IOVA and the kernel virtual address of the page table page that contains the entry governing that IOVA’s translation, the kernel virtual address of the entry can be calculated by adding an offset derived from the IOVA to the kernel virtual address
of the page table page.

Thus the kernel virtual address of a page table page, which is known at import time, can be passed to the allocation code, enabling the allocation code to directly calculate the kernel virtual address of the page table entry it must modify and obviating the need for a software page table walk. To pass this information from the import code to the allocation code, the lowest-level vmem arenas in the hierarchy could be modified to allocate values that encode both IOVAs and the kernel virtual addresses of corresponding page table pages.

Before a range of virtual addresses can be freed, the I/O virtual address management subsystem must update the page table to remove translations for the range. Existing designs typically determine which page table entry to update by performing another walk in software (similarly to the post-allocation case). In the vmem hierarchy design, the kernel virtual addresses of page table entries for allocated IOVA ranges are simply stored in the data structures that hold information about the mapping. It is then possible to replace looking up those kernel virtual addresses in software with a single memory access.

The result of the above scheme is to remove software page table walks from frequently-called code paths that create and destroy IOMMU mappings. The technique applied to encode both IOVAs and kernel virtual addresses in allocations from the lowest-level vmem arena could be applied to each level of an arena hierarchy to remove software page table walks from the IOMMU driver completely.

5.1.5 Discussion and evaluation

The design presented in this section makes a number of performance advancements compared to existing IOMMU driver designs. It is the first to combine IOVA allocation
and IOMMU page table management into a single subsystem. By combining these tasks and putting them behind a per-core caching layer, it reduces the amount of synchronization and total work required to create and destroy IOMMU mappings. Its policies of importing additional IOVAs only once it has exhausted existing sets and of releasing completely unused ranges help to minimize page table memory usage, and it is the only scalable page table management scheme that supports reclaiming unused page table memory. It is also significantly simpler to implement than existing solutions. In particular, because it makes use of the vmem allocator, it can be implemented in hundreds rather than thousands of lines of code and does not require developers to implement complex synchronization or allocation procedures explicitly. Features like vmem’s inherent quantum caching and support for boundary- and alignment-constrained allocations come at no additional development cost.

To evaluate the improvements of the vmem hierarchy design, we consider the performance of a 40Gbit/s NIC in a FreeBSD system with the IOMMU enabled. The machine under test was a Supermicro SuperWorkstation 5038D-I with a single four-core 3.40 Ghz Intel Xeon E3-1231 v3 (Haswell) CPU and hyperthreading disabled. The machine had four 8GB DDR3 DIMMs. The supporting machine was a Supermicro SuperWorkstation 5039D-i with a single four-core 3.50 Ghz Intel Xeon E3-1240 v5 (Skylake) CPU. The machine had two 16GB DDR3 DIMMs. The supporting machine was slightly more powerful so it would not be a bottleneck in throughput measurements. The machines were connected by two Intel XL710 QDA1 40Gbit/s NICs. The support machine ran FreeBSD 11.0 STABLE with the GENERIC kernel configuration, and the machine under test ran a modified version of FreeBSD 11.1 with kernel debugging options disabled. The support machine never enabled the IOMMU.

Experiments were performed to measure network throughput, CPU utilization,
network latency, and performance profiles for the system under test with various types of IOMMU support enabled. In particular, the system was measured with the IOMMU disabled, with the stock FreeBSD IOMMU driver, with the new vmem hierarchy design, and with the vmem hierarchy design plus the scheme to remove software page table walks. The design of modern NIC drivers is complex. To help interpret the results of these experiments, we review their operation, initially described in Section 4.4.1, in more detail.

**NIC drivers**

NIC drivers are primarily responsible for programming NICs to transmit and receive packets. As discussed earlier, the transmit and receive paths of a NIC utilize different descriptor rings and data flows. In particular, the transmit and receive paths carry out work in different threading contexts, and they use the DMA KPI differently.

The transmit path begins when packets containing data are passed from the OS’s network stack to the NIC driver. The driver calls the `map` operation on the packets, which returns the IOVAs at which the packets were mapped. Next, the driver adds those IOVAs to the transmit descriptor ring and indicates to the NIC that packets are ready for transmission. The NIC uses DMA to read and transmit the packets. After transmission, the NIC interrupts the CPU. The NIC driver’s transmit interrupt handler calls the `unmap` operation on the packets before freeing their memory. Note that this `unmap` call occurs in a different thread than the `map` call. In particular, the `map` call is made by the thread that submits packets to be sent by the NIC, and the `unmap` call is made by the interrupt handling thread that removes already-sent packets from the descriptor ring.

The receive path begins when the NIC receives packets from the network. The NIC
The NIC driver keeps the receive descriptor ring full of empty network buffers that have been mapped into the NIC’s I/O virtual address space by calls to the `map` operation. When the NIC receives data, it uses DMA to write the data into empty packet buffers in the receive descriptor ring. It then interrupts the CPU. The NIC driver’s receive interrupt handler calls the `unmap` operation on the received packets then passes them to the OS’s network stack for processing. Note that in this case, the only thread responsible for NIC-related work is the interrupt handling thread, which must perform an `unmap` operation on network buffers before passing them to the operating system’s network stack for further processing and delivery. Note also that some of this processing is frequently carried out by the interrupt handling thread itself. Often, this interrupt handling thread is also in charge of replenishing the receive descriptor ring with empty network buffers, so that buffers are replenished as they are consumed. In this case, the thread also has to make a number of `map` calls periodically as it processes received buffers.

These differences in the transmit and receive paths mean that IOMMU use affects them differently. The `unmap` operation is significantly more expensive than `map` because it involves an IOTLB invalidation. Because the receive path must make both `unmap` and `map` calls in its single performance-critical thread, enabling IOMMU support impacts receive performance more than transmit performance. Modern NIC drivers also support multicore systems, generally by allocating at least one transmit descriptor ring and one receive descriptor ring per core. Each core transmits and receives packets using its own descriptor rings. When the NIC receives a packet, it selects a core to receive it and adds the packet to that core’s descriptor ring before sending an interrupt.
Streaming TCP performance

Experiments were conducted using netperf’s TCP_STREAM benchmark to determine the effect of IOMMU usage and the new vmem hierarchy driver design in particular on high-throughput networking. The benchmark performs bulk data transfer over TCP from a transmitter to a receiver. The machine under test was evaluated as both the transmitter and the receiver, running netperf in the transmit case and netserver in the receive case. For each case, netperf was used to measure throughput in Gbits/s and CPU utilization as a percentage. Since the machine has four cores, each core makes up 25% of the total CPU utilization. Various forms of IOMMU usage were examined: no-iommu indicates that the machine under test was run with IOMMU support disabled, stock-strict indicates that stock FreeBSD IOMMU support was enabled with strict IOTLB invalidation, vmh indicates that the stock FreeBSD IOMMU driver’s IOVA allocation and IOMMU page table management subsystems were replaced with the vmem hierarchy design, and ptwalk indicates that the vmem hierarchy design was augmented with the scheme to eliminate software page table walks when mappings are created and destroyed.

The results of running the TCP_STREAM experiments with a single instance of netperf and netserver are presented in Table 5.1. In the no-iommu baseline, both transmit and receive throughput are near line rate. Receive has higher CPU utilization because, in general, the receive side has to perform more processing per byte of throughput than the transmit side due to network limitations on received packet sizes. Compared to the no-iommu baseline, the stock-strict design reduces transmit throughput by 66.9% and receive throughput by 92.45%. The transmit side results, including the CPU utilization increase, can be explained by the fact that the packet-enqueueing thread must call map on every packet, and the interrupt-handling thread
must perform expensive IOTLB invalidations during `unmap` calls where it previously simply freed memory. The receive side results can be explained by noting that the single throughput-critical thread must make both `map` and `unmap` calls while receiving packets, which reduces the rate at which packets are made available for processing by the network stack. This rate reduction leads to a significant drop in throughput and CPU utilization.

Recall that the stock FreeBSD driver uses a red-black tree IOVA allocator that shares the same coarse lock as its page table management code. IOTLB invalidation uses a different lock. Relative to the stock driver, the vmh and ptwalk designs significantly increase throughput for both paths without affecting CPU utilization. This result suggests that both the transmit and receive sides are bottlenecked by CPU-saturated threads. In particular, the amount of work required to transmit and receive packets saturates the throughput-critical threads (one thread on the receive side and two on the transmit side) that control the rate at which packets can be received or transmitted. When the cost of performing IOVA- and page table-related operations is reduced by the vmh and ptwalk designs, those throughput-critical threads are able to process more packets per unit time, which explains the throughput increase, but remain CPU saturated, which explains the lack of change in CPU utilization numbers.

Overall, the designs presented in this section resulted in single-core throughput improvements of 1.85x on the transmit side and 3.28x on the receive side compared to the stock FreeBSD driver while maintaining the same CPU utilization. The vmem hierarchy changes are responsible for the performance gains, while the elimination of software page table walks did not contribute significantly in the single-core case. Interestingly, a pathology in the FreeBSD DMA KPI prevented our design’s single-core transmit performance from reaching that of the no-iommu case. In particular, the
FreeBSD DMA KPI has a particular function for exposing mbufs to peripheral devices. If an mbuf chain, which is a linked list of mbufs, is passed into that function, it will make calls to the IOMMU driver functions to create separate mappings for each mbuf’s data instead of combining the data into a single virtually contiguous region. Mbuf chains can regularly approach lengths of 15 or more mbufs on the transmit side, and this design choice increases the number of IOMMU-related operations, including IOTLB invalidations that need to be performed, by a factor of fifteen in these cases. Changing this behavior to combine data from an entire mbuf chain into a virtually contiguous region results in a transmit side throughput of 35.75 Gbits/s, almost on par with the no-iommu case. Mbuf chains on the receive side only consist of one mbuf, so this change does not improve its throughput.

The results of running multicore TCP_STREAM experiments are in Table 5.2. In these experiments, four instances of netperf and netserver were run on the transmitter and receiver. They were pinned to separate cores, and each instance of netperf sent data to a particular instance of netserver. These steps were carried out in the interest of reproducibility of results and because netperf is single-threaded.

We see in the baseline results that both the transmit and receive cases are able to get slightly more throughput out of the NIC than in the single core case. With all four cores transmitting or receiving packets, the system takes advantage of the driver’s multicore support, and CPU utilization increases significantly. Interestingly, though the receiving system could theoretically make use of all four cores to process received packets, NICs generally send packets from the same flow, or connection, to a single core to take advantage of locality in the network stack code. I confirmed empirically that the machine under test received most packets on a single core when only one instance of netserver was running.
<table>
<thead>
<tr>
<th>Design</th>
<th>Transmit Throughput (Gbits/s)</th>
<th>CPU Util. (%)</th>
<th>Receive Throughput (Gbits/s)</th>
<th>CPU Util. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-iommu</td>
<td>36.44</td>
<td>34.34</td>
<td>36.56</td>
<td>49.08</td>
</tr>
<tr>
<td>stock-strict</td>
<td>12.06</td>
<td>65.65</td>
<td>2.76</td>
<td>29.98</td>
</tr>
<tr>
<td>vmh</td>
<td>22.05</td>
<td>68.29</td>
<td>8.83</td>
<td>32.23</td>
</tr>
<tr>
<td>ptwalk</td>
<td>22.34</td>
<td>67.71</td>
<td>9.05</td>
<td>32.59</td>
</tr>
</tbody>
</table>

Table 5.1: Single-core netperf TCP_STREAM performance results.

<table>
<thead>
<tr>
<th>Design</th>
<th>Transmit Throughput (Gbits/s)</th>
<th>CPU Util. (%)</th>
<th>Receive Throughput (Gbits/s)</th>
<th>CPU Util. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-iommu</td>
<td>37.61</td>
<td>82.71</td>
<td>37.63</td>
<td>79.39</td>
</tr>
<tr>
<td>stock-strict</td>
<td>11.84</td>
<td>62.82</td>
<td>2.3</td>
<td>57.88</td>
</tr>
<tr>
<td>vmh</td>
<td>22.52</td>
<td>73.94</td>
<td>12.49</td>
<td>61.97</td>
</tr>
<tr>
<td>ptwalk</td>
<td>24.76</td>
<td>78.11</td>
<td>15.61</td>
<td>53.68</td>
</tr>
</tbody>
</table>

Table 5.2: Multicore netperf TCP_STREAM results.
The stock FreeBSD system is unable to scale because none of the IOMMU driver subsystems scales. Threads in the NIC driver attempting to call `map` and `unmap` compete with each other not only to perform IOTLB invalidations but also to handle IOVA allocation and page table management. This contention reduces the rate at which any one thread can complete a `map` or `unmap` operation, thereby reducing the throughput since throughput-critical threads involve these operations. The new design demonstrates some scalability because the IOVA allocation and page table management subsystems scale. We observe different scaling behavior on the send and receive paths.

On the receive path, we see reasonably effective throughput scaling from 9.05 Gbits/s with the ptwalk design in the single core case to 15.61 Gbits/s in the multicore case, a 72.49% increase. This scaling is made possible because the throughput-critical receive interrupt handling threads that compete for the lock to perform IOTLB invalidations must also perform a significant amount of work between IOTLB invalidations. In particular, after an IOTLB invalidation, the corresponding IOVA must be freed, the packet must be processed (for example, by being taken up the network stack), additional empty mbufs may need to be replenished in the receive descriptor ring, and the next packet’s mapping must be removed from the IOMMU page table before the thread initiates another IOTLB invalidation. All of this other work is scalable, so adding additional receive-side interrupt-processing threads allows certain threads to perform useful work in parallel with IOTLB invalidations, thereby increasing the CPU utilization as well as the number of IOTLB invalidations performed per unit time and hence throughput.

On the transmit path, we see the same phenomenon result in worse scaling behavior. Having four pairs of transmit-side threads means that certain of them can perform work in parallel with IOTLB invalidations, thereby increasing the CPU utilization and
the number of IOTLB invalidations performed per unit time. However, these threads perform less non-IOTLB-related useful work than in the receive case: after an IOTLB invalidation, the corresponding IOVA must be freed, the underlying physical memory is freed, and the next packet’s mapping must be removed from the IOMMU page table before the next IOTLB invalidation. Because the transmit path can perform more IOTLB invalidations per unit time in the single core case, it experiences less of a relative improvement in the multicore case than the receive path. We see a 10.83% throughput increase for the transmit path compared to the receive path’s 72.49%.

**TCP RR performance**

Experiments were also carried out using netperf’s TCP _RR benchmark to measure the latency of processing packets in the system. This receive/response test sends one-byte packets back and forth between two machines to measure the number of transactions completed per second. The results for the various designs are presented in Table 5.3. The number of transactions per second drops to 64.0% of that in the no-iommu case when stock IOMMU support is enabled. With the new design and page table walk optimization, it increases to 89.9%. The general trend of these data makes sense, as we see the latency of packet processing go down as the cost of IOMMU-related operations is reduced. However, it should be noted that this benchmark gives results with high variance. TCP was chosen rather than UDP for the receive/response test because the TCP version gave more consistent results.

**Page table memory usage**

To evaluate the amount of memory used for IOMMU page tables by the new design, the number of page table pages used for the NIC was tracked over the course of
## Table 5.3: Netperf TCP RR results.

<table>
<thead>
<tr>
<th>Design</th>
<th>Trans/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-iommu</td>
<td>48553.53</td>
</tr>
<tr>
<td>stock-strict</td>
<td>31078.07</td>
</tr>
<tr>
<td>vmh</td>
<td>43366.97</td>
</tr>
<tr>
<td>ptwalk</td>
<td>43631.36</td>
</tr>
</tbody>
</table>

many netperf TCP_STREAM benchmark runs until it reached a steady state. The stock FreeBSD IOMMU driver, which does reclaim unused page table pages, had a steady-state page table of 29 page table pages, or 116KB. The new design used 23 page table pages, or 92KB, a 20.7% reduction. Our design with page table reclamation disabled used 32 page table pages, or 128KB. This result suggests that the new design’s method of allocating IOVAs is more amenable to page table reclamation than that of the stock driver, and that its page table reclamation strategy is effective overall. Recall that the vmem hierarchy design is the first to provide scalable page table management with unused page reclamation.

### Conclusion

Evaluating the novel IOMMU driver design demonstrated that reducing the performance cost of IOMMU-related operations led to significant improvements in I/O throughput and latency for a high-performance NIC compared to the stock FreeBSD driver. We saw throughput improvements of up to 6.79x and a 40.39% improvement in the number of receive/response transactions per second. The new design also has a number of qualitative improvements over the state of the art and manages page table
memory more effectively than existing designs. We saw a 20.7% reduction in page table memory usage compared to the stock FreeBSD driver. Improvements made to IOMMU driver functions are important, as these functions will be called no matter how a system makes use of the IOMMU. Improving the scalability and efficiency of strict IOTLB invalidation would be valuable future work in this area. However, as the complicated use of these IOMMU driver functions by the NIC driver has demonstrated, improvements to the IOMMU driver alone cannot bring the cost of IOMMU usage down to a level acceptable for general-purpose use. In the coming sections, we will examine how systems might make use of the IOMMU driver differently to further improve performance and address security vulnerabilities.

5.2 DMA KPI changes

This section presents additions to the DMA KPI. New KPI functions enable performance improvements without sacrificing security properties by using the IOMMU in novel ways. For example, we will see that exposing asynchronous IOTLB invalidations via functions with clearly defined semantics enables batched IOTLB invalidation while addressing the non-strict IOTLB invalidation vulnerability described in the previous chapter. We will also see how DMA KPI functions created specifically for immutable memory could reduce the cost of IOMMU usage in important cases like the sendfile system call without giving up security properties.

5.2.1 Asynchronous unmap

Modifying the DMA KPI to support asynchrony could make IOMMU use more efficient for important I/O workloads while addressing the non-strict IOTLB invalidation vulnerability. The DMA KPI functions proposed to support asynchrony are async_map
and `async_unmap`. These functions accept the same arguments as their synchronous counterparts as well as a callback function and a callback argument. Both asynchronous functions operate on a queue of descriptors that represent IOMMU operations to be completed asynchronously. When `async_map` and `async_unmap` are called, they simply enqueue a descriptor that stores their arguments and return to their calling contexts. When a queue becomes full, or the additional proposed KPI functions `flush_map` and `flush_unmap` are called, a separate thread processes the specified queue one element at a time. In particular, for each descriptor in the queue, the thread performs the appropriate mapping or unmapping IOMMU driver operations then calls the descriptor’s callback function once the operations have fully completed. This thread can dramatically reduce the overhead of unmapping by performing a single batched global IOTLB invalidation after all of the descriptors’ mappings have been removed from the IOMMU page table.

By waiting to execute callbacks until after the corresponding mapping or unmapping IOMMU driver operations have completely finished, this asynchronous design enables improved performance while offering the same security properties as a synchronous design with strict unmapping semantics. For example, a driver could use asynchronous unmapping to take advantage of batched IOTLB invalidation but ensure that memory is not reused until after it has been unmapped by freeing it in the asynchronous callback function. In addition to the performance benefit of batched IOTLB invalidation, the asynchronous design can also improve throughput by changing the context in which IOMMU-related work is performed. Many high-performance I/O workloads have certain threads that execute throughput-critical loops, as we saw in the previous section. Offloading IOMMU-related work to a different thread that runs in parallel with the performance-critical loop can increase overall throughput. The `flush_map` and
flush_unmap functions allow work queues to be flushed by a DMA KPI consumer at any time, which reduces concerns about increased latency when using the asynchronous KPI functions. These asynchronous functions can also be adopted incrementally, so that certain devices drivers could benefit from them without requiring a complete rewrite of all existing drivers.

Evaluation

Because the throughput-critical thread in the receive path of a NIC is a packet processing loop that performs an expensive unmap operation during each iteration, the receive path could benefit significantly from using the proposed async_unmap function. To evaluate the effectiveness of such a change, experiments were performed using the same setup as in Section 5.1.5 above. An implementation of async_unmap was added to the DMA KPI on the machine under test. The receive path of the NIC driver was modified so that its throughput-critical thread simply called async_unmap on each mbuf with a callback that passed the mbuf to the network stack appropriately. The asynchronous queue was flushed (and batched IOTLB invalidations were performed) either when it filled up or when the interrupt handler ran out of received packets to process. This design ensures that no packets will be left unprocessed in the queue after the interrupt handler returns. Additional strategies for flushing the queue, such as performing a flush multiple times during the interrupt handler’s execution or after some timeout, should be explored.

Table 5.4 evaluates the effect of using async_unmap on throughput and latency by comparing its single-core receive-side netperf TCP_STREAM and TCP_RR results to those obtained by evaluating the designs from the previous section that used unmap. The async-unmap design uses the vmem hierarchy IOMMU driver as well as the new
### Table 5.4: Single-core receive-side asynchronous unmapping evaluation.

<table>
<thead>
<tr>
<th>Design</th>
<th>Throughput (Gbits/s)</th>
<th>CPU Util. (%)</th>
<th>Trans/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>no-iommu</td>
<td>36.56</td>
<td>49.08</td>
<td>48553.53</td>
</tr>
<tr>
<td>stock-strict</td>
<td>2.76</td>
<td>29.98</td>
<td>31078.07</td>
</tr>
<tr>
<td>vmh</td>
<td>8.83</td>
<td>32.23</td>
<td>43366.97</td>
</tr>
<tr>
<td>ptwalk</td>
<td>9.05</td>
<td>32.59</td>
<td>43631.36</td>
</tr>
<tr>
<td>async-unmap</td>
<td>17.25</td>
<td>61.52</td>
<td>37465.52</td>
</tr>
</tbody>
</table>

**async_unmap** DMA KPI function. We see that asynchronous unmapping increases receive-side throughput by 90.6%, which is due to the fact that the receive interrupt-handling thread can process received packets much more quickly and in parallel with the asynchronous thread that performs the unmapping. The number of IOTLB invalidations performed overall is also greatly reduced by batching. CPU utilization increases as expected when adding a work-intensive asynchronous thread to the system. The number of transactions per second in the receive/response benchmark decreases by 14.1% relative to the ptwalk design, which is also expected because packets on the receive side are not processed until the asynchronous queue becomes full or gets flushed. Similar but less-pronounced effects could be achieved by adopting **async_unmap** on the transmit path, since it already offloads expensive IOTLB invalidations to the transmit interrupt thread.

### 5.2.2 Sendfile optimization

Sendfile is a system call that sends a given file over a specified stream socket. It is known for efficiency, and many high-performance applications such as the Nginx webserver and Netflix’s video server use it to transmit data. In particular, sendfile
implements a zero-copy design in which file data is read from disk into an immutable (read-only) cache then sent directly over the specified socket. The data is usually read directly from the immutable memory by a NIC via DMA. This design avoids copying to and from userspace as well as duplicating cached file data in a socket buffer. Currently, high-performance systems that serve data with sendfile do not use the IOMMU because of its significant performance cost. Adding functions `map_immutable` and `unmap_immutable` to the DMA KPI could significantly reduce the performance penalty of IOMMU use for sendfile and APIs like IO-Lite [11].

The `map_immutable` and `unmap_immutable` functions are designed to govern peripheral device access to an immutable region of system memory. In particular, they make use of the insight that mappings to immutable memory can be left open even after a peripheral device driver unmaps them. Because the memory is immutable, leaving the mappings open does not expose any new information to peripheral devices. The arguments to `map_immutable` and `unmap_immutable` are identical to those of `map` and `unmap`, except `map_immutable` does not accept arbitrary memory permissions. When `map_immutable` is called, it maps the given region of memory into the specified I/O virtual address space with read-only permissions. It then (1) wires the underlying physical pages to ensure they are not swapped out before the device accesses them (this step is also performed in the synchronous map operation), (2) marks the underlying physical pages with the flag `iomapped` and their corresponding IOVAs, and (3) returns the IOVAs. If `map_immutable` is called on some physical page that already has the `iomapped` flag set, the physical page is not mapped a second time. Instead, `map_immutable` wires the page if necessary and returns its existing IOVA. When `unmap_immutable` is called, it does not actually unmap the memory region from its I/O virtual address space. It simply unwires the underlying physical pages.
different context, when the page daemon attempts to reclaim (i.e., free) an unwired physical page that has the \texttt{iomapped} flag set, it must first unmap that page in the IOMMU.

This addition to the DMA KPI would allow mappings to immutable memory to be reused without any IOMMU overhead until memory pressure prompts the page daemon to reclaim the underlying pages. Workflows like sendfile could benefit greatly from this change, and experiments should be carried out in the future to determine its impact quantitatively. With the existing DMA KPI, sendfile must have \texttt{map} and \texttt{unmap} operations performed each time an immutable buffer is transmitted via the NIC. Simply calling the \texttt{map} operation and leaving the mappings open would also leave the underlying pages wired, which blocks their reclamation by the page daemon. This constant mapping and unmapping leads to unacceptable performance overheads. Replacing the calls to \texttt{map} and \texttt{unmap} with calls to \texttt{map_immutable} and \texttt{unmap_immutable} would allow IOMMU mappings for the immutable buffers to be reused with no IOMMU overhead. It would also allow active mappings that are not in use by a device to be unmapped and reclaimed by the page daemon, thus reducing memory pressure concerns. In a situation where files are completely served from in-memory caches, this KPI change could remove almost all IOMMU-related overhead from the steady-state sendfile workflow. High-performance servers could then enable IOMMU protections without affecting their data transfer capabilities.

5.3 Memory management techniques

This section will introduce techniques for managing memory that is exposed to peripheral devices via the IOMMU. While the previous two sections focused on IOMMU driver design, which is concerned with software that controls IOMMU-related
operations, and DMA KPI design, which is concerned with the semantics of functions that expose those operations to peripheral device drivers, this section will focus on techniques that are concerned with what memory gets exposed to peripheral devices. In particular, these techniques aim to address the problem of leaked sensitive data and control structures caused by the 4KB granularity of IOMMU access control.

5.3.1 Shadow buffers

Shadow buffering is a technique for managing memory exposed to peripheral devices via the IOMMU proposed by Markuze et al. in 2016 [37]. Under a shadow buffering scheme, each device has pools of memory with different access permissions perpetually exposed to it. IOMMU-related operations in the map and unmap DMA KPI functions are replaced with memory copy operations. In particular, when a peripheral device driver calls the map function on a region of memory, that region is copied into a so-called shadow buffer in the device’s appropriate memory pool. When the driver calls unmap, the shadow buffer is copied into the original buffer’s place. Shadow buffering eliminates the vulnerability in IOMMU usage related to protection granularity, since these memory copy operations can be performed at arbitrary granularity. As long as the device driver supplies a buffer that does not contain sensitive information, no sensitive information will be leaked to a peripheral device.

Shadow buffering also delivers I/O throughput and CPU utilization similar to that of other methods that have weaker security properties. In high-performance NIC benchmarks, shadow buffering was demonstrated to generally deliver throughput within 10% of that provided by the current Linux driver with non-strict IOTLB invalidation [37]. For comparison, the current Linux driver has scalable IOVA allocation and page table management with slightly worse performance characteristics than the
vmem hierarchy design presented above. However, shadow buffering and the Linux driver still deliver up to 25% less throughput than the no-IOMMU case. Shadow buffering also negatively affects cache performance for other tasks being performed on the system, a characteristic which was not evaluated in the work that introduced the scheme. An experiment evaluating this negative effect is presented below.

**Evaluation**

The following experiment was performed using the same setup as in Section 5.1.5. To measure the effect of shadow buffering on the cache performance of other system work, the machine under test ran a single instance of the netperf TCP_STREAM benchmark. The netperf instance was pinned to one core, and a SPEC CPU2006 benchmark known to be highly cache-sensitive called h264-ref3 was pinned to another core. The two workloads ran in parallel and shared the same last-level cache (LLC). The number of LLC cache misses suffered by the h264-ref3 benchmark was recorded using the LONGEST_LAT_CACHE.MISS Performance Monitoring Counter (PMC), and its wall-clock execution time was recorded.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>h264-ref3 LLC misses (millions)</th>
<th>h264-ref3 execution time (s)</th>
<th>Netperf throughput (Gbits/s)</th>
<th>CPU Utilization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>h264-only</td>
<td>30.60</td>
<td>352.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>no-iommu</td>
<td>304.96</td>
<td>355.8</td>
<td>37.50</td>
<td>54.48</td>
</tr>
<tr>
<td>stock-strict</td>
<td>219.14</td>
<td>354.6</td>
<td>10.65</td>
<td>73.31</td>
</tr>
<tr>
<td>bounce</td>
<td>1471.08</td>
<td>374.2</td>
<td>31.62</td>
<td>65.32</td>
</tr>
</tbody>
</table>

Table 5.5 : Shadow buffering cache effect evaluation.
Measurements taken for various system configurations are presented in Table 5.5. The h264-only configuration indicates that the h264-ref3 benchmark was run as the only job on the machine (netperf was not run), the no-iommu configuration indicates that the h264-ref3 benchmark was run with netperf and that the IOMMU was disabled, the stock-strict configuration indicates that the h264-ref3 benchmark was run with netperf and that FreeBSD’s stock IOMMU support with strict IOTLB invalidation was enabled, and the bounce configuration indicates that the h264-ref3 benchmark was run with netperf and that bounce buffering was enabled. Bounce buffering is a concrete implementation of FreeBSD’s DMA KPI functions that does not use the IOMMU. Instead, it causes map calls to copy passed-in buffers to a bounce zone and unmap calls to copy buffers back out of the bounce zone. It has the same copying behavior as shadow buffering and thus can be used to approximate shadow buffering’s cache effects. Bounce buffering was initially developed to allow legacy 32-bit PCI devices to access memory above the 4GB boundary by copying it into a bounce zone below the boundary. The presented results for all configurations were averaged over five runs.

The results show that bounce buffering, and therefore shadow buffering, has a significant impact on cache performance. Running the h264-ref3 benchmark and netperf together with no IOMMU support increased the number of cache misses that the h264-ref3 benchmark experienced by 9.97x compared to when it was run alone. Switching from no IOMMU support to stock FreeBSD IOMMU support reduced netperf’s effect on the cache by 28.14%. This result makes sense because the NIC’s throughput was reduced significantly (as also shown in previous experiments). Enabling bounce buffering increased the number of LLC cache misses suffered by the h264-ref3 benchmark by 6.72x compared to the stock-strict case, and the benchmark’s runtime
increased by 5.02%. These results indicate that while shadow buffering’s memory copying makes it an effective strategy for addressing the protection granularity vulnerability, its performance cost may make other schemes more attractive for general-purpose use.

5.3.2 Device memory allocators

The copying overhead of shadow buffering could be avoided by maintaining per-device memory allocators that perform allocations directly from pools of perpetually exposed memory. Routines that currently allocate memory that will be exposed to a peripheral device could instead make calls to the device’s memory allocator and receive memory that the device can already access. In this case, the only overhead of IOMMU use would come from creating the pools and exposing them to devices. This is a fixed cost and could be paid at boot time so that the practical cost of using the IOMMU for high-performance I/O workloads would be very low.

Per-device memory allocators can be used to address the protection granularity vulnerability but not in exactly the same way that shadow buffering does. In particular, if memory to store a certain data structure is allocated from a per-device pool, the entire data structure, including potentially sensitive information or control structures, will be exposed to the device. For this reason, kernel data structures should be allocated from the standard kernel allocators and use external buffers allocated from per-device pools to facilitate device communication. These external buffers will only be co-located on physical pages with other data that have been specifically allocated from the pool, eliminating the chance that sensitive data might unintentionally be leaked to a device. These external buffers will always be exposed to devices and cannot be unmapped, so additional steps should be taken at the application level to
ensure that devices cannot use this perpetual access to subvert system protections. For example, a NIC might be able to modify fields of a packet after they have passed firewall checks, and steps should be taken to prevent this behavior.

Some scenarios are not well suited to using per-device memory allocators. In the sendfile example discussed above, memory to cache file data should not come from a particular device’s memory allocator because it must be exposed to both a NIC and a storage controller. Similarly, a machine acting as a network bridge may want to expose the same data to multiple NICs. Other techniques discussed in this chapter could be used to improve performance and address vulnerabilities in these scenarios.

5.4 Discussion

This chapter presented a number of strategies for improving the performance of IOMMU usage and addressing the vulnerabilities presented in the previous chapter. First, a novel IOMMU driver design was introduced. It combines the tasks of IOVA allocation and page table management into a single subsystem to achieve improvements in performance and ease of implementation. The idea of using a structured resource allocation system to implicitly manage some underlying data structure is novel and could also be applied in other scenarios. The new design was shown to improve the performance of a 40Gbit/s NIC with IOMMU protections enabled significantly compared to the stock FreeBSD IOMMU driver.

Novel DMA KPI functions that expose different IOMMU mapping semantics to peripheral device drivers were also presented. These functions can be used to improve the performance of certain workloads without sacrificing security properties. An asynchronous version of the unmap function was shown to greatly improve receive-side performance for a 40Gbit/s NIC while simultaneously addressing the non-strict IOTLB
invalidation vulnerability. Functions to support the sendfile system call efficiently were also introduced.

Finally, techniques for managing memory to be exposed to peripheral devices were considered. These techniques can address the protection granularity vulnerability and potentially reduce the performance cost of IOMMU-based protection. The shadow buffering scheme introduced by Markuze et al. in 2016 was presented and evaluated. It still involves a nontrivial performance cost and was found to result in negative cache effects for other work running on the system. Per-device allocators may significantly reduce the cost of IOMMU use but are not applicable in all scenarios.
Chapter 6

Conclusions

Though IOMMU usage is widely believed to protect against DMA attacks, this thesis demonstrated that current systems are vulnerable to sophisticated DMA attackers in practice. We developed a model for IOMMU usage in existing systems that protects against all previously published DMA attacks. We further introduced vulnerabilities related to non-strict IOTLB invalidation and IOMMU protection granularity that can even affect systems that instantiate the IOMMU usage model.

We saw that recent trends in hardware interconnects, synthesis, and emulation have increased the feasibility and potential sophistication of DMA attacks. Malicious peripherals can emulate the behavior of legitimate devices, and interconnects like Thunderbolt expose the system bus and hence DMA to devices that can be hotplugged into modern machines. Thunderbolt also supports video, PCIe, and power delivery over the same interface, so attacks from Trojan devices are possible. For example, a projector at a conference venue connected to a target computer via a Thunderbolt dongle could simultaneously project video and completely compromise the machine.

We also examined Thunderclap, a novel FPGA-based DMA attack platform that greatly exceeds the attack capacity of previous such platforms. Using a complex hardware and software stack, it can emulate the behavior of any peripheral device supported by QEMU while simultaneously executing arbitrary attack code to exploit vulnerabilities in IOMMU protections on a target machine. We saw how Thunderclap was used to completely compromise FreeBSD, macOS, Linux, and Windows, even
with their current IOMMU support enabled.

We then saw techniques for improving the performance and security of IOMMU usage in existing operating systems. A novel IOMMU driver design that combines IOVA allocation and page table management improved throughput by up to 6.79x and reduced page table memory usage by 20.7% compared to the stock FreeBSD driver when driving a high-performance NIC. An asynchronous unmapping DMA KPI function addressed the vulnerability related to non-strict IOTLB invalidation and improved receive-side throughput by 90.7% for the high-performance NIC. DMA KPI functions to reduce the performance cost of mapping and unmapping immutable memory were proposed. Techniques for managing memory to be exposed to peripherals were also introduced. They can address the protection granularity vulnerability and potentially reduce the performance cost of IOMMU use.

A system that provides complete protection against all forms of DMA attacks must only expose data that are strictly necessary for communication to devices. In particular, it must not expose any sensitive kernel data or control structures, and it must configure the IOMMU so that no vulnerabilities allow devices to subvert its protections. Data coming from devices must also be validated at the application level to prevent peripheral attackers from inducing undesirable system behavior. This work highlighted shortcomings in existing systems and introduced strategies for addressing vulnerabilities in and improving the performance of operating system IOMMU usage. These strategies could be combined to make strides towards a system that provides efficient and effective protection from DMA attacks.
6.1 Future work

6.1.1 Further improvements in IOMMU usage

Future work should be carried out to determine additional strategies for increasing the performance of IOMMU usage while maintaining security. Strategies that were introduced in this thesis but not evaluated, such as the DMA KPI functions for immutable buffers and the per-device memory allocation scheme, should also be implemented and profiled on existing systems. Finally, a scheme for combining these strategies in a practical way should be developed, evaluated, and incorporated in real systems.

In a different line of investigation, application-level protections could be devised for various I/O subsystems to prevent peripherals from carrying out attacks by presenting malicious data to a target machine. For example, a malicious NIC could carry out an ARP poisoning attack via buffers that were correctly exposed to it for communication. Such an ARP poisoning attack would not violate system security properties but may induce undesirable behavior at the application level. This work would be pursued with the ultimate goal of removing peripheral hardware devices from the trusted computing base.

6.1.2 IOMMU hardware designs

This thesis focused on improving operating systems’ use of existing IOMMU hardware. However, future work might consider how IOMMU hardware can be modified to support more effective and efficient protection. For example, IOMMU hardware that uses a range-based approach rather than a page table to manage translation information could be considered. Such range-based translation has been proposed
for traditional MMUs by Karakostas et al. [31]. Ranges support access control at arbitrary granularity, which addresses the page table protection granularity problem. A small number of large, contiguous ranges could also be used to enforce isolation among virtual machines in a virtualized system. With an IOTLB large enough to manage an effective working set of range-based translations, this method could deliver good performance. Additionally, IOMMU hardware that explores ways to reduce the cost of IOTLB invalidation, for example by implementing self-invalidated mappings [6], should be explored.

Hardware-supported capability systems, such as CHERI [56], provide fine-grained memory protection for software running on the CPU by enforcing bounds and permission checks on pointer dereferences. This protection might be extended to DMA engines, so that peripheral devices receive unforgeable capabilities from the system that provide fine-grained control over how they can access memory. Exploratory work would be required to prototype and evaluate how such a system compares to existing IOMMU-based protection.
Bibliography


[38] Microsoft. Virtualization-Based Security. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs


[46] QEMU. URL: https://www.qemu.org/


