EXPLOITATION FROM MALICIOUS PCI EXPRESS PERIPHERALS

Colin Lewis Rothwell

Churchill College

September 2017

This dissertation is submitted for the degree of Doctor of Philosophy.

This thesis is set in the serif *Crimson* and sans-serif *Linux Biolinum*, both open source. It was produced using the LuaTeX engine for LaTeX, and the memoir document class. As much as possible, I have tried to follow the rules given in Robert Bringhurt's *The Elements of Typographic Style*, which is also my source for the first of the following quotes.

–

> A true revelation, it seems to me, will emerge only from stubborn concentration on a solitary problem. I am not in league with inventors or adventurers, nor with travellers to exotic destinations. The surest – also the quickest – way to awake the sense of wonder in ourselves is to look intently, undeterred, at a single object. Suddenly, miraculously, it will reveal itself as something we have never seen before.

– Cesare Pavese

–

POLONIUS What do you read, my lord?

HAMLET Words, words, words.

(Shakespeare, *Hamlet*, Act 2, Scene 2)

EXPLOITATION FROM MALICIOUS PCI EXPRESS PERIPHERALS

Colin Lewis Rothwell

The thesis of this dissertation is that, despite widespread belief in the security community, systems are still vulnerable to attacks from malicious peripherals delivered over the *PCI Express* (PCIe) protocol. Malicious peripherals can be plugged directly into internal PCIe slots, or connected via an external Thunderbolt connection.

To prove this thesis, we designed and built a new PCIe attack platform. We discovered that a simple platform was insufficient to carry out complex attacks, so created the first PCIe attack platform that runs a full, conventional OS. To allows us to conduct attacks against higher-level OS functionality built on PCIe, we made the attack platform emulate in detail the behaviour of an Intel 82574L *Network Interface Controller* (NIC), by using a device model extracted from the QEMU emulator.

We discovered a number of vulnerabilities in the PCIe protocol itself, and with the way that the defence mechanisms it provides are used by modern OSs. The principal defence mechanism provided is the *Input/Output Memory Management Unit* (IOMMU). The remaps the address space used by peripherals in 4KiB chunks, and can prevent access to areas of address space that a peripheral should not be able to access. We found that, contrary to belief in the security community, the IOMMUs in modern systems were not designed to protect against attacks from malicious peripherals, but to allow virtual machines direct access to real hardware.

We discovered that use of the IOMMU is patchy even in modern operating systems. Windows effectively does not use the IOMMU at all; macOS opens windows that are shared by all devices; Linux and FreeBSD map windows into host memory separately for each device, but only if poorly documented boot flags are used. These OSs make no effort to ensure that only data that should be visible to the devices is in the mapped windows.

We created novel attacks that subverted control flow and read private data against systems running macOS, Linux and FreeBSD with the highest level of relevant protection enabled. These represent the first use of the relevant exploits in each case.

In the final part of this thesis, we evaluate the suitability of a number of proposed general purpose and specific mitigations against DMA attacks, and make a number of recommendations about future directions in IOMMU software and hardware.

To my parents, of course.

DECLARATION

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit of 60,000 words.

Colin Lewis Rothwell
September 2017

ACKNOWLEDGEMENTS

There is a slight danger that any statement that begins 'I would like to thank' takes on the histrionic tone of an Oscar winner's speech. Nevertheless, I would like to thank...

First and foremost, Simon Moore and Robert Watson, my supervisor and secondary supervisor. Despite both being extremely busy, they have always been available to provide me with both guidance and friendly conversation. The level of trust they have put in me to find and complete a difficult research project has often exceeded the level I put in myself. I am particularly grateful to Simon for supporting my interest in hardware from my undergraduate degree onwards; and to Robert for gently steering me towards an area of security that I found interesting, and providing me with the well-founded suspicions about the state of DMA security that motivated my eventual thesis.

My examiners, Andrew Moore and Herbert Bos, for their careful and considered reading of my thesis, and providing suggestions that I believe have resulted in a substantially better dissertation.

Ross Anderson for his detailed reading of my thesis and encouraging remarks.

Mbou Eyole, my supervisor at ARM, for his insightful comments about my work, for giving me the freedom to pursue my interests as they developed, and for his supervision during my placement at ARM.

Theo Markettos, my principal collaborator on this project, for suffering a lot of FPGA-related pain so that I didn't have to and for being willing to check my ideas and work through problems from across the other side of the desk at short notice.

Brett Gutstein and Allison Pearce, for their contributions to the project as a whole, and their writing and proof-reading on our papers. Allison, specifically, for her initial work to confirm the likelihood of vulnerabilities on macOS; Brett for proof-of-concept attacks on macOS.

Jon Woodruff and David Chisnall, for supervising my third year project, which was my first experience of the CTSRD research group, and for making the experience sufficiently enjoyable that I went on to continue in the same group, using the same skills, for my PhD. I also thank Jon for his support for the BERI processor, and David for the opportunity to contribute to the ASPLOS 2015 paper on CHERI and the C programming language.

John Fawcett, my former Director of Studies at Churchill, for encour-

aging me to consider research as an option, and for providing me with many opportunities to teach and conduct interviews, without pressuring me to do too much.

Several fellow students in the Computer Lab, including my one-time office mate Alexandre Joannu, who has shared the stresses of his PhD at the same time as me; and the various people I have eaten lunch or played table football with, for some interesting conversations in many senses of the word, including Alan Mujumdar, Robert Norton, Niall Murphy, Steven Herbert, and Michael Hull.

My friends generally, for keeping me sane. Friends from before university, for irregular meetings that have kept me grounded (I hope). Friends from my undergraduate degree, who have been here the same seven years I have, and people we have met after graduating. Friends that I used to teach, in particular those who joined me as PhD students, Sam Ainsworth and Ian Orton, for their entertainingly singular take on life in the Computer Lab. Sam and Ian also, for providing feedback on the draft.

Anybody I have met because they have been involved with a show with me. The opportunities I've had to perform theatre of various sorts have far exceeded the expectations I had when I joined Cambridge, and it's been a lovely experience of a world that is very different to computer science.

Finally, my parents. Firstly, for struggling through this technical document to provide comments on the grammar, but also for their support in many less tangible ways. They have not had an easy time while I have been doing my PhD – largely for reasons that are not my fault, of course. Despite this, they have always provided me with warmth, wisdom and encouragement, whether in food, visits to or from the dog, or simply by their companionship. Thank you for everything.

CONTENTS

1 INTRODUCTION 13

2 BACKGROUND 19

2.1 Threat Model 21

2.2 Attack Scenarios 22

2.3 PCI Express 25

2.4 Thunderbolt 30

2.5 Network Interface Controllers 30

2.6 IOMMUs 33

2.7 Control Flow Vulnerabilities and Exploits 38

2.8 Vulnerability Taxonomy 41

2.9 Attack and Escalation Techniques 42

2.10 Early DMA Attacks 44

2.11 Existing Attacks against the IOMMU 45

3 KNOWN MITIGATIONS 53

3.1 Deployed Protection 53

3.2 Research 58

3.3 Summary 65

4 THE RESEARCH PLATFORM 67

4.1 Platform Structure 68

4.2 The Terasic DE5-Net and Soft SoCs 70

4.3 The Software Interface 72

4.4 QEMU 77

4.5 Postgres Back end 84

4.6 Summary 86

5 NOVEL ATTACKS 89

5.1 Attacks from a Basic Platform 90

5.2 Custom Configuration Responses 97

5.3 Full NIC emulation 99

5.4 Effective Defences 102
5.5 Summary 103

6 FUTURE DEFENCES 105
6.1 Robust Drivers 105
6.2 Suspicious Behaviour Detection 106
6.3 Cryptographic Handshaking 106
6.4 Encrypted Memory 107
6.5 Finer Granularity Mappings 108
6.6 Spatial Segregation 113
6.7 Bounce Buffering 114
6.8 Address Space Layout Randomisation 114
6.9 Symbol Hiding 114
6.10 Function Pointer Indirection 115
6.11 Novel Peripheral Interconnects 116
6.12 Summary 117

7 CONCLUSION 119
7.1 Contributions 119
7.2 Recommendations 121
7.3 Future Work 122

GLOSSARY 123

Bibliography 131

List of Figures 143

List of Tables 144

INTRODUCTION 1

A traditional model of a personal computer is of a *central processing unit* (CPU), some memory, and a collection of peripherals that allow the computer to interact with the outside world. This model has become increasingly inaccurate, as peripherals are becoming more and more complicated, and are performing increasing amounts of workload-specific computation that would once have been performed by the CPU. In the earliest PCs, peripherals were unable to directly access host memory: they could communicate with the host only in response to requests from the CPU. To transfer data to a device, the CPU had to read from memory into its registers, then write from its registers to the device. Transferring data from the device involved the same operation in reverse. CPUs are not particularly good at transferring data. Their registers are narrower than the width of a modern memory bus, so even though the processor may execute multiple instructions in the time between sequential memory operations completing, it is hard for the processor to saturate the bus. Processors also implement a vast number of features for control flow and arithmetic that are largely superfluous for copying data.

To allow CPUs to spend less time copying memory, and more time performing the complex calculations for which they are designed, certain peripherals were allowed *Direct Memory Access* (DMA), so they could read and write host memory without requiring copy instructions to be performed by the processor. In a familiar story, this is good for performance, but has worrying implications for security. As time has gone on, DMA has become widespread. The ubiquitous standard in internal peripheral interconnects is now *PCI Express* (PCIe), the successor to the *Peripheral Component Interconnect* (PCI) standard, described in Section 2.3. In contrast to its predecessor, PCIe potentially allows every connected peripheral DMA. This allows *confused deputy* attacks, where a device is forced to use its DMA privileges maliciously [35]. Worse, an attacker may be able to directly install a malicious peripheral into the computer itself. With DMA, an attacker can entirely rewrite the kernel in memory, so any protection mechanism that is implemented only in software can potentially be rewritten, and therefore bypassed, itself.

This perhaps does not seem like too big a security concern. After all, any attacker who is willing to use a screwdriver to open a device is clearly very determined. However, the IEEE 1394 standard, which was released in 1995 [39] and implemented by Apple as 'Firewire', provided DMA to peripherals that were outside the computer's case. This led to the development of a number of attacks that were carried out via the somewhat unexpected vector of a hacked iPod running a custom software stack [11]. No truly satisfying defences were deployed against these attacks, with only partial mitigations at best being released. Apple, for example, disabled the relevant driver when the user was not logged in [4]. However, Firewire was never really ubiquitous.

In 2011, Apple and Intel released 'Thunderbolt', another high-speed external peripheral interconnect that granted DMA, covered in more detail in Section 2.4. The same attacks that were possible over Firewire were possible over Thunderbolt, simply by using a Thunderbolt-to-Firewire adaptor [50]. Like Firewire, for several years after its release, Thunderbolt was far from ubiquitous, being found on most Apple computers, but only a few Windows PCs. At the beginning of December 2015, Thunderbolt 3 was released. This dropped the proprietary Thunderbolt connector, and switched to using the USB Type-C (more commonly, USB-C) connector. Consequently, Thunderbolt 3 started appearing on most new mid- and high-end Windows laptops, as well as Apple products.

In the meantime, both AMD and Intel started releasing processors with *Input Output Memory Management Units* (IOMMUs), beginning around 2008. A traditional *Memory Management Unit* (MMU) maps between a virtual address used by a program and a physical DRAM address: an IOMMU does the same thing, except with virtual addresses used by a peripheral. Indeed, deployed IOMMUs greatly resemble conventional MMUs: they employ a *page based* architecture where memory can be remapped in 4KiB chunks, and have an analogue of the *Translation Lookaside Buffer* (TLB) which caches translations in order to improve performance – the IOTLB. Any area of memory that a device does not have a mapping for, it cannot access. Many early attacks operated by rewriting OS code in memory. IOMMUs are covered in more detail in Section 2.6. As code and data are in distinct areas of memory, the IOMMU could protect against these attacks. This level of protection seems to have been enough for the attack community to form a loose consensus that the IOMMU was sufficient to prevent all DMA attacks.

From 2009 onwards, various groups of researchers carried out work that used gaps in IOMMU protection to disable the IOMMU and regain DMA to all of physical memory. These gaps in protection included a time period at boot when the IOMMU did not protect its own configuration structures, forming a *race condition* [59, 60], and the use of interrupts, which were initially not remapped [92]. At around the same time, some research work was done in improving the performance of the OS algorithms that made use of the IOMMU [53, 68]. Some of this work posited that the lack of granularity in the IOMMU would prove to be a security vulnerability, but did not suggest exploits or attempt to carry out any attacks [55]. We describe these attacks in Section 2.10.

Beginning in 2015, we began investigating the possibility of carrying out DMA attacks on modern systems. We first conducted a survey of the IOMMU support provided across several modern operating systems, using system tracing tools to reveal the existence of some probable vulnerabilities [66].

In order to exploit these vulnerabilities in practice, we created an SOC on FPGA, which used the BERI 64-bit open source processor, capable of booting the FreeBSD operating system. We describe the platform in detail in Chapter 4. We used this platform to develop several novel attacks against PCIe, thereby demonstrating that the current level of protection against DMA attacks is inadequate. More information on these attacks is given in Chapter 5. We attacked macOS with an exploit that allowed hijacking of a function pointer along with some of the arguments that function was called with inside the kernel. This attack required finding a kernel symbol that was leaked to the attack platform in order to break the *Kernel Address Space Layout Randomisation* (KASLR – see Section 2.7) which randomises the location of the kernel in virtual memory. We also exploited a feature of PCIe that allows a peripheral to cache translations on a device itself. This was achieved by creating the simplest possible behavioural model of an Intel 82574L NIC that Linux recognised as functional, and modifying the PCIe configuration of this device to report that it was capable of caching translations. This allowed the device to mark DMA accesses as having being translated by the device, allowing full access to host memory in spite of IOMMU protection.

We also exploited the same vulnerability that allowed control flow subversion on macOS on FreeBSD, despite FreeBSD's superior protection. To do this, we extracted a complete behavioural model of an Intel 82574L NIC from the QEMU emulator, and cross-compiled it for BERI. Ordinarily 'DMA' for a QEMU device involves accessing memory running on the emulator platform. We modified the device model so that its DMA requests were sent over PCIe to the real host. We also used this fake-NIC-based platform in order to read data encrypted by IPSEC and sent over the host's real NIC in macOS can be read by an attacker.

Contributions

In this dissertation, we make the following principal contributions:

- We characterise a new threat model for DMA-based attacks (Section 2.1). It is not acceptable to state that all peripherals must be part of the trusted computing base. Peripherals that are granted DMA to a system may have insecure upgrade channels that leave them liable to exploitation, may be sourced from an untrustworthy supplier, or may be plugged into an external Thunderbolt 3 port.
- We create, test, and debug an attack platform to enable a thorough exploration of the space of DMA attacks (Chapter 4). In particular, we simulate an existing, benign PCIe device to allow attacks that rely on higher-level protocol behaviours.
- We conduct the first thorough investigation of relevant literature, covering work in the areas of both attack and defense, fields that each seem to exist without thorough understanding of the other (Sections 2.10 and 2.11; Chapter 3).
- We demonstrate exploits based on novel mechanisms against widely-deployed, modern operating systems (Chapter 5).
- We suggest some mitigations for these vulnerabilities (Chapter 6).

Over the course of my PhD I have made contributions to the following papers:

- *Under Submission* – Markettos, A. T.*, Rothwell, C.*, Gutstein, B., Moore, S., Pearce, A., and Watson, R. *A Thunderbolt from the Blue: Exploiting IOMMU Bypass Vulnerabilities in Operating Systems.* (* These authors contributed equally to the work.)

  The work detailed throughout this thesis forms the majority of this paper.
- *October 2016* – Watson, R., *Rothwell, C.* et al. *Fast Protection-Domain Crossing in the CHERI Capability- System Architecture.* IEEE Micro Volume 36, Issue 5 [88].

  Early in my PhD, I made contributions to the CHERI capability research processor, including improvements to the memory bus, and the development of a floating point unit.
- *March 2015* – Chisnall, D., *Rothwell, C.* et al. *Beyond the PDP-11: Architectural support for a memory-safe C abstract machine*. Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems [18].

  I undertook a partially automated investigation of a large corpus in order to determine the ways in which it used C language pointers that resulted in invalid behaviour with reference to the specification.

Acknowledgements of Technical Contributions

With the exception of the following, the work detailed in this thesis was all carried out by myself:

A. Theodore Markettos The investigation of Microsoft Windows discussed in Section 3.1. The construction of the hardware aspects of the platform covered in Chapter 4.

Brett Gutstein Some investigation of macOS, discussed in Section 3.1. An initial version of a control flow subversion attack against macOS, further detailed in Section 5.1.

Allison Pearce An initial survey of IOMMU use using sytem-tracing tools, covered earlier in the introduction. Some investigation of macOS, discussed in Section 3.1.

A Note on Branding

Much of the work detailed in this thesis is carried out on Intel FPGAs. Formerly, these were Altera FPGAs, but over the course of the work Intel bought Altera, and have rebranded their products accordingly. We use Intel throughout, as this enables information about the products we use to be found most easily online, despite the physical objects themselves carrying an Altera logo.

BACKGROUND 2

A modern PC is a collection of disparate parts produced by different manufacturers. This is in contrast to the early days of computers, where every component of a mainframe was produced by a single manufacturer, so logic for IO could be handled by homogeneous compute units, such as *channels* or *peripheral processors* [8, 19], which we discuss further in Section 2.6. In order for the different peripherals in a PC to communicate with the processor and main memory, they use a standardised substrate, traditionally thought of as a bus. In the same way that peripherals are no longer the simple electrical interfaces that they used to be, this is no longer a particularly accurate representation of the present situation. A modern Intel system, for example, uses an array of specialised protocols for different tasks. The CPU is not connected directly to most peripherals, with this instead delegated to a *chipset*, which Intel call a *Platform Controller Hub* (PCH). The CPU is composed of multiple independent cores that communicate among themselves with a proprietary, nameless ring-based interconnect [85]. A *system agent* on the same chip as the processor cores connects the cores directly to DRAM, and typically includes a PCIe channel to communicate with a *Graphics Processing Unit* (GPU) and a separate proprietary interface to connect to a chipset. In a system with a single-processor chip, this interconnect is *Direct Media Interface*: on a system with multiple physical processors, it is *Quick Path Interconnect*, which connects to the other processors in the system, and to some number of chipsets that are shared between different cores [10]. Figure 2.1 represents a single-processor Intel system diagrammatically.

There are many criteria by which peripheral interconnects can be compared. Probably the simplest is whether the interconnect is external or internal – whether the connector is easily accessible on the outside of the computer case or not. Originally, most external peripherals had simple requirements, and were designed for human-computer interaction. However, there has been an increasing desire for high-performance external peripheral interconnects. One particular impetus for this has been the growth in popularity of laptops, which frequently have

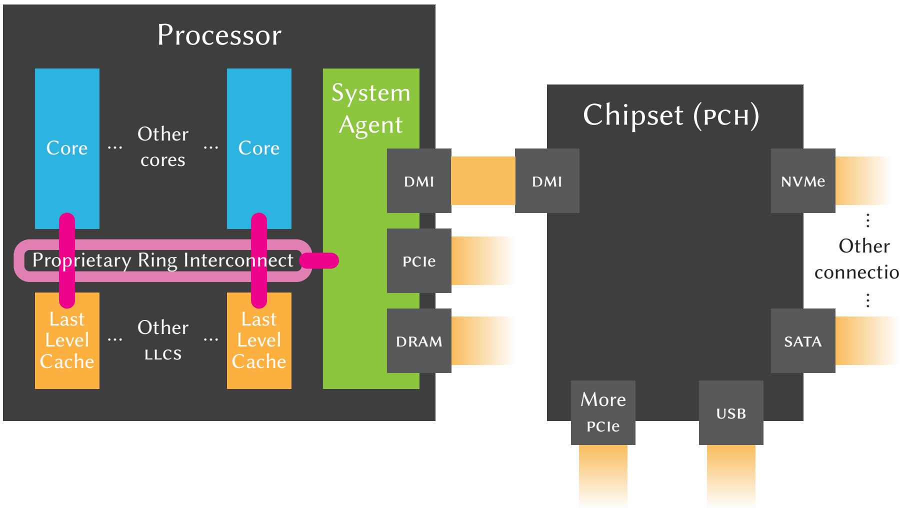


Figure 2.1: Diagram showing some of the different protocols that appear in a single-processor Intel system. If the system had multiple processors, these would be connected with Quick Path Interconnect. Intel's server class processors are beginning to move to a mesh-based internal interconnect, rather than the ring presented here [41].

very little flexibility for internal peripherals. Another is the appearance of external mass storage devices – Firewire was often used on early digital camcorders.

Another criterion for the comparison of interconnects is *Direct Memory Access* (DMA) support. The use of DMA-enabled peripherals provides higher speed data transfer while incurring a lower CPU overhead, but means the peripheral must be trusted. The general approach to this tension between performance and security in the past has been to confine DMA to internal interconnects. The assumption seems to have been that if an adversary has physical access to a machine, then they will be able to subvert it somehow, so a weakness that requires opening up the case with a screwdriver is outside the threat model.

However, as demand for higher-performance external interconnects has grown, we have seen external interconnects with DMA: first Firewire, and now Thunderbolt. When devices were created to subvert machines using the weaknesses in Firewire, only stopgap solutions were implemented (Section 2.10). MacOS prevented the use of Firewire at the login screen to prevent attacks that allowed a user to log in without a password [4]; advice from the attackers recommends uninstalling the drivers that enable DMA [50]. The consensus among the community is that attacks via Thunderbolt can be prevented with the use of an IOMMU [50, 26], but IOMMUs were designed to allow higher performance virtualisation, not to prevent attacks from a malicious adversary [40, 7].

2.1 THREAT MODEL

To begin to develop attacks, we must first characterise a threat model. The field has not converged on a threat model for peripherals in the same way that it has for, for example, memory safety. We have found a range of different views on the threat provided by peripherals across the literature. Much work assumes that a user who has physical access to a machine is able to fully compromise its security, a view shared by the Intel VT-d specification, as discussed in Section 2.6. This presents the IOMMU as providing better performance for virtualisation and protecting against badly programmed drivers; malicious adversaries are never explicitly mentioned. We define a threat model that we believe includes the capabilities of many plausible attackers. Some examples of plausible attackers are discussed in Section 2.2.

The threat model assumed in this thesis is an attacker that:

- Has access to a PCIe slot inside the system or a Thunderbolt port outside it. Access to a PCIe slot may be through firmware override of a device that is already inside the system.
- May make arbitrary valid PCIe transactions.
- Does not have the ability to run any software on the victim computer, although we do mention cases where we hypothesise that lifting this restriction would allow more attacks in the face of more stringent defences.
- When using a PCIe slot, may power-cycle the machine. This is because systems do not generally support PCIe hotplugging, and represents the device being rebooted over its normal course of use.
- Is not able to induce behaviour outside the specification of the computer hardware. In particular, we consider the use of DRAM disturbance, or *rowhammer* attacks (for example, [45, 70, 86]) to be outside the scope of this thesis.

In order to be perfectly secure in the face of such an attacker:

1 The system must not unduly trust unverified peripherals. Verification must be the responsibility of the user, who must ensure that each peripheral in the system is identified according to its purpose.

2 Each peripheral in the system must be granted access to only the areas of address space required to carry out its specified functionality.

3 Any areas of address space that a device has read-access to must not contain data other than that intended for the device while that device has access. Correspondingly, any areas of address space that a device has write-access must not effect the semantics of the running system in any way other than that specified.

4 It must be impossible for a peripheral to impersonate any other device in the system, to ensure that the victim does not attempt to allow that peripheral access to data that is meant for the impersonated peripheral.

A mechanism that provides all of these benefits with an acceptable performance cost may not be possible within the confines of PCIe, or the expectations that users have of their system.

Even with a protection mechanism that provides these guarantees, there are still some plausible peripheral-based attacks. Out-of-scope attacks include those that use legitimate, specified behaviour of the system to induce faults. They also include certain sorts of Trojan attacks. For example, where a peripheral is provided by a manufacturer that carries out its specified behaviour, but also performs some sort of malicious activity. Examples might include NICs that record the data that they transmit, or a webcam that works in tandem with other parts of the system to record the user unexpectedly. Defences against even these attacks may be possible, for example by cryptographic verification of device firmware by a trusted system component; a full analysis is complex, which is why we define it as explicitly out-of-scope. We also exclude attacks where the user, for example, picks up a malicious USB pendrive they find on the ground, and runs a binary they find installed [83].

2.2 ATTACK SCENARIOS

We identify several plausible scenarios in which an attacker could attain the capabilities assumed by our model:

1 *Deliberate production of malicious peripherals for sale to the consumer.* This could be carried out by a particularly unscrupulous manufacturer, either for explicitly malicious purposes, or to enforce a notion of rights control. Large multinutational companies have demonstrated that they are not above this behaviour in the past, as in the case of the Sony BMG rootkit scandal [77]. It may also be that a state actor coerces a manufacturer into the inclusion of backdoors that operate in this fashion.

2 *Accidental production of malicious peripherals for sale to the consumer.* There are many ways in which a manufacturer could produce a device with good intentions that later acted as a vector for an attack. For example, a rogue employee could potentially modify the firmware; the mechanism for firmware updates could be externally subverted; and for sufficiently programmable devices, it is plausible

that an attack can be written to run on the device [24, 96].

3 *Distribution of 'trojan' devices by individuals.* A device does not necessarily only carry out its expected functionality. People are willing to plug peripherals into their laptops with minimal assurances of trustworthiness.

A Thunderbolt dongle for video output at a conference is an obvious example, as one provided by a conference hotel could be easily and discreetly swapped by a guest or employee without difficulty. As conference speakers do not consider the video-output port on their laptop to be a source of vulnerabilities, a single malicious dongle could lead to the compromise of significant numbers of laptops.

The ability of the new USB-C connector to provide power delivery, and data protocols other than USB make even more attacks of this type possible: what appears to a user to be a simple USB device could opportunistically attempt to operate as a Thunderbolt device in order to carry out an attack. We are already seeing socket outlets that provide USB for charging mobile devices. At the same time, laptops that charge over USB are appearing. A malicious state actor could easily provide USB-C charging points in communal locations, such as airports, that install a rootkit before charging an attached device. Any peripheral that uses a USB-C connector could equally carry out the same attacks.

Another attack of this nature, as suggested in [75] is in a workplace where users share computers. A user could install a PCIe card to install a rootkit in a machine that they are currently using.

One impediment to the mass production of attack devices that employ Thunderbolt is its cost. Intel maintains a tight grip on the market, and is the only company manufacturing Thunderbolt controllers. To gain access to developer information, a form must be submitted detailing the proposed Thunderbolt device. Our attack platform, described in detail in Chapter 4, uses an expensive FPGA and development board, and a 'MaxExpansion.com™' PCIe to Thunderbolt dock, which retails at $850. As well as being expensive, the attack platform is bulky and conspicuous.

However, with some ingenuity, it would be possible to create an attack device at a lower cost that was much more discreet. Even Intel's relatively low end FPGAs now include PCIe support. The Cyclone V SX chip, which includes a dual-core ARM Cortex-A9, would probably be able to carry out any of the attacks we present if equipped with appropriate memory and mounted on a PCB. These are available for around $80. A PCIe-to-Thunderbolt bridge can be extracted from an Apple Ethernet to Thunderbolt 2 adaptor, which retail for $29. Apple's Ethernet to Thunderbolt adaptors are principally built around two chips: a Thunderbolt controller and an Ethernet to PCIe chip [78]. These are joined by a short length of cable carrying the PCIe signal. An attack platform could be constructed by cutting this cable in two and connecting it to another device capable of generating PCIe signals. This means that the minimal cost of an attack, excluding man-power and development tools is around $150. With the construction of a custom PCB for mounting the FPGA, it would probably be possible to create a convincing Trojan device in the same space as a Thunderbolt to VGA adaptor. This places attacks on specific targets well within the price range of a determined individual or organisation.

Additional engineering work could lead to a further large decrease in the cost of exploits. Although the operation of Thunderbolt controller chips is secret, the chips themselves are widely available at a cost of around $10. A determined adversary would be able to reverse-engineer the operation of these chips and drive them from a cheap microcontroller. Previous large-scale attacks have shown that criminal gangs are capable of creating hardware-based exploits [73, 31]. It is plausible that special shims and adaptors could be distributed to many vulnerable locations.

Another possibility is an attack that exploits the firmware of a device that is already in the system. This approach was demonstrated in [24]. This could lead to a worm, where a device is uploaded with malicious firmware that uploads itself to other vulnerable devices that it comes into contact with. Firmware-based attacks seem like they will continue to be a concern in the future. The GPUs present in all modern system, whether embedded or discrete are powerful, Turing-complete processors, which are already starting to be used as vectors for malware [96]. This is part of a general trend towards more flexibility in peripherals, also seen in the programmable NICs being introduced to cope with increasing network bandwidth demands. These typically include an FPGA fabric or other specialised architecture to allow packet processing at very high performance. These devices are primarily targeted at high frequency trading but may have applications in other areas. These users are potentially high value targets, so this may already be a serious risk. It seems unlikely that devices of this nature will spread to home users, as they do not require very high bandwidth network access. This may, however, be due to the limitations of bandwidth available. Nvidia are currently trialling a game-streaming server that has a recommended bandwidth requirement of 50Mbps [27], beyond the capacity of most home internet connections.

2.3 PCI EXPRESS

PCI Express (PCIe) is the ubiquitous standard internal peripheral interconnect. It was primarily designed to replace the PCI bus, and deliberately uses a compatible-but-expanded software interface, although its physical layer is very different.

PCIe is designed rather like a network protocol, as a stack of encapsulated layers. Only the *Transaction Layer* of PCIe, which broadly corresponds to the network and transport layers in the OSI model is directly relevant to our work.

At the physical layer, PCIe is a *differential signalling* protocol. This means that data is encoded as the difference in signal between a pair of conductors, rather than the difference between a single conductor and ground. Inside a computer, the conductors are tracks on a PCB. Data on the conductors is encoded with a *line code*. For PCIe versions one and two, this is an *8b/10b* encoding, where 8-bit data words are transmitted as 10-bit signals. For PCIe versions, three and up, a *128b/130b* encoding is used, which has a lower overhead. The use of line encodings brings more favourable electrical properties than just transmitting the data raw.

The basic conceptual unit of the PCIe interconnect is the *link*. A link consists of two single-way channels, each running in a different direction between a pair of components. Links may operate at one of a number of different signaling rates. Each link supports at least one *lane*, which consists of a pair of differential signalling pairs (so four conductors in total) – again, one in each direction. Having multiple lanes per channel enables PCIe to provide different bandwidth at different locations.

These links connect a number of components of different types together. The interconnect can be thought of as a hierarchy: at the top is the *root complex*, which connects the rest of the interconnect to the CPU and memory. The root complex has some number of ports, each defining a separate *hierarchy domain*. Each of these hierarchy domains consists either of a single *endpoint* or a number of endpoints connected together through one or more *switches*.

Endpoints represent what we typically think of as PCIe peripherals – for example, NICs and GPUs – as well as peripherals that are integrated into the root complex, or peripherals that are chips soldered onto the motherboard elsewhere.

In order to maintain backwards compatibility, PCIe switches are represented conceptually as a collection of PCI-PCI bridges, one for each port of the switch. A switch has one upstream port, which connects towards the root hub, and several downstream ports, which connect away from it. Often, upstream ports connect directly to the root hub, and downstream ports directly to peripherals, but they

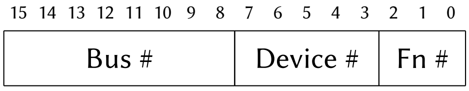


Figure 2.2: PCIe ID Layout. PCIe now supports an 'Alternative Routing ID' format, but we have not seen this used it in practice.

may connect to other PCIe switches or bridges of various types. The ports connect to an internal logical bus that represents the routing logic of the switch in terms of the PCI protocol.

The unit of communication at the transaction layer is the *Transaction Layer Packet*, or TLP. At the top level, TLPs are categorised into *requests* and *completions*. Requests are further categorised based on the address space they target, which is one of memory, IO, or configuration; and whether they are a request to read from or write to the space. Requests may also be of a special type, *message*, which are largely concerned with protocol bookkeeping. Completions may carry data with them, but don't necessarily. A completion is only generated in response to some sort of request. If a request is made to an endpoint that it cannot fulfil in accordance with the specification, then a special completion is returned – an *unsupported request response*.

The behaviour of the configuration space is defined by the PCIe specification, and as such covers behaviour that is common to many different peripherals. The memory space is used for interaction in a device-specific way. Peripherals can access system memory – that is, perform DMA – by making memory requests to the root complex. The IO space is included for backwards compatibility, so while it is not technically deprecated the PCIe specification states that it may be at some point.

TLPs are divided into header and data sections. The header section of a TLP is structured according to the PCIe protocol. It is interpreted both by the component that is the target of the TLP, and also by the switches on the interconnect to make routing decisions. The data section is opaque to the protocol, and is only given semantic meaning by the endpoints. Not every TLP has a data section. Of those that do, the TLPs that interact with configuration or IO space have a fixed-length data section; the TLPs that interact with memory, a variable-length data section.

Configuration requests are addressed with PCIe IDs. These are sixteen bit values consisting of an eight bit bus number, a five bit device number, and a three bit function number, as illustrated in Figure 2.2. This format is inherited from

PCI. PCIe IDs correspond to a physical slot for a device. Each switch 'knows' the IDs of the peripherals attached to it, and only forwards configuration TLPs with the appropriate IDs. Peripherals are expected to derive their IDs by reading the destination field of the TLPs they receive.

The configuration space defines several fields that are common to every endpoint. These include, for example, information about the model number of the device. It also includes two linked-lists of *capabilities*. These are not capabilities in the unforgeable-token-of-authority sense, but rather features that a device has. There are two linked-lists as one is inherited from PCI, while the other is PCIe specific. The inherited linked-list is simply known as the *capability list*; the new one is the *extended capability linked-list*. The configuration address of the start of the capability list is presented by the device in a register in configuration space. The address of the start of the extended capability list is fixed by the PCIe specification.

IO and memory requests are routed by address. Each PCIe peripheral defines a number of *Base Address Registers* (BARs) in its configuration space header. These are used to designate areas of address space that the card inhabits. Somewhat confusingly, the on-device memory that is mapped into host address space starting at the address written into a BAR is commonly referred to simply as a BAR. It is worth noting that on-device 'memory' is something of a misnomer. Devices typically assign complex semantics to *registers* within their memory space. For example, repeatedly reading the same register may result in an internal FIFO being dequeued. When allocating address space for a device, the system probes each BAR in turn. First, it writes the current BAR to all ones. It then reads it back. As the address of a BAR has to be aligned to its size, some number of low bits of the BAR will have been forced to zero. From this, the system can determine the size of the BAR.

In general, a device that is able to make requests of other devices is known as a *master*. In true buses, mastering is a special feature, granted to the CPU and perhaps certain privileged peripherals. This helps to keep down complexity: as soon as multiple devices can make requests, there must be an arbitration scheme to decide which device is permitted to make a request at any given time, and this scheme will typically scale in complexity along with the number of masters in the system. In PCIe, memory requests are just another type of TLP, and can be generated by any device on the system. Decisions about arbitration are made using a flow control system to throttle devices that are unfairly using system resources.

Each PCIe device has a *bus master enable* bit as part of the command register in its configuration space. The PCIe specification states that if the bit is zero, the device is not allowed to issue memory or IO requests. However, it is unclear whether this behaviour is enforced. Root and switch ports also have the bit, and will not forward upstream requests if it is zero, but again, it is unclear if the host makes use of this as an policy enforcement mechanism in practice. While the bit is zero by default at power on, we have found that hosts quickly enable it.

To prevent the processor from having to repeatedly read from a device register to determine if an event of interest has occurred, internal interconnects implement *interrupts*. An interrupt is a signal that a device can trigger in order to disrupt processor control flow so that an event can be handled soon after it has occurred. PCIe has three separate interrupt implementations. The first is a legacy mode, designed for backwards compatibility with PCI, that sends a special message request TLP which is a simulation of a physical interrupt wire on PCI. The other two are both considered current, and are variants of *Message Signalled Interrupts* (MSI and MSI-X). Both operate via generic memory write TLPs to special memory addresses. Information about the level of MSI or MSI-X support that a device has is encoded in capabilities in its configuration space. The use of a memory write packet allows metadata to be carried with an interrupt. This usually encodes the cause of the interrupt as an ID, so writes of different values to the same memory location can signify 'different interrupts'. For example, a NIC might have separate interrupts to signify that a packet has arrived and that it has transmitted the packets assigned to it by the host.

Address Translation Services

The addresses associated with memory requests made to the root complex do not necessarily refer directly to physical locations in memory. They may be translated by an IOMMU, as described in Section 2.1, but this carries with it a performance overhead. To ameliorate this performance overhead, PCIe allows a peripheral to cache its translations on the device itself. This feature is known as *Address Translation Services* (ATS). It is not entirely straightforward. To be able to store translations, a device must be able to request mappings from the host, and the host must be able to communicate to the device when this mapping is no longer valid. A device that supports ATS must present the *ATS Extended Capability* structure, shown in Figure 2.3.

In order to specify that a request has been translated, the ATS specification repurposes two previously reserved bits of the PCIe header to become an *Address*

31 30 29 28 27 26 25 24 23 22 21 20	19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Next Capability Offset	*CV*	ATS Extended Capability ID
ATS Control Register		ATS Capability Register

Figure 2.3: ATS Extended Capability. *CV* is the capability version. The control and capability registers are further divided into subfields. Only relevant here is bit 15 of the control register, the enable bit.

Value	Meaning	Description
`00`	Untranslated.	For backwards compatibility with the reserved bits, which have a value of zero.
`01`	Translation Request.	The host returns the translated value of the address as a read completion.
`10`	Translated.	The address of the transaction has been translated on the device. The host will not pass the memory address of the transaction through the IOMMU.
`11`	Reserved	Host will return an unrecognised request.

Table 2.1: Meanings of the AT field in a PCIe header.

Type (AT) field. The meanings of each possible value of the AT field are given in Table 2.1.

Access Control Services

It was recognised relatively early on that the ability of every component in the system to communicate independently with each other had problematic security implications. In 2005, the 'PCI Express Access Control Services' (ACS) specification was released [65]. This is a set of optional features – in practice, even in 2017 we found implementations only on high-end server machines. The features are:

Source Validation Ports, which either connect directly to a peripheral, or to a bridge, must test that requests they forward have a source ID that is within the range of IDs attached to the port. This is necessary as PCIe makes it the responsibility of the endpoint to set the source ID to a valid value. This mechanism can prevent certain attacks where a malicious device sends TLPs that appear to come from another device on the system [76].

Translation Blocking Makes switches block requests that have a 'non-default' AT field. As the default value is `0`, this blocks all requests for translations, and all requests that are marked as having been translated on-device.

P2P Redirect Features The remaining features combine to force all requests and their respective completions to go through the root complex, even when a more direct peer-to-peer path is available. This prevents devices from illegally accessing peers they are forbidden access to by bypassing the IOMMU, which is found in the root complex.

2.4 THUNDERBOLT

Thunderbolt is a proprietary high performance external peripheral interconnect. It first appeared as part of a consumer product in February 2011. Despite being proprietary, it is known to carry the PCIe protocol, and there are a number of chassis allowing a generic PCIe card to be plugged into a Thunderbolt port.

Thunderbolt versions 1 and 2 saw limited uptake, with ports included in large numbers only on Apple laptops. This may be due Thunderbolt's lack of backwards compatibility with other interconnects, and a lack of a pressing use case. However, Thunderbolt 3 has seen more uptake, appearing in the high-end laptops from a number of manufacturers. Thunderbolt 3 uses the USB-C connector, rather than the Mini DisplayPort connector of previous generations.

Thunderbolt 3 provides a data rate of 40Gbit/s, in comparison to USB 3.1's 10Gbit/s. The use of Thunderbolt and USB on the same port may lead to confusion for the consumer. For example, USB-C ports on a computer are not guaranteed to support Thunderbolt, and cables with USB-C connectors are also not guaranteed to be able to carry Thunderbolt signals.

USB-C ports may also implement *USB power delivery*, allowing up to 100W to be supplied to or from the port. This means that the same port that is used to charge a laptop may also grant DMA.

2.5 NETWORK INTERFACE CONTROLLERS

Some of the exploits we cover rely on an implementation of a *Network Interface Controller* (NIC) so a basic grounding in what NICs are, and how they communicate with their host is pertinent. NICs are an extremely common form of peripheral that allow a computer to connect to a *Local Area Network* (LAN), often with the purpose of then connecting through a *gateway* to the internet. It would be astonishing to find a personal computer of any type without some form of NIC, and increasing numbers of non-personal computers have one to enable them to take part in the 'Internet of Things'. NICs have been around for a long time in the context of computer science, predating the PC. The development of Ethernet, the earliest LAN standard, and predecessor to the most-widely-used current wired LAN, began in 1973; the Cambridge Ring in 1974 [57, 61]. NICs are created by many different manufacturers, and come in many forms, from components in systems-on-chip to discrete integrated circuits on motherboards and pluggable expansions cards operating over PCIe, USB and Thunderbolt. They are built with a wide range of differing requirements. They may implement an old and well-understood technology, needing to make no particular guarantees about performance, or may be an implementation of a bleeding-edge protocol where nanoseconds are important. They can exhibit complex behaviour. Applications on the host and in the network can generate data in totally unpredictable patterns, and the host needs to react to network data, potentially at the same time as having to saturate the NIC with traffic at a rate that may require the transfer of multiple bytes per processor clock cycle.

These factors make a NIC a likely choice for a model peripheral to use to carry out an attack. In order to attempt to handle such a broad spectrum of requirements in a uniform way, the kernel code for interacting with NICs is complex, and therefore liable to suffer from bugs. Before starting implementation, we had a good idea that a feature of the network stack shared across macOS and FreeBSD was likely to lead to vulnerabilities. This is detailed in Section 5.1. On top of this, we knew the QEMU full system emulator had a software model of a NIC that we could potentially transplant. We describe this process in Section 4.4.

Perhaps surprisingly, given the range of different requirements, most NICs use the same basic interface to communicate with the host, although each device has individual differences. The basic mechanism is that of descriptor rings. These come in two broad types: transmit and receive rings. Both work in broadly similar ways. Rings are contiguous areas of host memory that consist of a circular list of *descriptor records*, which contain a small amount of metadata about a fragment of packet to send, consisting of at least an IO base address and a length. The card is told about the locations of the rings by writes to registers in memory space. When the host has a set of packets to send, it populates the transmit ring with pointers to packet data in memory. It then notifies the card that it has data to send by updating the register specifying the end of the ring. The card will transmit the data when it is able, updating the register specifying the start of the ring as appropriate. Depending on the card's configuration, it will trigger an interrupt to

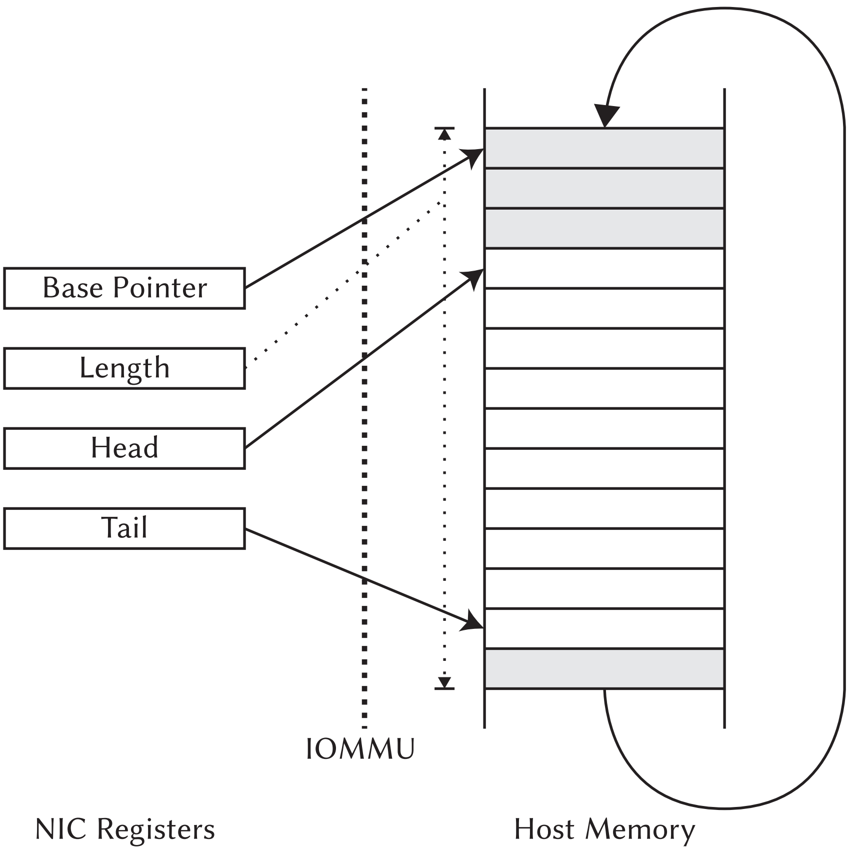


Figure 2.4: Diagram of Descriptor Ring, based on those of an Intel 82574L NIC. The registers contain addresses in host memory as IOVAs. The following descriptions describe the case for a receive ring. Each receive descriptor also contains an address in host memory as an IOVA, a length and some metadata. When the NIC receives data it places it into the buffer described by the head pointer. If this fills that buffer, the NIC advances the head pointer. When the NIC needs more buffer space, the host allocates buffers, places their IOVAs and lengths in the descriptors after the tail pointer, and increments the tail pointer. The buffer is circular: if the tail pointer has a smaller address than the head pointer, the buffer wraps around.

the host when it is done. Receive rings behave in a similar way, except the host populates the rings with buffer addresses that it expects the card to write received data into. This is shown diagrammatically in Figure 2.4: figures similar to this are almost mandatory wherever descriptor rings are explained.

Variants of this simple scheme are possible, and are particularly prevalent in higher performance devices. For example, while packets at the IP layer are not connection-orientated, many IP packets will form part of TCP flows, which are. An advanced NIC may have separate descriptor rings for separate TCP flows and disambiguate between them on the card, rather than the host.

2.6 IOMMUS

A regular MMU maps from virtual addresses to physical addresses: an IOMMU maps from *IO Virtual Addresses* (IOVAs) to physical addresses. Similarly to regular MMUs, IOMMUs have a variety of purposes, including virtualisation and security. The protection offered by a particular usage mode of an IOMMU may be said to be either inter-OS, or intra-OS. Inter-OS protection stops a malicious OS or peripheral running within a hypervisor from attacking other OSs within the same hypervisor; intra-OS protection stops a malicious peripheral from attacking its host.

The idea of a specialised piece of hardware for rewriting IO is not new. The IBM System/360, which was introduced in 1964, used *channels* for IO, highly specialised cores with an instruction set designed to facilitate memory copying [8]. This idea was also seen in the CDC 6600, introduced in 1965, which had a collection of ten *peripheral processors*. These shared execution units, and executed in a *barrel* configuration [19]. In this configuration, each execution unit in the pipeline executes part of an instruction from a successive processor each cycle. This idea is similar to some variants of *multithreading* in modern processors.

There are many differences between the situation then and now. Now, different components of the computer are made by different companies, rather than the computer being built from the ground up by one company. This has led to the replacement of concepts like peripheral processors with logic on the chip that accesses host memory by DMA. Nonetheless, viewing the IOMMU as a device that rewrites IO data arguably makes it a distant descendant of these processors.

One of the earliest actual IOMMUs appeared in Sun's 'sun4m' platform, which was the basis for a series of workstation PCs sold from 1992-1999. Linux has support for this IOMMU, but it is unclear what Sun's rationale was for using it at this time. Early bus protocols supported a narrower address format than some of the PCs that they appeared in. This meant that peripherals could only access a restricted area of the address space. If the kernel allocated buffers in an area the device did not have access to, then the data in the buffers had to be copied to an area the device did have access to. This process, which reduced performance, was called *bounce buffering* [12]. Early IOMMUs are frequently described as a higher-performance alternative to bounce buffering, and improving IO performance may have been the reason behind Sun's IOMMU implementation. Linux uses the Sun

IOMMU by tying it to the API for allocating device memory buffers, so it provides some per-device isolation.

Another early IOMMU was part of the *Accelerated Graphics Port* (AGP) specification, which was released in 1996 and gradually phased out with the introduction of PCIe in 2004. The AGP IOMMU was called the *Graphics Address Remapping Table* (GART). Its purpose was to allow the graphics card to have a contiguous view of large sections of host memory. This simplified access to, for example, texture data, which might have been in physical memory across several non-contiguous pages.

A later IOMMU implementation appears on IBM's xSeries x366 and x460 servers, which appeared in 2005. Again, Linux has support for this IOMMU. The KConfig documentation, which describes the features that can be compiled into the Linux kernel, states that the IBM IOMMU exists to allow 32-bit PCI devices to work on systems with more than 3GIB of RAM, and also that it provides isolation benefits in order to catch bugs and find misbehaving drivers [82]. The IBM servers implementing the IOMMU use commodity x86 processors: the IOMMU is implemented in PCI-X bridges on the chipset. PCI-X is an early successor protocol to PCI. It is a direct extension of PCI, rather than the ground-up rebuild of PCIe. Its precise details are not relevant. Based on the marketing documents available online, the IOMMU was not a marquee feature of these servers.

ARM also have a specification for an IOMMU, called the *System Memory Management Unit* (SMMU). As there are no currently-widely-used ARM systems with external peripheral interconnects that support DMA, we have not thoroughly investigated the SMMU. However, we are beginning to see phones that have a USB-C port into which can be plugged a keyboard, monitor, and mouse to provide a PC-like experience. If this form factor catches on, it seems plausible that Thunderbolt support will be added, and the protection offered by the SMMU will be important.

VT-d

The IOMMU that we focus on attacking is part of 'Intel Virtualization Technology for Directed I/O' (VT-d) [40]. The VT-d specification includes several distinct features: IO device assignment, to give VMs direct access to peripherals; DMA Remapping (DMAR), which includes the IOMMU; interrupt remapping; and 'reliability features'. Based on our analysis of recent processor IOMMU support, detailed in Section 3.1, we believe this to be the most widely deployed implementation of a

PC IOMMU.

Intel suggest a variety of uses for their IOMMU. A number of these are OS use patterns: protecting vital OS internals from all IO devices; the use of 32-bit devices in a 64-bit host; isolating each device, so that it has the minimal necessary access to host memory; and allowing devices to share a virtual address space with user level programs. They also suggest that the IOMMU can be used for mapping a peripheral directly into a *Virtual Machine* (VM) from its *Virtual Machine Manager* (VMM).

To perform remapping, platforms with VT-d provide one or more *DMA Remapping Hardware Units* (DRHUs). Each DRHU is responsible for remapping requests from different devices. DRHUs interpose between peripherals and main memory, performing look-ups of the virtual address provided by the peripheral, and re-writing the transaction with the corresponding physical address if a mapping is provided, or returning an unsupported request response if none is found.

A page-based system, like the VT-d IOMMU, only translates the top bits of the address, translating a *frame number* to a *page number*. The bottom bits, which specify an offset within a page, are the same in the translated and untranslated address.

The DRHUs are configured by a multi-level page table, as shown in Figure 2.5. As these tables configure *DMA Remapping*, they are frequently referred to as the DMAR tables. The top level structure is the *root table*, which is populated with *root entries*. The root table occupies one 4KiB page, and contains 256 128-bit entries. The first step in translating a request from a given peripheral is to index the root table with the request's PCIe bus number, giving a root entry. Each root entry table contains a present flag, indicating that the translations are present for the device, and a pointer to the next table in the hierarchy, the *context table*.

Like the root table, context tables each fill one 4KiB page, and contain 256 128-bit entries. They are indexed by the device and function numbers of the relevant PCIe requester ID. Each entry contains a pointer to a further *hierarchical translation structure*, and various metadata bits. VT-d makes use of the concept of *domains*, non-intersecting areas of address space that a given hardware device may be assigned to – that is, be able to read or write. The context entry metadata includes a *domain identifier*, such that devices assigned to the same domain must have equal domain identifiers, and it is recommended that devices that make use of the same page tables explicitly have identical domain identifiers, 'for best hardware efficiency'.

The root and context tables both support an *extended mode*. These modes

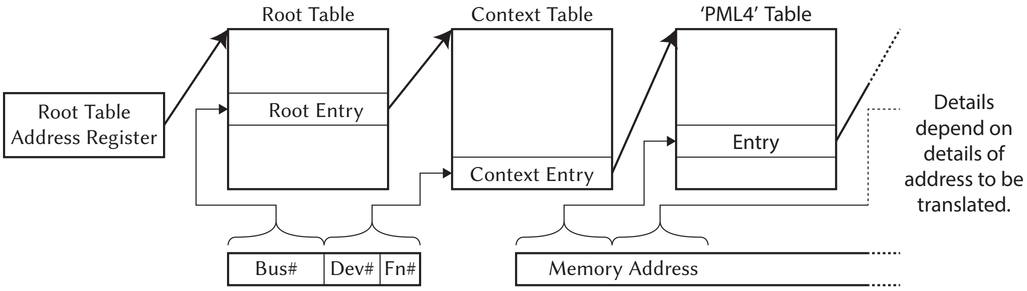


Figure 2.5: Diagram showing the page tables for Intel VT-d. The AMD table layout is similar, but the entirety of the PCIe requester ID indexes one much larger table – the first root table and context table are combined.

use 256-bit entries in order to translate PCIe requests that specify a targeted *Process Address Space Identifier* (PASID), and to hold some additional attributes. In practice, we have not seen these features used.

The layouts of the hierarchical translation structures vary. Firstly, systems can have different address widths, and smaller addresses require fewer levels of table to translate. Additionally, VT-d has support for *large pages*, which are 2MiB or 1GiB in size. Each step of larger page size reduces the number of tables that need to be indexed by one, with the possible sizes of large page deliberately chosen for this to be the case. The largest – and indeed, standard – number of tables that need to be walked to carry out the translation is six, including the two tables to find the per-device hierarchical translation structure. This corresponds to two translations to find the address of the hierarchical translation structure from the PCIe ID, then the use of four successive 9-bit chunks of IOVA to index successive levels of the hierarchical translation structure, which results in a 36-bit physical page number, which is in turn spliced with the bottom 12-bits of the IOVA, resulting in a 48-bit physical address.

Performing a six-level page lookup for each request would carry a large performance penalty, so VT-d specifies several varieties of cache in order to improve performance. The two most relevant are the *context cache* and the IOTLB. The context cache caches context entries, which are used to specify the hierarchical translation structures for a device based on its source ID. The IOTLB caches translations from IO virtual frame number to physical page number. IOTLB cache entries are tagged by source ID to allow translation requests from different devices to be disambiguated, and by the upper bits of the requested address. They may also be tagged by domain ID to allow batch invalidation of entries associated with a given domain.

If a device attempts to carry out a translation for which no mapping exists – a *translation fault* – the iommu returns with an unsupported request response. It also records some data about the fault in a *fault recording register*: the iommu can have an implementation specific number of these, from one per drhu up to 256. An implementation may also support *advanced fault logging*, which writes to a log in host memory. On a fault, the iommu also raises an interrupt to the host. The iotlb can cache translation faults, meaning that it explicitly remembers that requests with specific tag bits are invalid.

There are two mechanisms for invalidating cached translation entries: a legacy register-based interface, and the current *queued invalidation interface*. These cannot both be active at the same time. Both mechanisms allow all entries to be revoked simultaneously, all entries in a given domain to be revoked simultaneously, and for individual pages to be revoked sequentially. The register-based interface operates synchronously, while the queued invalidation interface is asynchronous and uses a circular buffer of *invalidation descriptors* in host memory.

The specification does not suggest specific designs for the iotlb. Instead, this is left up to the implementer and hidden from software.

vt-d specifies support for 'Device tlbs', in accordance with the pcie ats specification, described in Section 2.3.

amd-v

amd also have have an iommu specification, and a number of partial solutions to the same problems solved by an iommu, which predate their system-wide iommu. These include the agp gart, as described in Section 2.6, and also a feature called the *Device Exclusion Vector* (dev) [7], which first appeared in 2005. The dev added a mechanism whereby the host bridge, which connects the cpu to high performance peripherals and ram, can mark areas of memory as being accessible to either all devices or no devices. The dev is implemented as a bit field, where each bit corresponds to a 4kib page of physical memory. When a request comes in, the appropriate bit is looked up and checked: if the bit is unset, the request is dropped. A register specifies the location of the bit field in memory.

This mechanism has some appealing features: it is simple and easy to understand, and the metadata corresponding to a given page is located in only one location and can be found quickly. It does have several disadvantages when compared to a fully fledged iommu, however. The two principal downsides are that it does not allow address remapping or per-device exclusion. These limitations will be explored in more detail in Section 3.1, as they are reflected in the way that certain OSs use the IOMMU.

Both DEV and AMD's full IOMMU fall under the umbrella of virtualisation technology, which AMD shorthand as 'AMD-V'. AMD's rationale for their IOMMU includes the use of 32-bit devices on 64-bit systems. They also claim that it provides 'more secure' use of peripherals in systems running VMs, and allows use of a peripheral to be granted directly to a user-level application. They specify that the IOMMU may have support for Device TLBs.

The AMD IOMMU is almost identical to Intel's. The principal difference is that they have only one table mapping from PCIe device IDs, rather than two levels of table. It uses 256-bit entries, and may be up to 2MiB in size in order to cover the whole PCIe space. This sacrifices memory space efficiency for eliding one memory read in the IO page table lookup. Given that most modern PCs have many gigabytes of memory and memory accesses are comparatively slow, this seems a reasonable trade off. Other than this, the two designs are essentially the same, at least to the level of detail that we have covered VT-d. The AMD IOMMU is also controlled via a circular buffer, which is primarily used to invalidate entries. *Events*, including translation faults, are reported in an *event log* in system memory. The IOMMU can be configured to trigger an interrupt in response to the occurrence of an event.

Again, there is support for the implementation of an IOTLB. AMD explicitly say that they expect most will use a set-associative structure, but they are not prescriptive. They recommend a two-stage organisation: the first stage mapping a device ID to a domain and IO page table base address, the second mapping a domain ID and device virtual address to a system physical address and protection. They recommend that the former can be small, but should have high associativity or a very good hash to spread the clustered nature of device IDs, and the latter should be large. They also recommend the use of the domain ID in hashing so that multiple domains with similar layouts don't compete.

2.7 CONTROL FLOW VULNERABILITIES AND EXPLOITS

In our work, we use a malicious peripheral to modify the control flow on the processor. At first glance, being able to force one return statement or function pointer to branch to a different location is not necessarily a severe vulnerability. However, there is a long-running and on-going strand of research concerned with escalating minimal control of a processor into the ability to execute arbitrary code. Defences designed to prevent these attacks are also effective against some of our exploits, and the approaches to privilege escalation they use are also relevant to our work.

A widespread class of early control-flow-hijack attacks are *buffer overflow* attacks [21]. These operate by finding a string input to a program that is not properly length checked. A malicious user can input a string longer than its allocated buffer, which overflows and overwrites the return address on the stack. When the CPU returns from the currently executing function, it branches to an address of the user's choice. As part of their payload, the user could include code that they wanted to run, and set the return address to point to this code.

Various mitigations have been implemented to thwart these attacks. One of the simplest is *Write xor Execute* (W $\oplus$ X). This uses the conventional MMU to ensure that either a page can be written to by user-mode code, or code located on the page can be executed, but not both. This prevents an attacker from being able to execute any code path that isn't already included in the running binary.

It turns out that this is not a sufficient protection measure. In the late 90s, a patch was made available for Linux that introduced the concept of a *non-executable stack*, with similar protection being implemented on Sun's Solaris. Attacks against these measures appeared on mailing lists, bug-trackers, and 'underground' hacking magazines [23, 91, 56, 62]. These attacks revolved around the idea of subverting a return instruction in order to carry out an unexpected function call, rather than to branch to user-provided code. As the usual targets were functions in libc, these attacks are often known as *return-into-libc* attacks. An example attack sequence writes some code to be executed into an area of address space, then uses the `mprotect` system call in order to switch the area's permissions from 'write' to 'execute', then branch to this area.

This work did not see much interest from the academic community until 2007, possibly because it was not published in a traditional venue. In 2007, the approaches taken by return-into-libc attacks were generalised into *Return-Oriented Programming* (ROP) attacks. [79, 72]. These assume an attacker with the ability to add several addresses to the return address stack, which is possible by exploiting a buffer overflow, and a target system running a known binary. In a ROP attack, the attacker scans the target binary for *gadgets*. These are short sequences of code followed by a return statement. The program that the attacker wants to run is broken down into a sequence of gadgets, then the addresses of those gadgets are pushed to the stack in reverse order, so that the address of the piece of code that should be executed first appears at the top. Then, the attacked function is allowed to return. Instead of returning to the caller, the CPU reads the gadget's address from the top of the stack, and branches to and executes that code. When that has executed, it calls return, and so the next address executed is that of the next gadget, and so on. It turns out that in practice even quite modestly sized binaries contain enough gadgets for arbitrary code execution – a set of gadgets that provide Turing completeness. A standard approach is to carry out a ROP attack against the Standard C Library, which is included in almost all C language applications. Gadget search is helped by the x86 architecture's variable length instructions, which means that the processor can be made to branch to the middle of another instruction, and interpret it to be an entirely different instruction.

Some early defences against ROP attacks operated by either detecting return instructions that occurred more frequently than would be expected in legitimate code, or by detecting return instructions that were executed without a corresponding call instruction having been issued. However, in 2010, Checkoway et al. observed that an actual 'return' instruction is not actually necessary to carry out a ROP-style attack [17]. They observe that the properties of the return instruction that enable ROP attacks are that they transfer control of execution by an indirect jump, and that they update processor state so that a subsequent return will not transfer execution back the expected path of the program. Sequences of instructions that have these properties and are not return instructions occur far less frequently than returns, so it is not possible to find a set of gadgets that end in an appropriate sequence of instructions, as it is with returns. Instead, they use a *trampoline* to update state and redirect control flow, where each sequence of instructions that they use as a gadget ends in an indirect jump that can be redirected to this trampoline. Their attack does not work simply with libc, but does work with Mozilla's libxul, which is distributed with Firefox and Thunderbird, and with PHP's libphp.

Another family of protections against buffer overrun attacks are *Address Space Layout Randomisation* (ASLR) techniques. Many buffer overrun exploits – particularly later, ROP techniques – rely on data or instruction sequences being in known locations in memory. A variety of techniques for achieving randomness in binary layout have been proposed in the literature[1] [80, 44, 28]. ASLR has also seen deployment across a range of OSs. Linux first deployed ASLR for user-mode processes in June 2005 [15], expanding it to cover the kernel (KASLR) in March

[1] Each with its own slightly different acronym.

2014. Windows has had support for ASLR in libraries and executables since Vista, although vendors must opt in for this at link-time [37], and KASLR since the same time [58]. MacOS introduced ramdomisation of system library locations in version 10.5, released in October 2010 [51]; randomisation of all applications in 10.7, July 2011 [1]; and the kernel in 10.8, July 2012 [63].

Deployed schemes tend to be simpler than those proposed in the literature, as they often need to maintain compatibility with existing code bases. MacOS KASLR, for example, loads the kernel at a random multiple of 2MiB from where it would ordinarily be. In general the simpler a scheme is, the easier it is to defeat. A scheme that uses a constant slide can be globally defeated if the address of a single known symbol can be found. Even if leaks of this sort are prevented, it may still be possible to defeat ASLR through the use of a timing side channel [38].

Further protections against control-flow exploits aim to mitigate the core vulnerability, where an attacker can force a branch to an location not intended by the program authors. An influential family of these techniques is those that attempt to enforce *Control-Flow Integrity* [3, 95, 94]. These attempt to ensure that the only control flow transitions that a program makes are valid. However, no current CFI scheme manages to provide both an acceptable performance overhead and a comprehensive defence against ROP attacks [22, 16, 30].

A scheme that operates along similar principles that displays very promising results is *Code-Pointer Integrity* (CPI) [46]. This guarantees the integrity of all code pointers in a program, including function pointers and saved return addresses, guaranteeing that they have not been illicitly modified. While it has low runtime overhead, it requires a program to be recompiled, making deployment on legacy codebases more complicated. It operates by placing function pointers into a 'secure region', which cannot be accessed without special checks. Follow up work discovered vulnerabilities in this work [25], but these were flaws in the implementation rather than in the fundamental mechanism of CPI.

The field of control-flow exploits and countermeasures is still in flux. At the current state of the art, control flow attacks are still feasible against the best protection deployed in conventional operating systems.

2.8 VULNERABILITY TAXONOMY

Given a description of a threat model and a corresponding perfect defence in Section 2.1, and a description of the PCIe and IOMMU specifications in Sections

2.3 and 2.6, we are now in a position to hypothesise potential vulnerabilities. We identify these as places where existing defences do not match the perfect defence, and present the taxonomy in Table 2.2.

We do not consider impersonation vulnerabilities to be fundamental, as they have been addressed in the PCIe ACS specification, as discussed in Section 2.3. In addition, they only allow data to be written, as responses to reads will be sent to the impersonated device. This allows ASLR techniques, discussed in Section 2.7, to mitigate many potential attacks.

Whether some vulnerabilities are fundamental or not is not clear cut. Batching is a choice on the part of the OS that uses the IOMMU. However, it is the default on Linux and FreeBSD, which indicates that the designers of those OSs considered that the IOMMU had an unacceptable performance cost without this optimisation. This is an example of a security-performance trade-off. Similarly, the 4KiB-granularity of mappings is a fundamental property of the IOMMU, but all attacks that leverage it can be prevented by more sophisticated use of the IOMMU, such as that presented in [55], discussed in Section 3.2, and further schemes discussed in Chapter 6. These schemes involve further security-performance trade-offs.

2.9 ATTACK AND ESCALATION TECHNIQUES

In order to turn the permission granted by a vulnerability into a workable attack, it may be necessary to follow a sequence of steps that each provide more useful information or greater permission than initially granted. The use of a ROP attack, as presented in Section 2.7, is an example of this, turning the ability to overwrite the stack into the ability to execute arbitrary code.

For an attack consisting of a number of steps, the ability to thwart any of the steps may be enough to thwart the attack as a whole. This often presents an attractive path, as it may be more efficient, in terms of implementation effort or performance overhead, to mitigate a single step. However, this can lead to a false sense of security, where an attacker can replace one technique with another that has the same effect but which is robust in the face of the new mitigation. This situation is a general case of the story of buffer overflow attacks given in Section 2.7.

Many of the attacks we present, both from preexisting and novel, draw from a collection of techniques, which we present here:

	Fundamental to	
Vulnerability	PCIe	IOMMU
1. *Trust of unverified peripherals. (Verification)*		
Full Access: PCIe grants full DMA to all peripherals plugged into the system. Allows the attacks in Section 2.10.	✓	✗
Default: IOMMU is often off by default. Allows the attacks in Section 2.10	✗	✗
Fragility: Invalid behaviour often triggers kernel panics from drivers and hosts. Used in [76], disccused in Section 2.11.	✗	✗
Boot-time: Some systems that use the IOMMU have periods at boot where it does not protect its own configuration structure. Allows the attack in [59], discussed in Section 2.11.	✗	✗
Interrupts: Early IOMMUs allowed devices to make arbitrary interrupts. Later iterations allow interrupts to be remapped and blocked. Allows the attacks in [92], discussed in Section 2.11.	✓	✗
NMIs: Interrupts that report fatal errors – *non-maskable interrupts* – are not remapped. Allows the attacks in [67], discussed in Section 2.11.	✓	✓
ATS: No mechanism to verify that a device needs ATS, and is using it correctly. Allows our novel attacks presented in Section 5.2.	✓	✓
No mechanism for verifying that a device is behaving as specified. All attacks within our threat model depend on this. Particularly relevant to our novel attacks in Sections 5.2 and 5.3.	✓	✓
2. *Access to unecessary areas of address space. (Spatial)*		
Enclave: Windows only protects a small area of memory. Discussed in Section 3.1. Allows most exploits in Section 2.10.	✗	✗
Shared Mappings: MacOS shares mappings for all peripherals, so peripherals have access to the mappings of other peripherals. Discussed further in Section 3.1. Allows our novel attacks in Section 5.1.	✗	✗
Granularity: Mappings occur at a minimum granularity of 4KiB, larger than many IO buffers. Allows our novel attacks in Sections 5.1 and 5.3.	✗	✓
3. *Data unintended for device appears during an active mapping. (Temporal)*		
Lifetime: MacOS keeps certain mappings open permanently. Discussed in Section 3.1, and allows our novel attacks in Section 5.1.	✗	✗
Batching: Many OSs, as discussed in Section 3.1, batch multiple IOTLB revocations in order to improve performance. May lead to situations where data appears in mappings that are unneeded, but not yet revoked. We do not use an exploit of this nature, as we believed that other vulnerabilities would provide more reliable exploits.	✗	✗
4. *A peripheral can impersonate another device in the system. (Impersonation)*		
PCI ID Sharing: PCI-to-PCIe bridges use the same source-ID for all peripherals behind the bridge. Used for an attack in [75], discussed in Section 2.11. We consider PCI outside our threat model.	✓	✗
Impersonation: The base PCIe specification trusts peripherals to set the correct source-ID for their own requests. We do not attempt to exploit this, as it is largely already mitigated.	✗	✗

Table 2.2: Taxonomy of Vulnerabilities.

Memory Scanning When granted access to an area of memory without knowing its semantic meaning, due to, for example, *full access* or *shared mappings* vulnerabilities, the memory must be scanned to find areas with known meanings. While this may seem to be so simple as to not count as an attack technique, an aggressive memory scan is not usual peripheral behaviour, and if it is detected and the attacker device disabled, any attack that relies on it can be prevented.

Function Pointer Overwriting Overwriting a function pointer from the attack device allows the attack device to cause any code at an address that they know to be executed. This may be the first step on the road to a ROP-style attack, as detailed in 2.7.

Configuration Rewriting Modifying the IOMMU's configuration from a peripheral allows it to be disabled, or, equivalently, for the attacker's device to open a one-to-one window into physical memory.

Symbol Leaks When coarse-grained ASLR is used to shift all addresses in a binary by a particular amount, if the shifted address of a known symbol can be found, then the shift can be derived, allowing an attack to proceed as though the ASLR is not used at all.

2.10 EARLY DMA ATTACKS

Early work on DMA attacks was largely performed over Firewire [11, 13, 29, 69, 50, 42]. Firewire, standardised as IEEE 1394, was an external peripheral interconnect that provided better performance than contemporary versions of USB. Firewire is not found on new devices, as it has an unfavourable cost-to- performance ratio when compared with Thunderbolt and USB.

Firewire attacks started appearing around 2005. They largely consisted of using a Firewire library to make an attacker's computer appear as a peripheral to a victim. A small amount of work needed to be done to ensure the attacker was granted DMA [13]. Some work involved using inconspicuous platforms for the attack: as contemporary iPods could be modified to run Linux, this was sufficient to act as an attack platform [11]. In order to make the attacks reliable, the attacker had to be sure that it did not inadvertently read from memory locations representing device registers [29], as this can trigger non-reproducible and unpredictable behaviour. The exploits generally involve scanning through physical memory for regions that exhibit bit patterns, or *signatures*, common to data structures and code regions that are known to be vulnerable. Attacks that operated by overwriting these areas gave root-shell access, leaked private keys, injected code and bypassed login screens [69, 50]. There is evidence that this type of attack has been used by state forensic and espionage agencies [42].

At the time, there was little available as an effective mitigation, short of physically destroying the Firewire port or ensuring that the computer didn't have any Firewire drivers, which was obviously an issue if you wanted to use Firewire. A deployed stop gap was to disable Firewire on locked computers, which prevented exploits from using a Firewire peripheral to bypass the login screen. The level of IOMMU support deployed in macOS since 10.8.2, which appeared in 2012, has been sufficient to thwart the most famous Firewire attack [50]. As the protection provided by Linux and FreeBSD prevents access to the kernel data structures used by these attacks, it too would prevent them.

2.11 EXISTING ATTACKS AGAINST THE IOMMU

A number of attacks have been carried out against systems with an enabled IOMMU. These vary widely in terms of scope. Some aim to violate the security promises made by hypervisors; others to violate intra-OS protection. Many of the attacks use vulnerabilities that are orthogonal to the memory remapping of the IOMMU. Current systems have been patched against many of the vulnerabilities exploited. Nobody has attempted to systematically and thoroughly probe the limitations of the IOMMU and its use in contemporary operating systems.

The first attack in the literature that mentions deliberately disabling the IOMMU appears in 2009 from Wojtczuk et al. [93], two years after VT-d was released. It is not specifically an attack against the IOMMU for its own sake, rather being concerned with attacking 'Intel Trusted Execution Technology' (TXT). TXT aims to prevent 'hypervisor attacks, BIOS, or other firmware attacks, malicious root kit installations, or other software-based attacks' [32].

The attack assumes that an attacker can execute code prior to system boot, as part of a custom or infected bootloader. It takes advantage of the mechanism used to report the presence of DRHUs to the system software. This mechanism is the DMAR ACPI table, which is configured by the motherboard manufacturer, and is placed in main memory by the BIOS. Different DRHUs remap the addresses in requests made by different peripherals. The attackers attempted to change the ACPI table to report that the DRHU that was responsible for the Intel Management Engine, which is responsible for 'out of band management' of the PC, instead covered all peripherals. Initially, their attack did not work as there is a routine responsible for verifying the DMAR ACPI table as part of the secure boot process, which is then copied into an area of tamper proof memory to be used by the system. The verification mechanism operates by reading registers from a read-only area of memory provided by the BIOS, and checking certain values. This region is mapped with a 64-bit register, but the code to read it treats it like it is a 32-bit register. This means that the attacker can force the BIOS to map the region at a different location, while the code reads from its old location. By manipulating a device BAR, the attackers can put an arbitrary memory region at the table's old address, which contains the subverted information that they want.

This is an interesting attack against some robust defences. However, it depends on the ability to run code in the bootloader, which is outside our threat model (Section 2.1). Without specific exploits that provide this, it would require root access to the system, or for the attacker to be able to perform the rewrite while running an attack tool from an operating system booted from an external disk or subverted network boot mechanism. The biggest problems with this attack are that it relies on a relatively minor bug that we expect will have already been fixed.

The next attacks appear in 2010 [75, 74]. Sang et al. posit a number of different IOMMU exploits, some of which have been exploited by other work, some of which we exploit, and one of which they exploit.

The attack they exploit relies on PCI-to-PCIe bridges. PCI, the predecessor to PCIe, is a true bus, and does not have a source-ID field. This means that every peripheral behind a PCI-to-PCIe bridge shares the same source-ID. Their threat model assumes a workplace where each colleague has an identical computer, where two workers are communicating over the network, with an attacker who is another colleague who wants to eavesdrop on this communication. The authors plug a Firewire card and NIC into the PCI slots in each computer, and find that they are behind the same bridge, so share a source-ID. The attack is carried out from a malicious iPod that is connected over Firewire. It uses an *ARP-poisoning attack*, where a fake packet that falsely binds an IP address to a MAC address on the network is inserted into the system by the iPod. They locate the receive buffers by using an address found from an identical machine. The authors also mention that it would be possible to scan memory from the iPod.

In some ways, this is not dissimilar to our proof of concept attack against macOS, detailed in Section 5.1. It has significant problems that mean that it is no longer directly applicable. Few computers now have Firewire ports at all or plug peripherals into PCI sockets. As Thunderbolt is a native part of the PCIe ecosystem, peripherals behind it do not need to share source-IDs. They attack only an unspecified version of Ubuntu Linux, and do not make specific reference to the way in which the IOMMU is used by the operating system, treating it instead as a 'black box' feature to be enabled. The attack could be relatively straightforwardly thwarted in a number of ways, most simply by variations on randomising the location of the receive buffer in memory. We cover defences of this nature more thoroughly in Section 6.8.

More work from Wojtczuk et al. appears in 2011 [92]. Unlike their previous work, this is pitched as a specific attack on the IOMMU. They attack the system by generating custom interrupts with the PCIe MSI mechanism, described in Section 2.3. While their focus is largely on systems running a hypervisor, they mention that the IOMMU can be used to harden the network stack, although they don't say how this is achieved in practice. Their attacker model has an attacker with full control over a *driver domain* – a VM with direct access to some piece of hardware that is capable of generating interrupts. They have discovered that by modifying a device's MSI capability, even a non-malicious peripheral can be made to emit arbitrary interrupts. Using a malicious peripheral would bypass this step and the requirement for a compromised VM. Even without this, they have discovered that many devices can be made to make MSI requests by using standard facilities for programmable DMA.

The precise attack is complex: it works by using an interrupt to cause the host to start running an interrupt handler with invalid arguments that cause the system to branch to an unexpected location. The attack is very much an attack against the Xen hypervisor, and is sensitive to changes in version and compiler. Crucially, VT-d supports interrupt remapping, which can block interrupts with the IDs needed to generate exceptions. The authors of the paper suggest that if the driver domain can change the MBR, it is straightforward to stop the host using interrupt remapping. However, this requires the boot disk to be mapped to the guest, which is unlikely to occur in practice. It also means that the basic attack technique can no longer be carried out solely from a malicious peripheral.

The next attacks appear in 2014, from Pék et al. [67]. They very specifically attack virtual machine monitors that have used the IOMMU to grant direct device access to guest OSs. Their work is more comprehensive in scope than previous efforts, reimplementing some of the previously discussed attacks in order to test their efficacy against then-current protections. In order to find exploits, they created an automatic fuzzing tool, PTFuzz, to generate interrupts with one of

PCIe's current mechanisms, MSI. PTFuzz is a modified version of the Intel e1000e NIC driver, and uses the same techniques as Wojtczuk et al. to generate MSIs. Using their tool they find that they are able to use a client VM to deliver a *Non Maskable Interrupt* (NMI), which represents a significant hardware fault, to the host. They were not able to use this to execute arbitrary code due to the nature of the NMI handler, but can reliably trigger kernel panics on the host, which is a significant problem due to the prevalence of cloud based hosting platforms based around VMs. Significantly, they are not able to suggest a good mitigation against these attacks, as NMIs have an important role in reporting genuine hardware faults.

Interestingly, Sang et al. were also conducting experiments in fuzzing the host at the same time [76]. They used an FPGA-based peripheral, called 'IronHide'. This is similar in many ways to our attack platform, as described in Chapter 4. It is based on Xilinx, rather than Intel, technology, and uses Xilinx's 'microblaze' processor. They implement a custom PCIe controller and bridge to allow them to fully control the transactions they produce, as they put a particular emphasis on invalid TLPs. IronHide is implemented much more as a proxy than our attack platform. Its control code is run on the PC, with the platform logging its activities over RS-232, and transferring TLP data back and forth over Ethernet. We considered this approach for our attack platform, but decided to conduct our attacks directly from the board instead. This was because we were worried about the extra latency in the request/response path implicit in using Ethernet, although we had not determined a specific limit; wished to avoid time spent developing the extra hardware integration for Ethernet; and could use the BERI processor, which has reasonable performance and can run a complete operating system.

Their work confirmed that some of PCIe's more surprising specified behaviours are implemented in practice. They were able to make PCIe transactions from IronHide with arbitrary source IDs. The IOMMU permitted write transactions as though they were from the imitated device. They also comment that they are unable to make effective read transactions, because the completion carrying the data is sent to the spoofed device. They say that this is because reads are 'nonposted', but this is a misuse of the term, which is used to describe write transactions as they do not require a completion. If a read request were nonposted, it would not require a completion at all; instead a completion is generated, but it goes to the device whose ID is spoofed by IronHide.

They use IronHide to conduct a side-to-side keylogger attack, directly reading the registers of a USB keyboard from IO space to log the data sent by the keyboard. They can then re-inject this sequence of keystrokes to force a login.

Their description of their attack is frustratingly short on detail. In particular, they do not say how they know the address of the USB controller in IO space. They go on to claim that the IOMMU cannot protect against this attack. On a strictly literal level, this is true. The IOMMU does not protect against accesses to IO space. However the host does not need to map devices into IO space at all. It can just interact with them in memory space, which can be protected by the IOMMU. The IO space can only be enabled by the host, as the root complex is the only component permitted to make configuration requests; we have verified this from hardware.

They then implement a fuzzing tool which generates packets in Python for the board to send. They find that many invalid TLPs are ignored by the host while some cause a kernel panic; and that some valid TLPs cause the host to keep running, while some cause the driver to invoke a kernel panic. These allow DOS attacks, but only against an OS, rather than a hypervisor, and require a malicious peripheral, rather than a malicious OS installation operating a conventional peripheral. Particularly, as their attack requires physical access, it is unlikely that a regular human cannot conduct the same DOS attack by pressing and holding the off button.

This paper has many of the limitations of the previous work by the same authors. It does not go into significant detail about the precise workings of their attacks, or of the operating systems used. It treats the IOMMU as a black box to be enabled, rather than a hardware component that is used by the OS in a particular way. This treatment of the IOMMU causes them to miss many attacks that we pick up on. While fuzzing can theoretically eventually carry out any exploit given enough time, it would be extraordinarily unlikely for it to be able to fool the operating system into attempting to send network data over the attack platform, for example, a cornerstone of several of our attacks.[2]

The latest work in the field in appears in 2016, from Morgan et al. [59, 60]. They exploit a period of time at boot where the tables used to initialise the IOMMU are not yet protected by the IOMMU. They use an unnamed Xilinx-based attack platform, based around a 'LatticeMicro32' processor. Their attack works with the knowledge that the DMAR tables (See Section 2.6) are always allocated in the same place across each system boot. As the system boots, they flood it with write requests to add an entry to the DMAR tables that will cause the IOMMU to open a 1:1 mapping for all of system memory for their attack platform. This is a neat attack, but straightforwardly thwarted. On systems where there is sufficient

[2] We independently discovered that invalid TLPS and TLPS that are valid but carry unexpected data can cause a kernel panic by generating several of them over the course of debugging...

control over the firmware, the IOMMU can be enabled immediately after boot, which now happens on macOS. On systems where this isn't possible, KASLR is sufficient to thwart the attack.

We summarise existing DMA attacks in Table 2.3. While much interesting work has been carried out to attack systems despite the presence an IOMMU, there are still significant gaps in the field. There has been a lack of attention to the way that the IOMMU is actually used by the operating system. The only work to point out the limitations of the IOMMU's 4KiB granularity is that of Markuze et al., which we discuss in Section 3.2 [55]. However, this work is largely about improving the performance of the IOMMU, and they do not attempt to demonstrate any vulnerabilities. Throughout the attacks presented here, there has been the assumption that once the IOMMU is turned on, no attacks against its protection are possible. This is implicit in the forms the attacks take: using interrupts, attacking the IO space, or changing the IOMMU configuration before it is enabled. Consequently, each attack that is more sophisticated than a denial-of-service attack can be straightforwardly prevented.

			Attack Vector								Vulnerability Utilised						Techniques Employed		Attack Achieved			
			Software	Firewire	Thunderbolt	PCI	PCIe	Within Threat Model	Fundamental	FPGA-based Platform	*Full Access*	*Fragility*	*Boot-time*	*Interrupts*	*NMIs*	*PCI ID Sharing*	*Memory Scanning*	*Configuration Rewriting*	Control Flow Hijacking	Eavesdropping	DOS	Disables IOMMU
Early Attacks				✓				✓	✗		✓						✓		✓	✓	✓	N/A
Wojtczuk et al.	2009	[93]	✓					✗	✗					*				✓				
Sang et al.	2010	[75]				✓		✗	✗							✓				✓		
Wojtczuk et al.	2011	[92]	✓					✓	✗					✓								
Pék et al.	2014	[67]	✓					✓	✓					✓	✓						✓	
Sang et al.	2014	[76]					✓	✓	✗	✓		✓		✓						✓	✓	
Morgan et al.	2016	[59]					✓	✓	✗	✓			✓					✓				✓

Table 2.3: Summarising of published DMA attacks. Descriptions of the vulnerabilities are presented in Section 2.8.

* This attack is outside the threat model, and uses a different vulnerability.

KNOWN MITIGATIONS 3

There is a widespread belief among the authors of DMA attacks that the use of an IOMMU is sufficient to prevent their attacks. However, an IOMMU is simply a piece of hardware, and is useless without proper configuration from the operating system. The extent to which deployed OSS are capable of using the IOMMU to prevent attacks from malicious peripherals, and how they are configured by default is not well documented. One of our contributions is an analysis of the protection offered by four major operating systems. We also examine the work done in the research space to improve upon deployed protection. Hypervisors, also called Virtual Machine Monitors (VMMS), make use of the IOMMU to assign hardware devices to VMS, and to stop hardware devices assigned to one VM from violating the security of another VM. We consider hypervisors to be outside the scope of this thesis, so do not cover them in detail.

3.1 DEPLOYED PROTECTION

Many, but not all, hardware platforms in use today provide an IOMMU.

In terms of software, we looked at Windows, macOS, Linux and FreeBSD. Between them, these OSS are the default on almost every PC today. We found that there is a large variation between the systems in terms of the protection they offer. Perhaps shockingly, macOS is the only system that turns on IOMMU protection by default. Only the latest enterprise version of Windows provides any sort of protection whatsoever, and the process to turn it on is very convoluted. FreeBSD and Linux both offer IOMMU-based protection, but both have extremely scant documentation on how to enable it. In the variants that we tested, FreeBSD 10.3, Ubuntu 14.04 and 16.04 and Red Hat Enterprise Linux 7.1, all of which are fairly recent[1], the IOMMU was switched off by default.

[1] FreeBSD 10.3 and Ubuntu 16.04 were released in April 2016; Ubuntu 14.04 in April 2014; Red Hat Enterprise Linux 7.1 in March 2015.

Hardware Support

VT-d first appeared in Intel's product line in 2008 as part of the Core 2 Duo E8000 series, but its appearance in specific models since then has been complicated and irregular. The cheapest model to include VT-d was the Core 2 Duo E8300, which cost $163 upon release, placing it as a low-to-middle-range chip.

VT-d was not present in the 'Bloomfield' first wave of i7 branded cores, which appeared in late 2008 directly after the E8000 series, but was present in all but one of the following 'Lynnfield' models. Releases vary in support for VT-d thereafter until 'Sandy Bridge' in 2011, from which point onwards all i7 branded models have it, with the exception of models with the suffix 'K'. The 'K' models are designed for home-enthuasiast overclockers, and VT-d is generally pitched as a business extension. After 'Broadwell' in 2016, these models support VT-d as well.

The situation is similar for i5 models, which have generally included VT-d since 'Clarkdale' in 2010, with the exception of some of the lower end models. Since 'Ivy Bridge' in 2012, all i5 models have supported VT-d. VT-d did not appear in the low-end i3 and Pentium series until late 2015, with the 'Skylake' series. Since then, it has been present in all models. It first appeared in the server and workstation Xeon range at the start of 2009 and has been present in all Xeon processors since then.

All models in the AMD FX processor line, which are currently sold at prices from $70, have AMD-V and IOMMU support [6]. These were introducted in late 2011. AMD's current high-end processor line, 'Ryzen', also has full IOMMU support. It seems likely that their processor-and-graphics SOCs, which they call 'APUs', also have support for the IOMMU. Although this is not stated unequivocally on the product webpages, it probably falls under the banner of 'AMD Virtualization'.

Intel SGX

Intel *Software Guard Extensions* is a security technology added to Intel processors with the Skylake microarchitecture in 2015 [20]. It aims to solve the problem of *secure remote computation*, where a user wishes to use a computer owned by an untrusted party to carry out a computation on their data while being assured of the integrity and confidentiality of that data. SGX requires that the user trusts the processor executing their code, but does not require them to trust any other component of the remote system including any software running at a higher privilege level than their code, or any peripherals attached to the remote system.

To provide these guarantees, SGX uses *enclaves*, which are containers for security-sensitive computation. Security for enclaves is guaranteed with an attestation mechanism that allows a remote user to authenticate the software running in an enclave, and an isolation mechanism that shields the enclave from the untrusted components outside it. Isolation is provided with the use of *Processor Reserved Memory*, which cannot be directly accessed by any software not in an appropriate enclave. This is enforced by the use of a memory encryption engine, and by the modification of the CPU's integrated memory controllers to drop any attempted DMA access. In order to ensure that the OS or VMM does not have to be in the TCB, this mechanism is separate to the IOMMU.

Running any part of an application in an enclave therefore shields it from any DMA attack that depends on accessing any code or data that is in the enclave. This is far from a universal solution to DMA attacks. It requires a specific application to be written with a proprietary technology, one that a specific company holds licensing restrictions on. There is not an obvious way to employ SGX in order to make a whole system secure against DMA attacks, although some of the techniques it employs, like encrypted memory and areas of memory that devices are forbidden to access may be useful.

Microsoft Windows

The level of intra-OS protection by Windows is not sufficient to protect against a wide variety of possible DMA attacks, despite the existence of a significant number of Windows machines that have a Thunderbolt 3 port. We found that the only version of Windows currently on the market that supports the use of the IOMMU for protection is Windows 10 Enterprise.

Windows IOMMU support is part of a suite of tools called *Virtualization Based Security* (VBS). This runs the primary 'root' OS inside a Hyper-V virtual machine with a second minikernel running in a container alongside. The minikernel provides two different feature sets. The first is *Device Guard*, which is designed to prevent a device from executing applications that aren't explicitly marked as trusted in security policy – the 'device' in the name refers to the computer being 'guarded'. The other is *Credential Guard*, which is designed to prevent credentials like Kerberos tickets from being stolen.

At present, Microsoft make some questionable and confusing claims about the level of protection against DMA attacks offered by VBS. One official web page states that 'In Windows 10, an IOMMU can be used to enhance system resiliency against memory attacks.' Further, that this 'Prevents advanced memory attacks' [64]. These statements are arguably true, but far from comprehensive. The page indicates that the IOMMU is only necessary for Credential Guard, not Device Guard. However, another page states that 'Virtualization-based Security (VBS) using IOMMUs' is a Device Guard feature, and that 'With this type of VBS protection, when the DMA-based attack makes a memory request, input/output memory management units (IOMMUs) will evaluate the request and deny access.' [48]

We have confirmed experimentally that only the security minikernel is protected from DMA. This is somewhat useful as it prevents a malicious install of the root operating system from being able to use a peripheral's scatter gather abilities to carry out a confused deputy attack, for example. But there are still a large number of extremely damaging attacks that could be carried out without this, however, as peripherals can read and write all of the memory assigned to the root OS.

On top of this, enabling VBS is difficult. When we attempted it, it took a man-day of work by a very capable computer user. It required a process that involved running Powershell scripts and rebooting the machine multiple times. As Windows only uses the IOMMU as part of VBS, there are many unrelated prerequisites that can prevent it being turned on. For example, it requires *Secure Boot* to be enabled, which in turn requires UEFI boot, which requires a disk with a 'GUID Partition Table', rather than an MBR style one. This means that disks that have been upgraded from Windows 7 would not be able to use VBS.

MacOS

MacOS was the only operating system we tested that enabled the IOMMU by default. IOMMU based protection was introduced to macOS in version 10.8.2, which was released in September 2012.

In macOS, PCIe drivers are given an object-oriented interface to create buffers that are exposed to peripherals. All devices share the same hierarchical translation tables, so windows are opened for all devices on the system simultaneously.

On macOS, the IOMMU windows for network data are opened at boot and persist for the lifetime of the system. This approach has a number of attractive features. In particular, it avoids the IOTLB shootdown operation necessary to revoke a per-buffer mapping, which is a significant source of overhead [9]. A lifetime mapping added directly to network subsystem also reduces the burden on driver authors, and may help with backwards compatibility with old drivers.

Linux

Linux has support for the use of the IOMMU for peripheral protection, with support for VT-d since 2006, and AMD-V since 2007. We have verified that VT-d can be enabled and provides protection over the course of several experiments with real machines. We have not been able to test AMD's implementation due to a lack of hardware, but have verified that the appropriate code exists. We tested Ubuntu versions 14.04 and 16.04, and Red Hat Enterprise Linux 7.1, which is Common Criteria EAL4+ certified, and found that none enabled the IOMMU by default.

VT-d can be enabled with a kernel boot parameter, `intel_iommu`. This accepts a comma separated list of parameters. The value '`on`' is sufficient to enable protection, with other values corresponding to features that may or may not be present in different versions of VT-d. Confusingly, Linux also has the boot parameter `iommu`: setting this to '`on`' is *not* sufficient to use VT-d for peripheral protection, but setting it to '`off`' will lead to it being turned off. Numerous forum posts attest that this has confused users.

Drivers open windows in memory through use of Linux's DMA API. This was originally written for peripheral interconnects that had a more restricted address space than that of the system. Its API specifies functions that take a pointer to a buffer as input, and ensures that data allocated in the buffers will be exposed to a given peripheral, returning a host-opaque handle to the data that can be passed to the peripheral. It also specifies free functions so that buffers can be marked as no longer needing to be device-accessible. Initially, this API was implemented by bounce buffering. The IOMMU back end allocates an IOVA and opens a window, without copying the mapped memory anywhere. The IOVA is the opaque handle returned to the function caller. By default, the kernel will batch revocations in order to reduce the performance overhead that comes from flushing the IOTLB. To enable synchronous revocation, the keyword '`strict`' can be added to the `intel_iommu` boot parameter. Linux has separate mappings for each peripheral, unlike macOS.

Linux attempts to allocate large mappings where possible, although it will not compromise on 4KiB window granularity to do this. For example, a 2044KiB window will be allocated as 511 4KiB mappings, a 2048KiB window as one 2MiB mapping, and a 2052KiB window as one 2MiB mapping and one 4KiB mapping. Note that this assumes that all of the windows are appropriately aligned. However, our experience suggests that most mappings are made on a per-buffer level, so super-page mappings are likely to be rare. For example, the Intel e1000e NIC driver never makes an allocation larger than 24KiB, so will never allocate a superpage. VT-d platforms predating Haswell do not support superpages at all. No window smaller than 4KiB can be opened, due to the structure of the page table.

The kernel documentation says that guard pages are added after each buffer. An investigation of the source does not support this, and it is unclear what additional security would be provided by guard pages.

FreeBSD

FreeBSD implements a very similar level of IOMMU support to Linux, although it only supports Intel VT-d, not AMD-V.

We investigated FreeBSD 10.3, and found that it does not enable the IOMMU by default. The loader tunable '`hw.dmar.enable=1`' must be set for it to be enabled. This tunable is not well-documented – information is provided on some mailing list posts, and in the FreeBSD source tree. The IOMMU is tied to FreeBSD's DMA API, `busdma`, in the same way as in Linux. It attempts to allocate large pages in the same way. On 64-bit machines, FreeBSD is always strict in its buffer unmapping: on 32-bit machines, it is always lazy. According the source, this is because of limitations of the `busdma` API on 32-bit machines.

In one machine we tested, attempting to turn on FreeBSD's IOMMU protection actually prevented the machine from booting. This problem was not experienced with Linux, and was due to the way that the BIOS reported devices to the OS. The BIOS was not reporting all of the devices that needed IOMMU windows: FreeBSD used the reported values, while Linux was more generous. This led to FreeBSD refusing to open the IOMMU windows necessary to load the rest of the OS from the hard drive. Problems of this nature are obviously a serious impediment to the widespread use of IOMMU protection by default.

3.2 RESEARCH

There has been concern about the performance penalty of enabling the IOMMU since 2007, which predates the introduction of VT-d. In an early piece of work on IOMMU performance, Ben-Yehuda et al. [12] found that, when running netperf and communicating with a 1Gbps NIC, no difference in throughput was seen with the IOMMU turned on. However, it did result in a processor overhead of 60% on a powerful system for the time. Their results indicated that there was nothing intrinsic about the IOMMU that reduced performance, with the overhead mainly found to reside in calls to map and unmap memory, rather than in the actual translation of the IOVAs.

Several schemes were proposed to alleviate this overhead:

1. *Don't call map and unmap.* A mapping of all or most of physical memory is made at boot time, and only unmapped at shutdown. This only makes sense in situations where the IOMMU is used for virtualisation. It provides no isolation.
2. *Allocate buffers in advance; free when done.* Re-use DMA buffers for as long as possible. This involves a large change to the Linux drivers, as drivers are generally written with the expectation that DMA mappings are a scarce resource, and should be mapped and unmapped as close to their use as possible. It was suggested instead that the DMA API IOMMU back end should cache and reuse mappings.
3. *Batched allocate and free.* This is similar to the previous scheme, but relies on making use of specific calls to map and unmap multiple buffers at once. As a suggestion, it is specific to the Linux DMA API. Again, this does not follow the conventional wisdom on how to write Linux drivers.
4. *Never Free.* This would still require keeping track of which entries were no longer used, in order to reallocate the mappings as necessary. It also reduces the isolation benefits provided by the IOMMU as the lifetime of the system progresses. It is essentially the same as having an infinitely long timeout on batched revocation.

2008 work from Tomonori highlighted that Linux's red-black-tree-based algorithm for allocating IOVAs had poor performance for common workloads when compared with a bitmap based solution [84]. It highlighted that batching IOTLB revocations led to enhanced performance, at a cost to safety. It also revisited the idea of opening a window consisting of all of host memory for a device, while acknowledging that this negates the beneficial isolation properties of IOMMU use. That this suggestion is thought of at all seems to highlight that the primary role of the IOMMU is virtualisation, rather isolation.

Also in 2008, Willmann et al. carried out an evaluation of several strategies for the use of IOMMU [90]. They were primarily motivated by support for virtualisation, and this means that both their performance and security results are not necessarily relevant to the use of the IOMMU for intra-OS protection. Their solutions require that each guest ask the Virtual Machine Manager (VMM) for IOMMU mappings, rather than granting them itself. These extra steps make it somewhat complicated to determine the relation to their measured performance with use directly by the OS. It also assumes a threat model where a malicious guest OS uses a DMA enabled peripheral to access the memory of other guests. This means that their definition of isolation ends up being different to ours, where we rely on protection against actively malicious peripherals.

Their strategies were:

1 *Per-buffer mapping.* Mapping and unmapping each buffer as close to the point of use as possible, as is implemented in Linux and FreeBSD now.

2 *Shared mappings.* Per-buffer mapping creates a new mapping for each buffer, regardless of whether that buffer shares a page with a pre-existing mapping. This scheme just involves re-using a mapping where one for a buffer on a page exists, which obviously reduces mapping and unmapping overhead. This is not how we use the term shared mappings in the rest of this thesis, where it refers to macOS' scheme of opening identical windows for each device.

3 *Persistent, re-usable mappings.* This is an extension of strategy 2. Rather than unmapping after use, the OS periodically collects un-used mappings.

4 *Per-guest static mappings.* Each guest has a large window opened into host memory, which exists for the lifetime of the guest.

5 *A software solution.* This was a best-existing software solution to act as a baseline. It was taken from [81]. It involved the VMM verifying that requests from the guest operating systems for DMA regions are valid, and issuing requests to the peripheral itself. With the assumptions made in their threat model, this provides isolation, although it obviously cannot prevent a malicious peripheral from reading other locations.

They found, as expected, a number of trade-offs between intra-OS protection and performance. Of the schemes they implemented, per-buffer mapping offers the most protection, but at the greatest performance cost. Static mappings offer no protection, but with performance almost indistinguishable from not using the IOMMU at all. They claim that enough IO operations happen concurrently that sharing them reduces overhead without making any security compromises. They further claim that allowing mappings to persist and be re-used allows an overhead very close to that of static mappings, whilst still providing security close to per-buffer mappings.

At the time their research was performed, they did not have access to a full x86 IOMMU. Instead, they used a GART on an AMD system, which has a somewhat similar feature set, but does not provide protection across all of memory. The GART is faster to update the page table and flush the IOTLB compared to a conventional IOMMU, meaning that their performance results underestimate the costs of frequent mappings and unmapping. They tested a range of benchmarks across a pair of 1Gbps NICs. This means their results are likely to be unrepresentative of machines with 10Gbps and faster NICs.

The paper does not carry out a comprehensive security analysis: none of the schemes they evaluate provide better than 4KIB granularity, so they potentially leak data surrounding the mapped areas. It is interesting to note that only the per-buffer strategy seems to have made it into deployment.

In 2010, Amit et al [9], identified that the limited size and high-flush-cost of the IOTLB carried a performance overhead, although this was negligible compared to the cost of the map and unmap calls in per-buffer based protection schemes. They suggested a number of guidelines and techniques to ensure maximum efficiency from the IOTLB.

We were then unable to find any work in the area until 2015. Almost eight years on from the initial exploratory work performed in the area, Malka et al. analysed the performance of the protection deployed in Linux and FreeBSD [53]. The bulk of their paper is given to analysis of Linux. They found a significant slow down due to a pathological case in the Linux IOVA allocator when using devices with multiple ring buffers. Due to the descriptor ring structure and Linux's per-buffer IOMMU strategy, they found that it was sufficient to maintain a cache of recently used IOVAs 'in front' of the main data structure that could then be quickly re-issued, as many IOVA requests were for memory areas of the same size, and a fairly low, constant amount were needed by the system at a time.

They evaluated their new scheme on two pairs of servers, with one pair connected by 40Gbps NICs, and the other by 10Gbps NICs. They evaluated a suite of benchmarks: Netperf TCP stream, Netperf UDP round response, ApacheBench, and Memcached. Their results showed a large variation across workload; we summarise them in Table 3.1. The authors provide two factors that explain why they see a better relative performance improvment for a 40Gbps NIC than for a 10Gbps NIC. Firstly, the driver of the 40Gbps NIC uses more ring buffers, and allocates more IOVAs than that of the 10Gbps NIC. In particular the 40Gbps driver makes two IOMMU allocations per packet, while that of the 10Gbps NIC makes only one. This means that IOVA allocations take a larger proportion of the life of the system. Secondly, with the 10Gbps NIC, their server is capable of achieving line rate with the non-optimised deferred scheme, so no improvement is possible. Their optimisations provide a modest yet useful improvement, and suggest that the effects of their enhancement will be more pronounced as – or if – faster and faster NICs become the norm. Their paper does not give the throughput achieved

	Strict	Deferred
10Gbps	1.38	1.03
40Gbps	2.60	1.25

Table 3.1: Geometric mean average performance improvement provided by Malka et al.'s improvements over a baseline. A score of '2' means their improved allocator achieves twice the throughput of the baseline. [53]

by the benchmarks with protection disabled completely, which is a significant oversight.

Their analysis of FreeBSD reveals that it does not have the same pathological case, but does have a number of cases where it is evident that the IOMMU protection mechanism has not been written with performance as its top priority.

Further work on IOMMU performance from Malka et al. [52] suggests entirely restructuring the IOMMU. This is based around the observation that high performance peripherals make use of ring buffers. Rather than having a unified page table, they propose a single page table per ring in the system. Each table is a flat array of physical addresses, and each IOVA is an index into the array. The IOTLB has at most one entry per ring. For network benchmarks, it performs better than a standard IOMMU, and typically has performance within 25% of no IOMMU in terms of both throughput and CPU usage, although this may still be high enough to prevent widespread adoption. At the same time, it can provide more granular protection than the standard IOMMU design.

However, this work is purely research, and there is little evidence that it will be deployed on current hardware. As there are several usage modes for which their 'rIOMMU' (Ring IOMMU) structure is unsuitable, they suggest it as a complementary device that could be included in the system alongside a conventional IOMMU. They do not go into detail about how it could be deployed. It seems more plausible that 'rIOMMU' would be deployed more as a different algorithm for walking the page table than a discrete device, with one bit of IOVA specifying whether a conventional or page-based lookup should take place. The per-device look-up phase would remain the same, with an option of a ring translation structure instead of a hierarchical one. It is also not clear if the use of the ring IOMMU is compatible with the current DMA API, or whether it necessitates drivers being changed in order to make the ring-based structure explicit.

More work was done on IOMMU performance by Peleg et al. [68] in 2015. They look specifically at performance of the transient per-buffer scheme on multi-core systems with NICs providing bandwidths in excess of 10Gbps. They observe that as IOVAs must be globally consistent, the somewhat naïve IOVA allocator is poorly optimised when multiple cores simultaneously are attempting to allocate IOVAs. The poor performance arises due to Linux's centralised and lock-protected IOVA allocator, so they propose three more efficient designs:

Identity Mapping A buffer's physical memory address is used as its IOVA. This has several problems in practice. Buffers in the same page will share a mapping, so that mapping now needs a reference count. That reference count can overflow, in which case they have to fall back on the default IOVA allocator. It means that extra bits have to be used to deconflate mappings with different permissions on the same page. It is less secure than alternative schemes, as not knowing the physical address of the mapped page can complicate attacks, as explained in more detail in Chapter 5. It is not suitable for buffers that need to be given contiguous IOVAs that are not contiguous in physical memory. It leads to larger page tables – with a dedicated allocator, IOVAs can be densely allocated, which is not the case when using physical addresses.

Kmalloc Kmalloc is the kernel's fast, multi-core scalable allocator. Rather than using a new instance of the same allocation algorithms, the authors allocate kernel memory and use the physical addresses of these buffers for IOVAs. For efficiency, rather than allocating byte-for-byte, they allocate a byte for each page of IO data required. While this leads to fairly modest overhead, it litters kernel memory with inaccessible and wasted bytes in order to lower implementation cost.

Per-core Caching When a IOVA is unmapped, a per-core cache instead stores it as available for use by another buffer managed by that core. When an IOVA is requested, the core first checks to see if it can be allocated by the cache. This is complicated as cores can have producer-consumer situations, where one core repeatedly deallocates buffers allocated by another core. They solve this problem by using magazines, a solution for multicore memory allocation first published in 2002 [14].

The other scalability bottleneck they find is IOTLB invalidation. Linux uses a centralised locked queue of mappings to be invalidated, which again causes contention. They solve this via the use of per-core queues, which are then invalidated in batches.

The paper uses a benchmark of 270 netperf instances on the client, and one instance per tested core on the server running a request-response workload on a 16-core x86 system with a 10Gbps NIC. Their designs achieve 93%-95% of the performance achieved with the IOMMU disabled in this scenario. The metric they use is transactions per second: bandwidth is not relevant, as streaming benchmarks can saturate network bandwidth without their enhancement using a single core at 22% CPU usage. It is not clear to me how representative this is likely to be of real systems, but as their scheme results in improved performance without significant loss of security, it is useful.

As one of their optimisations is to batch invalidations, their system is not suitable for the case when strict mappings are used. It does not enable better security to be attained.

2016 saw work by Markuze et al., that involved fundamentally changing the manner in which the IOMMU was used to provide better protection and performance [55]. This is a contrast to a lot of the previous work in the field, which, since the protection was initially implemented, has been largely engineering improvements to the same algorithm.

They identify page granularity and delayed unmappings as potential security vulnerabilities, although without actually exploiting them. They operate by allocating windows of memory that are visible to each device for the lifetime of the system. To expose a buffer to a device, instead of opening a window, they copy its data to a *shadow buffer* in one of these preallocated windows. This flies in face of the received wisdom for implementing network stacks – that copying is to be minimised – due to the changed overheads resulting from the IOMMU.

For buffers larger than 4KiB, they copy only the unaligned sections of the mapped buffers into shadow buffers, eliminating the need to copy very large sections of data. The copying happens only on map and unmap, which is consistent with the semantics of the DMA API. The authors have explored some optimisations to the copying process, using a fast `memcpy` implementation, and allowing the driver writer can signal that it is not necessary to copy all of a buffer to a shadow.

They evaluate their solution on a high-performance server equipped with a 40Gbps NIC. They examine only the Netperf TCP stream benchmark, for which they achieve performance that is equivalent to the best existing work while providing a better level of protection. They do see a slightly elevated level of CPU usage, and do not examine the effect of their scheme on other applications on the system. As their copying approach will be extremely polluting to the cache, this may well be severe. Their performance analysis was carried out on Linux, which is generally regarded to have a less efficient network stack than FreeBSD [47].

3.3 SUMMARY

The use of the IOMMU for intra-OS protection, summarised in Table 3.2, is extremely patchy. Every deployed implementation appears to have security vulnerabilities. It is frequently extremely difficult to enable protection, and in the majority of PCs in use at the moment, it is impossible without installing a different operating system.

Perhaps because IOMMU hardware has a reputation as intrinsically causing performance overhead, until recently, deployed protection mechanisms have often had inefficient software implementations. The research engineering that revealed this and suggested solutions to some of the performance issues is summarised in Table 3.3. The results of this work was included in the Linux kernel as of version 4.7, which was released in July 2016. However, there is no evidence that similar work has been done for FreeBSD.

There has not been a recent, concerted cross-platform effort to evaluate the best way to use the IOMMU for protection across a range of workloads. As a consequence, almost every deployed system is vulnerable to DMA attacks, as we demonstrate in Chapter 5.

		Protects Against Vulnerability Class										
	Present Across Range	*Default*	*Boot-time*	*Interrupts*	*NMIs*	*ATS*	*Enclave*	*Shared Mappings*	*Granularity*	*Lifetime*	*Batching*	*Impersonation*
Windows 10	✗	✗	?	?	?	?	✗	✗	✗	✗	✗	✓
MacOS 10.12	✓	✓	✓	?	?	✓ *	✓	✗	✗	✗	✗	✓
Linux 4.7	✓	✗	?	?	?	✗	✓	✓	✗	✓	◐ †	✓
FreeBSD 11.0	✓	✗	?	?	?	✓ *	✓	✓	✗	✓	✗	?

✓ Full Protection. ? Untested. ◐ Partial Protection. ✗ No Protection.

* These OSS don't support ATS at all, so it does not work as an attack vector.
† Batching is on by default, but can be turned off with a kernel boot argument.

Table 3.2: Summary of protection deployed across recent versions of operating systems. 'Present across range' means that IOMMU support is included regardless of the version of the OS that the user purchases and installs. Vulnerability classes are described in Section 2.8.

			Vulnerabilities Addressed					
			Verification	Spatial	Temporal	Impersonation	Reduce Overhead	Novel Hardware
Amit et al.	2010	[9]	✗	✗	✗	✗	✓	✗
Malka et al.	2015	[53]	✗	✗	✗	✗	✓	✗
Malka et al.	2015	[52]	✗	✓	✓	✗	✓	✓
Peleg et al.	2015	[68]	✗	✗	✗	✗	✓	✗
Markuze et al.	2016	[55]	✗	✓	✓	✗	✗	✗

Table 3.3: Summary of IOMMU-based mitigation research. The vulnerability taxonomy can be found in Section 2.8. We do not include [12] and [84], as these exist primarily to evaluate the performance of the schemes that existed at the time of the research. We also do not include [90], as this concerns the use of the IOMMU for inter-OS protection, which we consider to be outside the scope of this thesis.

4 THE RESEARCH PLATFORM

In order to carry out a real-world adversarial analysis of DMA attacks, and thus prove or disprove the thesis of this dissertation, we had to build a platform that could emulate the full range of PCIe transactions and operate over Thunderbolt.

The platform that we have created is the first to run a complete OS directly on the attack platform. It has a robust and user-friendly API. We aim to release the platform as open-source, which will enable other users to conduct research into PCIe behaviour, including more investigations into stability and security. Combined with a PCIe analyser, the platform is suitable for experiments that measure latency in a system. Further modifications would allow it to saturate the PCIe link, for experiments concerning system throughput.

We include photographs of the attack platform in context with two of the victim systems we used for our attacks in Figures 4.1 and 4.2.

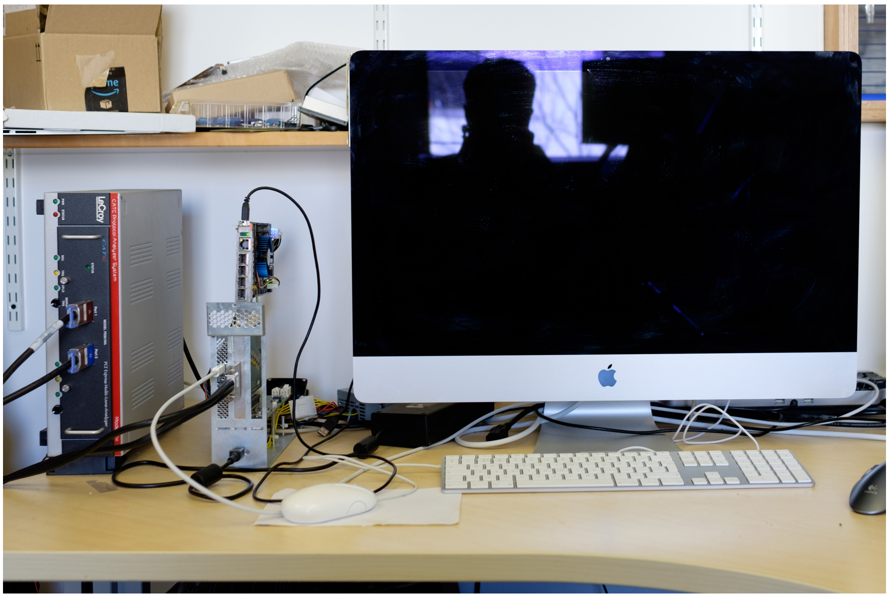


Figure 4.1: Photograph of macOS experimental setup. From left to right: the PCIe protocol analyser; the PCIe-Thunderbolt bridge, with the FPGA that runs the attack attached through a riser that connects to the analyser; the victim machine.

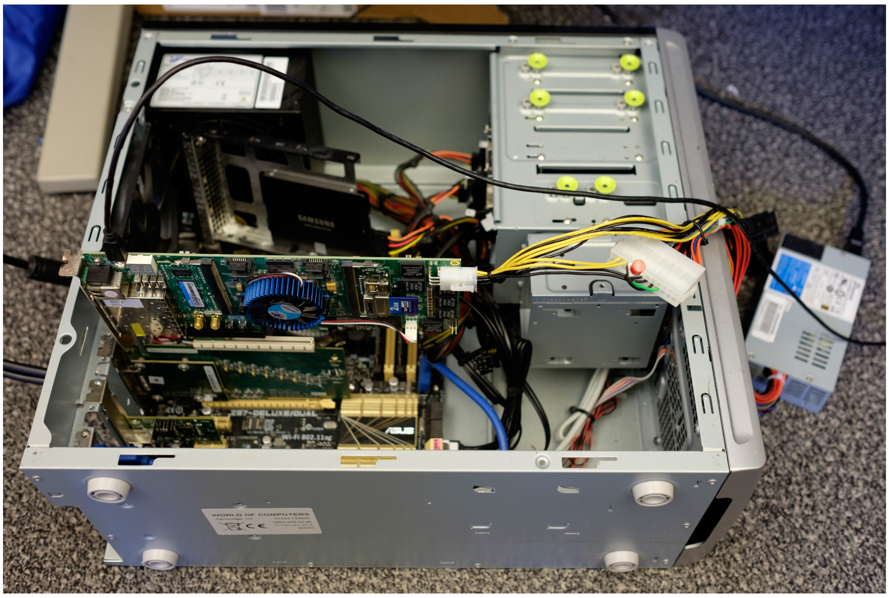


Figure 4.2: Photograph of Windows, Linux and FreeBSD experimental setup. This includes the FPGA in the riser for the PCIe analyser. The OS was varied on the victim by replacing the hard drive.

4.1 PLATFORM STRUCTURE

The platform is divided into front- and back-ends. The attack code is part of the front-end, and generates PCIe traffic that is delivered by the back-end. All code in the system is written in C. The attack platform as a whole is shown as a block diagram in Figure 4.3.

Front-end

Attack Code The top level of the system is code written by the user. This can poll to receive PCIe TLPs, and create and dispatch new PCIe TLPs of its own. The attacks we developed are detailed in Chapter 5.

NIC Model The attack code may optionally make use of a full behavioural model of an Intel 82754L NIC. To the host, this appears to transmit Ethernet packets in response to requests, and can respond to DHCP requests to assign the host an IP address. To create this behaviour, we extracted the NIC model from the QEMU emulator, and replaced the functions that perform simulated DMA with ones that perform actual DMA. This allows more sophisticated attacks to be performed. The user can build their attack around the NIC model by using hooks that allow callbacks to be run, for example, before a packet is

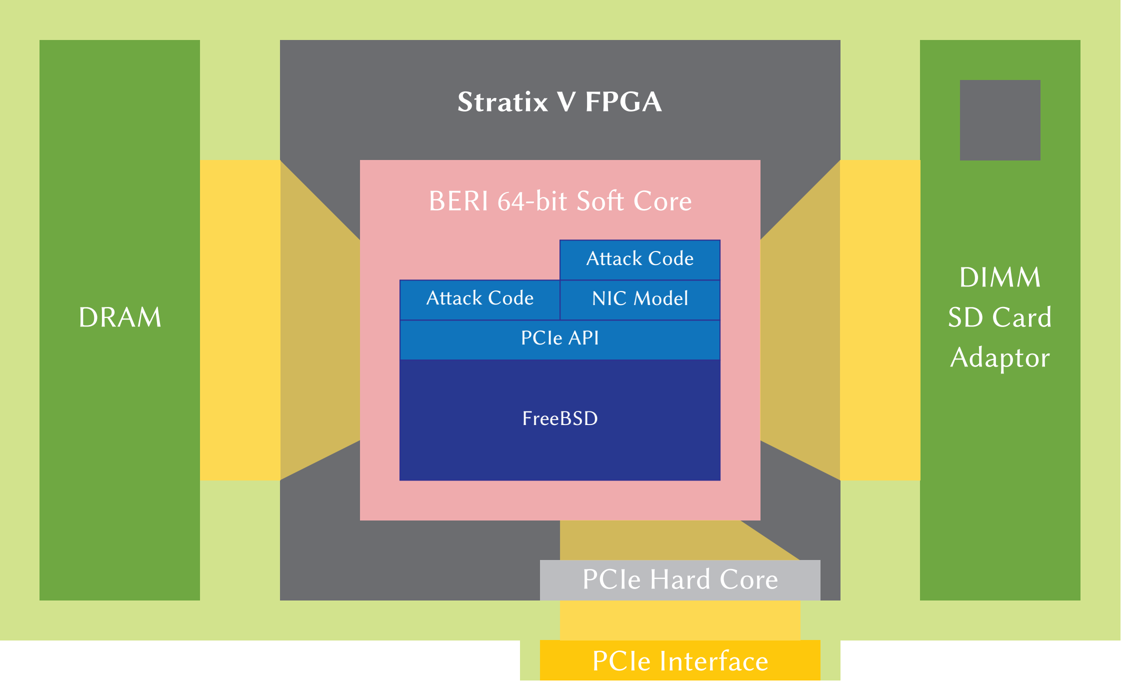


Figure 4.3: Block Diagram of the Attack Platform on the DE5-Net Board. 'Attack Code' is included twice, as different attacks can be carried out with and without the NIC model.

transmitted. This NIC model is detailed in Section 4.4.

PCIe Library Our library allows the creation, parsing, dispatch and reception of TLPs with a user-friendly interface. It also allows high-level blocking DMA transactions, which better match the standard idioms of the C programming language. The library and its operation are described in Section 4.3.

Back-end

FPGA The standard back-end is a Terasic DE5-Net FPGA Development Kit. This has a PCIe interface that allows it to be plugged into a system like any other PCIe peripheral. The Intel FPGA on the DE5 board includes a PCIe *hard core*, an area of fixed-function silicon that handles the low-level details of the PCIe protocol. We describe the FPGA back-end in Section 4.2.

Postgres Database This is an alternative back-end to the FPGA which helped with the development of the NIC model. In order to ensure that the model was responding in a functionally identical manner to a real NIC, we recorded the PCIe traffic between a real host and NIC over the course of a system boot, and replayed this into the NIC model, verifying that its responses matched those of the real NIC. This is described in Section 4.5.

FreeBSD Operating System The front-ends made use of the functionality provided by FreeBSD. On the FPGA, this was provided as part of the BERI system, as described in Section 4.2. With the Postgres database, it was simply the OS of the host.

Thunderbolt Bridge Some systems we attacked by plugging the FPGA directly into the PCIe slots on the motherboard. This was not possible for the Macs we used, so we plugged the FPGA into a 'MaxExpansion.com™' Thunderbolt-to-PCIe bridge.

4.2 THE TERASIC DE5-NET AND SOFT SOCS

The main system back-end is a Terasic DE5-Net, an FPGA development kit primarily aimed at network applications. An FPGA is a device that can be programmed to behave like different digital circuits. A compiled description of a circuit that is ready to be programmed to an FPGA is known as an *image*. We used an FPGA-based platform because we had the DE5 boards readily available, and they include a programmable PCIe interface. Due to their flexibility, programming FPGAs is timing consuming and complicated, and involves long compilation cycles. To ease development, we used soft-core processors, which are programmed onto the FPGA fabric, rather than built directly in silicon. In contrast to using a custom hardware design for each attack, this allows a program to be written in a commodity programming language and run on the board.

The DE5 board connects some of the pins on the FPGA to a PCIe interface, and provides a *hard core* that handles the low level details of the PCIe protocol. It hosts an Intel Stratix V series FPGA, has two SODIMM sockets, and a PCIe connector. The FPGA is primarily composed of a collection of *adaptive logic modules* (ALMs). These consist of an eight-bit-input look-up table, registers to enable sequential logic, and two embedded adders, to improve the performance of this commonly-used operation. These are connected by a programmable interconnect. A circuit can be emulated by programming the ALMs to act as small collections of gates, with the interconnect taking the place of wires in a real circuit. The trade off for this reprogrammability is that a circuit implemented on an FPGA will require a larger area and operate at a lower clock frequency than an equivalent implemented directly on silicon. On the other hand, the ability to debug an in-progress design is invaluable, as is the possibility of creating a one-off device economically, which is not possible on silicon due to the high once-per-run costs. In order to narrow the performance gap between ICs and FPGA, the Stratix V series of FPGAs have a series of *hard blocks*, also known as *hard IP* or *hard cores*, which implement commonly used functionality directly on silicon. These include multiplier blocks, transceivers and PCIe hard cores. It is increasingly common for FPGAs to include processor cores implemented in this way. These typically achieve performance of the order of a factor of ten better than a soft core[1]. Developing for the FPGA directly is time consuming and involves long test and compile cycles. Over the course of the project, we developed two separate *soft SoCs* as attack platforms. These consisted of a processor, a software interface for the PCIe core, and a mechanism for getting code onto processor. Creating a soft SoC for the DE5 platform involves using the 'Qsys System Integration Tool' to add components and specify the connections between them.

Nios II

The first platform was built around a Nios II soft core. This core is provided as part of the Intel FPGA tool chain, along with tools for cross compilation from Linux. It was designed by Altera before they were bought by Intel, so bears little relation to Intel's silicon processor offerings. It implements a 32-bit instruction set, has reasonably good performance, and occupies a very small area of the FPGA. Software can be downloaded to the Nios II over the same link used to program the design onto the FPGA without recompiling the whole of the system.

This platform provided a useful basis for introductory work. However, it became clear that we wanted to build attacks that involved reusing code that required an operating system to run. Despite there being some information online about running Linux on a Nios II based system, we were not able to implement this with two days of exploratory work. Instead, we decided to build a platform around the BERI processor.

BERI

BERI implements a 64-bit MIPS instruction set [89, 87] and is capable of booting a full FreeBSD operating system. This allowed us to develop attacks in C using a

[1] An Intel Arria V FPGA has two ARM Cortex-A9 cores running at 1.05GHz: these are partially out-of-order superscalar cores. The Stratix IV FPGA, which is from a higher-end product line, but from a generation before, runs an in-order BERI core at 100MHZ.

full suite of pre-existing libraries, allowing re-use of existing code and relatively fast compile times. As it is developed at the University of Cambridge, we had considerable in-house expertise to draw on. It is written in Bluespec System Verilog (BSV), a hardware description language with a much higher level of abstraction than 'vanilla' Verilog. Using BSV results in a much reduced development time, at a potential cost in performance and resource usage. The cost in resource usage is for a variety of reasons. BSV uses standard interfaces for module boundaries, which may not always have optimal performance. It makes it easy to include complex data structures, which may be unnecessary for a given task. However, our access to FPGAs capable of instantiating complex designs meant that these costs were a reasonable trade-off. BERI is a higher performance core than Nios, with several features designed to increase the *instructions per clock* (IPC) that it executes, while still running at 100 MHz. The use of BSV, and BERI's performance enhancing features mean that it uses a significantly larger amount of FPGA resource than Nios. However, the Stratix V is a high-end FPGA, so can comfortably fit a BERI-based SOC.

4.3 THE SOFTWARE INTERFACE

To manipulate the Intel PCIe hard core from C, a relevant part of the file `/dev/mem`, which provides an interface to physical memory, is mapped into the address space of the attack program through the use of the `mmap` system call.

We built a C-language library to allow this to be used to interpret and create PCIe packets. Getting this working was one of the most technically challenging parts of the project. This was for a variety of reasons: firstly, PCIe is a complicated protocol, and the Intel interface was not always intuitive. Secondly, the compile and test cycle was time consuming and involved, despite our best attempts to simplify it. Thirdly, it is difficult to get useful debugging information out of the system. PCIe is very timing-dependent, meaning that printing information to the screen can drastically affect the behaviour of the protocol, and it operates at a low level with many parts that are opaque.

Writing the interface software was only possible because we had a PCIe analyser. This could read all the PCIe traffic passing between one PCIe expansion card and the host, which could then be downloaded onto a PC for inspection. This was extremely helpful in demonstrating when the hard core was not sending the packets we were expecting. However, traces are hard to analyse, because they contain many thousands of packets. The analyser can also not indicate what endianness of data the host is expecting.

The library as described is largely that for BERI. There were some differences with the Nios, as it is a 32-bit processor, so operations that require only one memory access on BERI required two on Nios. More differences were caused by endianness differences. The Nios is little-endian, while BERI is big-endian – the interactions between a big-endian processor and a little-endian host were the cause of many small bugs over the development of the project.

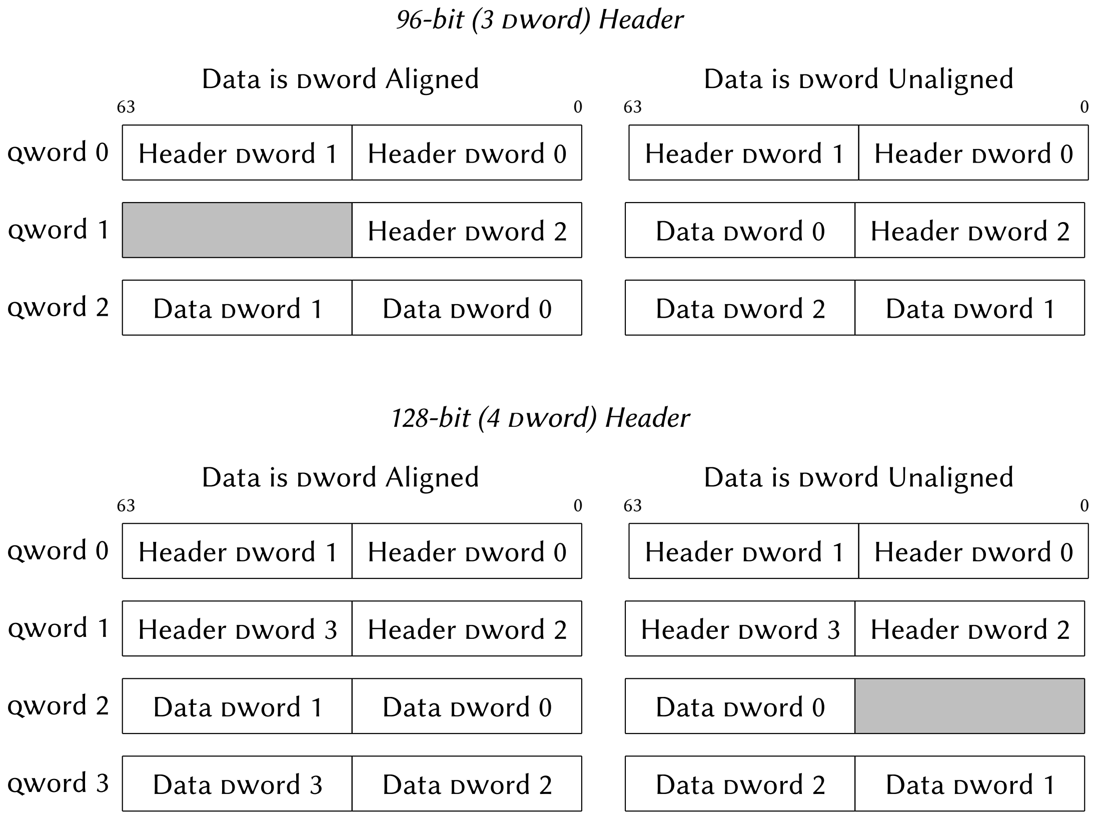
The PCIe Hard Core

The data provided by the Intel PCIe Hard Core is a representation of the Transaction Layer Packets (TLPs) that are the highest level encapsulation of data supported by PCIe. Ideally, the software library used to produce and send, and receive and interpret TLPs would have an interface entirely agnostic to that of the hardware component used to actually send and receive them. However, we found that paying attention to the core's interface simplified creating the library, and potentially allows for higher performance.

Intel's PCIe core can be configured to present an interface with a width of either 64, 128 or 256 bits. We used the 64 bit interface, as this allowed the BERI CPU to read or write a complete frame of data with a single instruction.

The core can be configured to handle configuration requests from the host by itself, or to report them over its interface as another variety of request TLP, allowing custom responses to be made. The former simplifies the development process, and means that responses to configuration requests are generated faster; the latter allows more flexibility.

The relationship between a TLP's header and data required by the hard core has a strange quirk. The fundamental unit of data in a TLP is the DWord, which is 32 bits (four bytes) of data. Every TLP has a header that is either three or four DWords in length. Conceptually, a TLP has the bits of its data immediately following the bits of its header. This is not how the hard core represents its data. It forces the data carried by the packet to have the same alignment within the packet as it is reported to have within memory. This means that if a packet has a three DWord header, and one DWord of data aligned on a 64 bit boundary, it would require three frames – reads of the interface register – to be read by the software. The first frame would have the first two DWords of the TLP's header. The second frame would have the third DWord of the TLP's header. The third frame would



96-bit (3 DWord) Header

Data is DWord Aligned

	63	0
QWord 0	Header DWord 1	Header DWord 0
QWord 1	▨	Header DWord 2
QWord 2	Data DWord 1	Data DWord 0

Data is DWord Unaligned

	63	0
QWord 0	Header DWord 1	Header DWord 0
QWord 1	Data DWord 0	Header DWord 2
QWord 2	Data DWord 2	Data DWord 1

128-bit (4 DWord) Header

Data is DWord Aligned

	63	0
QWord 0	Header DWord 1	Header DWord 0
QWord 1	Header DWord 3	Header DWord 2
QWord 2	Data DWord 1	Data DWord 0
QWord 3	Data DWord 3	Data DWord 2

Data is DWord Unaligned

	63	0
QWord 0	Header DWord 1	Header DWord 0
QWord 1	Header DWord 3	Header DWord 2
QWord 2	Data DWord 0	▨
QWord 3	Data DWord 2	Data DWord 1

Figure 4.4: The various alignment options that complicate using the Intel PCIe Hard Core from software. The QWords are sent by software sequentially in order. Notice that the alignment of the Data DWord is the same as its alignment within memory. The alignment can be derived from the 'Lower Address' field in a TLP Completion. This means that the '`send_tlp`' function has to interpret the semantics of the TLP it is sending.

have the DWord of the TLP's data. This various possible combinations that can occur as a result of this are illustrated in Figure 4.4.

This particularly complicated the design and implementation of the software interface. At first, we did not realise that this behaviour was specified. Debugging it was extremely difficult. The problem manifested as packets being sent later than they should have been, and with a malformed data payload. This was because the PCIe core would interpret part of the header of the next packet as data for the previous packet.

It complicated the design of the upper layers of software because it meant that to get the best performance, buffers would have to be allocated in different places depending on the reported alignment of the data. In order to avoid having to do this, the function to send TLPs, `send_tlp`, checks the data and alignment it is expected to send, and copies the data into a memory location with the correct

```
/*
 * Length fields are in bytes.
 */
struct RawTLP {
        int header_length;
        TLPDoubleWord *header;
        int data_length;
        TLPDoubleWord *data;
};
```

Figure 4.5: RawTLP definition

alignment before sending. This constitutes a layering violation. Ideally `send_tlp` would simply take an area of memory representing the TLP as its argument, and copy this to the PCIe core. Instead, it takes the header and data sections separately, and then has to inspect a TLP-dependent location in the header field to determine how to send the data.

We believe this behaviour exists for cases where the PCIe core is used by a piece of specialised hardware with a memory that has an interface the same width as the PCIe core's. In this case, having the data alignment like this would remove the need for a potentially large multiplexer to shuffle the data alignment.

RawTLP

The lowest level of library was built around a data structure called `RawTLP`. The definition of this is given in Figure 4.5. This is principally used by the functions `wait_for_tlp` and `send_tlp`. The former takes as its arguments a pointer to a buffer, the length of the buffer and a pointer to a `RawTLP`. When a TLP arrives from the host, it copies the data into the buffer and sets the fields of the `RawTLP` argument to point to the relevant areas of the buffer.

Ideally this function would be be able to treat the TLP it receives from the PCIe core as an opaque collection of bits, but this is not due to possible due to the alignment issues mentioned above. Instead, the `wait_for_tlp` function has to interpret the address field of the TLP in order to set the `data` pointer correctly. Doing the calculation at this layer means that it is not necessary for upper levels to calculate the data offsets themselves. If the upper levels were required to calculate data offsets, it would potentially require the same calculation to be performed in many different locations in the codebase. Doing the calculation at a lower level also means that, were the library to be ported to use a different PCIe core that

<table>
<tr><th colspan="8">+0</th><th colspan="8">+1</th><th colspan="8">+2</th><th colspan="8">+3</th></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">Requester ID</td><td colspan="8">Tag</td><td colspan="4">Last
DWord BE</td><td colspan="4">First
DWord BE</td></tr>
</table>

```
struct TLP64RequestDWord1 {
        uint32_t requester_id:16;
        uint32_t tag:8;
        uint32_t lastbe:4;
        uint32_t firstbe:4;
};
```

Figure 4.6: The definition of the second DWord for all request transactions, as given in the PCIe specification. The bytes are presented in ascending order from left to write, with most significant bit of each byte on the left. The BE fields are *byte enables*. This is followed by the C-language definition of the `struct` used to represent this DWord of header.

didn't have this behaviour, the same upper layer could be used.

The `send_tlp` function takes a `RawTLP` as its only argument. This function may fail to work if it is given a `RawTLP` with header and data pointers that are not QWord aligned. While perhaps unintuitive, it was not difficult to ensure this condition by allocating packet buffers as QWord arrays and casting the pointers. It has a special case for packing together the header and data where necessary. We also perform endianness correction at this point, by endian-swapping each DWord of data.

Constructing TLPs

In order to interpret the semantics of the data arriving from the PCIe core, we wrote a series of struct definitions, with fields derived from the PCIe specification. We defined a struct for each possible form of each possible DWord in each position of a TLP's header, which are named according to their content and position relative to the start of the header. To use these structs, a reinterpret cast is performed on the header pointer within a RawTLP. For example, Figure 4.6 shows the definition of the struct for the second dword of a request transaction.

For convenience, we provide some functions to construct some of the more frequently used headers with a single call.

Higher Level Functions

While reading and writing TLP structs manually suffices for some of the simpler interactions carried out by PCIe, the operations necessary to carry out arbitrary DMA from C are more complicated. To simplify this, we provide the functions `perform_dma_read` and `perform_dma_write`. The former takes a pointer to a buffer, a potentially-unaligned address, and a length in bytes. It converts the length in bytes to a length in DWords, as required by PCIe, and calculates the byte-enables necessary to handle the unaligned address. The byte enable calculations generate a mask that varies in length based on the disparity between the length of the request and the length rounded to the next greatest DWord. PCIe specifies that it is possible for a DMA read to return its response across multiple completions, so the `perform_dma_read` function loops, consuming responses until the amount of data it has received is the same it has requested. This means the function blocks, and the caller cannot do anything whilst waiting for the responses to come back. The read function also has to deal with unrelated packets that can arrive on the PCIe interface at any time. These may be to do with power management, for example. The simplest solution is just to drop packets like this while waiting for an appropriate response, despite running the risk of dropping requests from the host that required a response. In practice, we found this was the most convenient model to work with.

The corresponding write function is somewhat similar. We found that attempting to send writes as one massive TLP caused the system to drop the write. Instead, we fragment the write into units of no larger than 128 bytes. PCIe writes are *non-posted*, which means that the write function does not need to wait for a response before returning. This eliminates some of the complexity associated with reads.

4.4 QEMU

Attacks built from scratch with the PCIe library on a Nios-based platform were sufficient to carry out attacks against less sophisticated platforms. However, some operating systems will not open any windows into memory for peripherals that they do not recognise. In order to subvert these platforms, we needed to more closely emulate a real PCIe NIC. It seemed expedient to modify an existing implementation, rather than build one from scratch. We found a model of the 82574L produced by third-party developers for the QEMU full-system emulator to fulfil this need. The model refers to itself as the 'e1000e', which is the name of the driver that supports the 82574L NIC, along with many other Intel gigabit ethernet controllers.

We also considered using a model from the 'uvNIC' project, which simulates a NIC as a userspace thread in order to facilitate driver development without a hardware implementation of the target device [33]. We decided not to use the uvNIC model for several reasons:

1 Its software stack is more complicated than QEMU's. Its NIC model runs in an entirely separate OS thread to the running application, which would increase attack latency.

2 It does not have a facility for producing packets without a real network interface.

3 No model for a commodity 1gbps NIC was available. uvNIC implements a model of a 1gbps NetFPGA NIC, which is considerably less widely-used than the Intel 82754L. It otherwise implements 10gbps NIC models, which are more complex without improving the power of exploit available to us.

4 It was not open source.

Cross compiling QEMU

Our first attempt to use QEMU's 82574L NIC model involved compiling QEMU for Nios, replacing the use of unavailable operating systems services with trivial stubs. To build the NIC model, we wrote a generic makefile that attempted to compile all the C files in a directory structure to form a program. We started by building up from a file that performed a basic instantiation of the peripheral. To automate the process of enlarging the number of used files, we wrote a Python program that interpreted compiler error messages and searched for missing files to include, offering the option to stub out functions rather than trying to pull in the definition.

Perhaps unsurprisingly for the reader, this didn't work. QEMU is too large, complicated, and intertwined for stubbing out functions and compiling to bare metal to be a viable compilation strategy. On top of this, it makes heavy use of OS features, including some threading operations. This lead to the development of the BERI SOC, in order to have a platform that natively supported QEMU's dependencies.

The BERI SOC is too slow to use as a development platform, so QEMU had to be cross compiled from an x86 host. QEMU has a complicated makefile, and we could not easily modify it to use the toolchain to build for BERI. Instead, we used the Python tool to iteratively copy in dependencies from the main QEMU source tree that could then be built with a simple generic makefile. This also contributed to a faster build cycle, as the subset needed to run the peripheral is significantly smaller than QEMU as a whole: 666 vs 2913 C and header files, representing 328,013 vs 1,193,459 lines of code, including whitespace and comments.

QEMU Modifications

In order to provide a uniform interface for device creation, QEMU defines an *object model*, with features common to most instances of the object-oriented paradigm. It also has clearly-defined APIs for interacting with device and host memory.

We took an iterative approach to converting QEMU. Rather than trying to convert everything in one monolithic chunk, we aimed to make the model steadily more correct. We had three tools for checking correctness:

Postgres Model To allow us to verify that the platform carried out the same behaviour as a real NIC, we could play back a sequence of TLPs from a Postgres database. This is described in detail in Section 4.5.

PCIe analyser The PCIe analyser we used was able to detect some errors within TLPs, like packets that included a length field that didn't match the actual length of that data carried by the packet. The analyser could not detect incorrect behaviour that was device specific, but allowed problems to be debugged that would have only displayed the symptom 'it doesn't work' from the host PC.

Modified FreeBSD driver The FreeBSD e1000e driver included a large number of debug print statements that were implemented using a macro that produced a no-op in its released form. Replacing the body of the macro with calls to the print function provided immediate and useful feedback about unexpected behaviour from the card. It meant that only the TLPs that were actually causing problems needed to be checked from the analyser, rather than requiring a laborious cross reference between the trace and the NIC specification.

A number of initialisation functions are required to be called for enough of QEMU to work for the peripheral to function. We discovered these by trial and error. Bugs caused by a failure to call these usually manifested as some variety of access to invalid memory, as an uninitialised pointer was deferenced. The Postgres model was particularly useful finding these bugs, as it meant that we could use sophisticated debugging tools like Valgrind and GDB which were not available on BERI.

QEMU splits network emulation into two parts. The NIC, which provides the interface, and the *client*, which is the back end that processes the packet data provided to the NIC. In order for its network emulation to work, the QEMU main loop has to be running. However, at the same time the platform has to be calling `wait_for_packet` in order to be able to respond to requests from the host.

There are several plausible solutions to this problem. The most obvious is multi-threading. However, BERI is not truly multi-threaded or multi-core, so there would be no performance advantages to operating in this way, and writing multi-threaded code is notoriously difficult. Instead, we made use of coroutines, which are functions which can be resumed after yielding control, in contrast to a normal function which has to be restarted after a return statement. Coroutines are natively supported by QEMU. Using them has given the program as a whole a slightly odd structure. It consists of a main loop that schedules the packet processing coroutine to run immediately, then calls the function representing a single iteration of QEMU's main loop. The packet processing coroutine will consume packets as long as there are any available, polling the receive queue a thousand times before allowing the QEMU main loop to execute. This is not particularly elegant, and the use of the value of one thousand for retries is based on trial-and-error in order to find a value that works.

The packet response loop is structured as a select-case statement based on the type of received TLP. Due to QEMU's device model, reacting to configuration requests is not particularly complicated. First the function ID of the incoming request is checked to ensure that the request is bound for the emulated NIC. Then, the address of the host's access to configuration space is reconstructed from register fields within with TLP. This is used as the argument to QEMU's standard functions for interacting with its model of the PCI configuration space. A standard completion header is created, and the appropriate response is sent.

Memory and IO requests are somewhat more complicated. QEMU's system for modelling address spaces is very powerful, but correspondingly complex and relatively slow. It represents the address space as an arbitrary tree of data structures called *memory regions*, which represent an area of memory with a base and bounds. They can do this directly, or they can have a number of child subregions. They can also reference a device model that is in charge of handling a particular area of address space. Memory regions can overlap, with a priority order between regions to determine which actually handles the request.

In order for QEMU to find the memory region associated with a particular address, a tree data structure called a *trie* is created. Each level of the trie can be thought of as a table that contains either pointers to the next level of tables, or data entries. To find a data entry, successive levels of the trie are indexed with successive bits of a search key. This is in contrast to a simple binary search tree, which has a data entry for a complete key at each node. Many applications that use tries do so with a branching factor larger than two, in order to reduce the number of successive look-ups that need to be performed at a cost in wasted space where the data structure is sparse. The multi-level page tables used by modern MMUs are tries. QEMU rebuilds the trie mapping between addresses and memory regions each time a memory region changes its address. Unfortunately, this happens fairly frequently early on the boot sequence. This is because the system writes to each of the registers that specify each device's address spaces, its *Base Address Registers* (BARS), twice in order to find out the amount of address space required by each device. This is covered in more detail in Section 2.3. These register writes cause their associated QEMU memory region data structures to update their addresses. Due to the structure of QEMU and the attack program, this expensive trie reconstruction took sufficiently long that the system viewed the packet as having being dropped. This meant that when the response was finally generated, the host thought it was a response to a different request, and the whole system eventually collapsed.

Diagnosing this bug was difficult, as while the analyser showed a request that did not have an associated reply, there was not an obvious way of quickly working out why this was. Fixing the problem was further complicated as there isn't a profiler that works on the BERI platform. We diagnosed the bug by intuiting that a delayed response was a possible cause, then by recording the trace generated by the attack platform when run in the system, and replaying this from Postgres. Profiling the x86 system revealed that the overhead was coming from the trie maintenance, which was similarly expensive across the platforms.

Rather than attempting to improve the performance of this operation, we worked around it. The only memory regions that were used by the NIC were bound directly to the device's BARS. Instead of using the trie to find the memory region corresponding to a given request, we simply iterate through the BARS until we find one with a base and bounds that correspond to the address in the request. This involves accessing fields that are private according to the object model, but is much faster than the official method. We simply removed the references to the code that causes the trie to be updated, as it is no longer necessary.

Enabling the platform to perform DMA to the host is fairly straightforward, due to QEMU's use of a sensible API. Most of the NIC's access to host memory is done via the functions `pci_dma_read` and `pci_dma_write`. We re-implemented these to use the real PCIe core. While few unexpected bugs arose, this did involve complexity in joining systems with different interfaces and different expected semantics together. This was handled in the high level functions we wrote for the PCIe library, as described in Section 4.3. Rather than blocking until data returned on a read, it may have been possible to perform useful work while waiting. It would also have been possible to respond to unrelated packets that occured while blocking, but this would have required a mechanism to pause the read, respond to the other packet, and resume the read. In practice, the simpler approach we took with the library worked. Part of this is, we suspect, due to the robust redundancy measures used by PCIe.

Some memory access was performed through the function `cpu_physical_memory_map`, which brings some amount of emulated host memory into the QEMU process address space. Reimplementing this to use real host memory was simplified as its semantics do not require it to track updates. This meant that it could be replaced with a call to `malloc` to allocate a buffer, and a DMA read of the relevant area of host memory. The `cpu_physical_memory_unmap` call then just freed the buffer, to avoid memory leaks.

Interrupts were again fairly straightforward to implement. The most modern PCIe interrupt mechanism, MSI-X, is implemented as PCIe DMA memory writes to specially configured areas of memory. Enabling interrupts required that the MSI-X BAR was properly set up, and that QEMU's `msix_notify` function used a real DMA write.

QEMU Network Backend

In order for the simulated peripheral to appear as a real NIC to the victim, it has to appear to process packets. It is possible for this to amount to simply advancing the register pointing to the head of the received or transmit ring, as covered in Section 2.5, without actually processing or interpreting the data contained therein. However, processing the data enables more sophisticated exploits, at the cost of requiring a network stack. Thankfully, QEMU provides several options for this. The simplest, and the one that we use in practice, is SLIRP. Originally designed to allow emulation of PPP internet connections from a shell account, it allows QEMU to provide network access for a guest operating system while being run as a userspace process on the host.

In the context of the attack platform, SLIRP operates as follows:

1 An application on the host begins sending a packet using the fake NIC.

2 The host's network stack wraps the data to be sent in the appropriate header information.

3 The host sends PCIe DMA memory-write requests carrying the data and headers it has created to the fake NIC.

4 The main loop on the fake NIC dequeues the DMA write data from the host, and calls the appropriate functions to cause it to be written into the memory regions of the QEMU model of the fake NIC.

5 This transfers the written data and headers into the SLIRP network stack.

6 SLIRP strips the headers from the packet, and makes a decision about how to process the packet based on this header information. If the data is a UDP packet, it uses the attack platform's FreeBSD network stack to transmit a UDP packet with the same data to the same destination as the original packet. If it is a packet that forms part of a TCP handshake, it opens a TCP socket to the same destination as is being attempted by the host, again using the attack platform's network stack. If the data is part of an ongoing TCP connection, it uses the socket that it will already have opened to transmit this to destination.

7 If the host is carrying out a TCP handshake, the next iterations of the QEMU main loop will carry out the correct DMA into host memory to represent this.

8 If data arrives at the attack platform over a connection opened by SLIRP, it is wrapped up in the appropriate headers, and the appropriate DMA requests are made from the attack platform to deliver it to the host. In practice, this does not happen, because the attack platform is not connected to a network.

SLIRP emulates a DHCP server as part of its network stack. This means that the DHCP requests are never forwarded over a socket, and SLIRP instead handles address allocation itself. This is particularly useful because it means that some packets are sent and received by the virtual NIC, without the necessity of it being plugged into a real network.

There is nothing apart from engineering work stopping us using the real network connections on the DE-5 board to implement the sockets used by SLIRP. This would allow the platform to function as a real NIC, albeit an extremely slow one.

The other main network back end provided by QEMU is the TAP back end, which uses a special network device to inject Ethernet frames directly into the host network stack. For the majority of cases, this provides much better performance than SLIRP. For us, it is less useful, as it does not support a built-in DHCP server, and would require connecting the artificial NIC to a real network.

4.5 POSTGRES BACK END

In order to enable more rapid debugging with improved state-visibility, we created a software back end.

We used our PCIe analyser to record the boot sequence of a host with a real board using a similar NIC to the Intel 82754L. The analyser interface allowed us to export this trace to CSV, which we then imported into a Postgres database using a Python script that simply did the appropriate data conversion on the fields of the csv file, and executed the queries to insert them into the database. We wrote a back end for the PCIe library that used the Postgres database as its packet source. Instead of delivering TLPs from a physical system, the `wait_for_tlp` function delivered TLPs from the Postgres database in the order they were received by the analyser from the host. The `send_tlp` function extracted the TLPs that the real NIC would have sent at a particular point in the execution cycle from the Postgres database, and checked that each was semantically equivalent to the packet that the model was trying to send.

The data from the analyser was not an exact binary representation of the data seen by hardware. The analyser outputs a semantic model of each packet, so a separate value was given for each field within the TLP. In the CSV file output by the analyser, this is all in a string format. The Python script that put a trace into the Postgres database was configured with the type of each field in the database, and converted from the string input as appropriate. To facilitate both sending and receiving TLPs, the python script placed transactions into different tables based on whether they came from the NIC or from the host.

The Postgres C library supports a stateful interface that allows the database to be queried, and results to be extracted one row at a time. This allowed a single received query and a single transmit query to be used for the lifetime of the system. In order to extract the value from a given field of the query, the `PQgetvalue` function is used, which takes a `PGresult`, representing the result of a query, a row number and column number. As we used Postgres' 'single row mode' to return a single row at a time, we always extracted from row 0. In order to make the interface somewhat self-documenting, and more robust in the face of reordering of the columns, we used the `PQfnumber` function to return the column number for a given field name string each time. This is less efficient than using a fixed column ID, but the performance of the software model is adequate. The `PQgetvalue` function returns a void pointer to a binary representation of the value in the database in host endianness. To get the value itself, the pointer must be cast to the

```
/* ... */
#define    POSTGRES_INT_FIELD(FIELD_NAME)                          \
  static inline uint32_t                                           \
  get_postgres_##FIELD_NAME(const PGresult *result)                \
  {                                                                \
    int field_num = PQfnumber(result, #FIELD_NAME);                \
    return bswap32(*(uint32_t *)PQgetvalue(result, 0, field_num)); \
  }
/* ... */
POSTGRES_INT_FIELD(length);
/* ... */
```

Figure 4.7: Macro to simplify access to data in a Postgres record from C. The code here causes a function called `get_postgres_length` to be defined.

appropriate datatype and dereferenced. Then, in order to match the endianness output by the BERI platform, it must be endianness swapped. As this process has to be performed for each field in the row, we used a macro to simplify creation of these functions, as shown in Figure 4.7. A similar macro is used for 64-bit fields. Several fields are better thought of as enums. These are stored as strings in the postgres database, and converted to C-language enums by functions that are analogous to those that do the appropriate conversion for integer fields.

The semantics of each position in a TLP depend on the type of the TLP, and not all TLPs have the same fields. In order to represent this data as a table, the analyser has a column for every possible field, with only a subset used for each row. The `tlp_from_postgres` function takes a query result, a buffer in which to construct the TLP, and pointer to a `struct RawTLP`, as described in Section 4.3. It constructs pointers for each possible semantic meaning of each location in the output header, using the same types described in Section 4.3 for constructed TLPs to be sent from the attack code. This is shown in Figure 4.8. To fill in the relevant fields, the `tlp_from_postgres` function performs a switch-case on the type of the tlp, an enum extracted from the database, and uses the generated functions to populate the fields.

The use of the same structs that are used to construct and parse TLPs from attack code to create them from the Postgres database is somewhat self-referential. In particular, it led to one bug where the bit pattern we reconstructed from the semantic data in Postgres was in the wrong endianness to that actually emitted by the PCIe hard core. This was not spotted until we ran the model against a real system. Additionally, the NIC we traced was not identical to that emulated by QEMU,

```
struct TLP64DWord0 *header0 = (struct TLP64DWord0 *)buffer;
struct TLP64MessageRequestDWord1 *message_req =
        (struct TLP64MessageRequestDWord1 *)(header0 + 1);
struct TLP64RequestDWord1 *header_req =
        (struct TLP64RequestDWord1 *)(header0 + 1);
struct TLP64CompletionDWord1 *compl_dword1 =
        (struct TLP64CompletionDWord1 *)(header0 + 1);
TLPDoubleWord *dword2 = (((TLPDoubleWord *)buffer) + 2);
struct TLP64ConfigRequestDWord2 *config_dword2 =
        (struct TLP64ConfigRequestDWord2 *)(dword2);
struct TLP64CompletionDWord2 *compl_dword2 =
        (struct TLP64CompletionDWord2 *)(dword2);
TLPDoubleWord *dword3 = dword2 + 1;
TLPDoubleWord *dword4 = dword3 + 1;
```

Figure 4.8: Being able to use each position in a header with different semantics. The `TLPDoubleWord *` are used for cases when the whole field is used for one purpose, like data or an address.

so there were some small differences that we had to ignore. The `wait_for_tlp` implementation provided by the Postgres back end could be configured to mask certain positions of certain TLPs in order to faciliate this.

Nevertheless, using Postgres greatly accelerated the build cycle. To test a binary in a real system required it to be built, copied to an SD card, plugged into the board, mounted in the board, copied into `/dev/null` to ensure that the binary was cached by the OS to enable reasonable performance, run, and for a trace from the analyser to be read by eye for mistakes. Using the Postgres back end immediately highlighted packets displaying unexpected behaviour.

4.6 SUMMARY

We created a soft-SoC including a BERI processor and a PCIe hard core, which allowed the raw byte data of PCIe TLPs to be sent and received over a PCIe interface. The PCIe hard core was mapped into the physical address space of the soft-SoC. We wrote a software library on top of this that allows TLPs to be received from the interface and interpreted, and to be generated and sent by it. This enabled a variety of attacks against platforms which do not provide different levels of access to host memory dependent on PCIe device and vendor IDs. However, some platforms only open windows into host memory based on a device's expected behaviour.

In order to carry out attacks against these platforms, we extracted a model of an Intel 82574L NIC from the QEMU full system emulator, cross compiled it to run on the BERI processor, and modified it so that previously-simulated DMA is actually performed for real on a target machine. This allowed more sophisticated attacks, as described in the next chapter.

NOVEL ATTACKS 5

In Chapters 2 and 3, we demonstrated that the state of the field when it comes to IOMMU use is confused at best. Attacks employing DMA assert that they can be stopped by the presence of an IOMMU, or will be in the near future. However, we have found that the IOMMU is largely disabled, and even when it is turned on, the algorithms using it appear to have vulnerabilities.

The most significant research contribution of this thesis is the first comprehensive analysis of DMA attacks against a variety of systems with the highest level of IOMMU support enabled. We have found novel vulnerabilities in recent versions of Microsoft Windows, Apple macOS, Linux and FreeBSD. To mitigate many of these would require significant engineering work, or entail performance degradation.

We attempted to find vulnerabilities of two kinds for each OS: those that allow private data to be read, and those that allow control flow to be subverted. We give brief notes on why we do not focus on data injection attacks below.

There are many different orders in which the attacks in this section could be presented. The attacks can be categorised according to severity of exploit permitted, type of vulnerability exploited, and operating system targeted, and ordered accordingly. We present them in order of ascending sophistication of attack platform used to carry out each attack. This reflects the order in which they were developed.

Windows has very little support for the use of the IOMMU for protection. In macOS and FreeBSD, we exploited the same key vulnerability, where a function pointer is exposed to a NIC as part of network packet metadata. We exploited macOS 10.11.5 without emulating a real NIC by taking advantage of its shared mappings. To attack FreeBSD, we configured the platform to act as a NIC. This means it is explicitly presented with the information it needs to carry out the exploit, and does not need to scan memory.

We exploited Linux by taking advantage of PCIe's ATS features, detailed in Section 2.3. These allow a device to request that the IOMMU is bypassed for its requests in order to remove translation overhead. Even though it negates any security benefit of the IOMMU, use of this feature is granted by Linux to any device that presents a PCIe capability indicating that it is capable of using it.

Notes on Data Injection

We did not attempt attacks that operate by injecting data into a system without subverting control flow, primarily because it was not necessary. We was able to find exploits that allowed control flow to be subverted in each operating system that we examined. This allows for exploits that are strictly more powerful than those enabled by data injection. With the assumption that a control flow subversion enabled a ROP-like attack, the processor can be made to execute arbitrary code. If this is the case, then it can be programmed to inject any desired data itself.

It may be that data injection attacks become more interesting if better defence mechanisms are deployed that manage to prevent control flow attacks. There are several promising avenues of exploration for data injection attacks against a system with a fully functioning IOMMU, particularly from the attack platform when it is acting as a NIC. These include ARP poisoning attacks, similar to those presented in [75], discussed earlier in Section 2.11; and also phishing attacks, where the fake NIC claims that it handles the IP addresses for email and banking services and presents a fake login page for harvesting usernames and passwords.

5.1 ATTACKS FROM A BASIC PLATFORM

Our earliest attacks used a Nios II processor, rather than a BERI. They used the Altera PCIe hard core to handle configuration requests. This meant that the attack platform presented its device and manufacturer ID as 'Altera FPGA', so the host would not load a driver for the device. We used this device to verify the lack of use of the IOMMU by default on Windows, Linux and FreeBSD by reading data from a variety of memory locations, and also ensuring that we could write to memory locations. We did not attempt to conduct any further attacks against Windows, as there are not significant further protection mechanisms provided with the operating system, as discussed in Section 3.1.

Attacking macOS

MacOS enables the IOMMU by default. Despite this, we were able to both read private data and overwrite control pointers from the attack platform.

Reading from host memory is straightforward. Although much of it is shielded by the IOMMU, there are still many areas of interest that can be read by the basic platform. If the platform does not have permission to access a given page, PCIe returns an 'unsupported request' error code. We were able to sweep through memory and find many open windows quite quickly, by simply ignoring the responses that had this error code.

It was also possible to carry out a control-flow subversion attack against macOS with a simple platform. We later reimplemented the attack to ensure that it still worked against modern versions of the operating system, and to ensure that each step of the attack was fully understood. The attack works by exploiting a vulnerability in the *mbuf* data structure, which carries network packet data. This is part of macOS that has been inherited from FreeBSD, although the codebases in the two operating systems have since diverged slightly.

Each mbuf in the system is exactly 256 bytes in size. A variable amount of each mbuf is used to hold network stack metadata. Network packet data can either be stored directly within the mbuf, or in a 2KiB external structure called a *cluster*. As network data may be larger than an mbuf's data section or cluster, packets are stored in *chains* of mbufs. The amount of metadata stored in an mbuf depends on whether it is the first in a chain, and whether its data is internal or external. We give the mbuf definition used in recent versions of macOS in Figure 5.1. The FreeBSD original defines the mbuf as one monolithic struct, with anonymous unions and structs for metadata. MacOS used to take this approach, but around five years ago switched to separating out the optional parts of the mbuf into their own structs. In order for the old references to work, it uses a series of macros. For example, '`#define m_next m_hdr.mh_next`', which prevents any other struct in the kernel from having a field named `m_next`. This approach allows the compiler to automatically calculate the space for data available in an mbuf, by subtracting the size of the various header structs from the mbuf's predefined total size.

Every mbuf has a common header, with fields detailed in Table 5.1. The `m_flags` field of this header determines which of the extra headers are present. This is a manual implementation of the tagged union data structures commonly found in functional programming languages. The `pkthdr` struct is present if the mbuf is the first in a chain, and contains 136 bytes of extra metadata. The `m_ext`

```
struct mbuf {
    struct m_hdr m_hdr;
    union {
        struct {
            struct pkthdr MH_pkthdr;     /* M_PKTHDR set */
            union {
                struct m_ext MH_ext;     /* M_EXT set */
                char MH_databuf[_MHLEN];
            } MH_dat;
        } MH;
        char M_databuf[_MLEN];           /* !M_PKTHDR, !M_EXT */
    } M_dat;
};
```

Figure 5.1: MacOS mbuf definition. _MHLEN is 88; _MLEN is 224.

Field	Type	Description
m_next	Pointer to mbuf	Next mbuf in the chain representing the same packet.
m_nextpkt	Pointer to mbuf	Mbuf that starts the chain representing the next packet.
m_data	Void pointer	The start of the data represented by the mbuf.
m_len	32-bit integer	Amount of data in the mbuf.
m_type	16-bit unsigned integer	Type of data in the mbuf.
m_flags	16-bit unsigned integer	Packet metadata.

Table 5.1: Fields Common to Every MacOS Mbuf.

struct, detailed in Table 5.2, is present if the mbuf's data is stored externally. It is worth noting that even if the pkthdr is not present, the m_ext fields always occur after it in the mbuf. The rationale here is that it simplifies the design of the struct, as there is precisely one place where each field can go; if the data is external it does not matter how much space for data there is internal to the mbuf.

The crucial detail here is the ext_free function pointer. Mbufs are allocated in page sized blocks, with each page containing 16 mbufs. This means that on macOS when a window is opened for a single mbuf that carries its data internally, the metadata for 16 mbufs is made visible to every peripheral in the system. This gives us everything we need for an attack. We have two variants of this attack, an initial proof-of-concept, and a later attack to ensure that each step of the attack was robust and understood. The attack proceeds as follows:

Field	Type	Description
`ext_buf`	Void pointer	Pointer to buffer.
`ext_free`	Function pointer	Pointer to a non-standard function for freeing buffer.
`ext_size`	32-bit unsigned integer	Size of buffer.
`ext_arg`	Void pointer	Extra argument for free.
`ext_refflags`	Pointer to struct	Reference count information.

Table 5.2: Header Data in MacOS Mbufs with an External Payload.

1 *Find a vulnerable mbuf.* There are several ways to go about doing this. Our initial attack looked for a NIC transmit ring in memory, and followed the addresses contained therein to find pages containing mbufs, but a simple linear scan of memory was sufficient. Both of these approaches rely on the use of heuristics to detect mbufs. The design of an appropriate heuristic relies on a substantial amount of trial and error. It must be broad enough to find a large number of potential targets, but specific enough to minimise destructive writes to unrelated areas of memory.

The proof-of-concept attack looked for a very specific descriptor ring, which had each buffer address being 4GIB aligned and in the top half of memory. We were not sure what these buffers were used for, and while the structures were found reliably in macOS 10.11, they were not found in macOS 10.12. The data pointer for each entry in the ring points to either an mbuf or a cluster. If an address in the ring is 2KIB aligned, it is a pointer to a cluster. Due to the presence of header information before the data section of the mbuf, the address of the data within it is never aligned.

Scanning for a page of mbufs directly is subject to a variety of complications. An mbuf that is not currently in use by the network can have all of its fields as zero, so it is possible for pages of mbufs to exist that consist entirely of zeroes. These pages are impossible to disambiguate from areas of memory that may be zero for any other reason. The existence of these pages slows down the process of scanning, as we potentially have to scan the whole page to determine if it contains any mbufs. Our heuristic for detecting mbufs makes sure that the pointers to other mbufs are 256-byte aligned, that the type and length fields are valid values, and that at least one of the next, next packet and data fields is non-zero, to allow a potential mbuf to be disambiguated from zero memory.

2 *Ensure the mbuf is marked as having external storage.* Otherwise, when the mbuf is

freed, the OS will not attempt to call the custom free function. This can be done in one of two ways: either an mbuf that already has external storage can be used, which is the approach we initially took, or the `M_EXT` flag can be set in the `m_flags` bitfield manually by the attacker.

3 *Set the reference count to one.* Otherwise the OS will kernel panic when it attempts to free it. This is complicated as the `ext_refflags` field is a pointer to a struct, rather than a struct itself. The solution is to create a fake struct and embed it in the data section of the mbuf. This then requires `ext_refflags` to be set to the kernel address of the mbuf's data section, but this can be found straightforwardly as it is the `m_data` pointer in the mbuf header.

4 *Set the address of the function pointer.* The kernel address of the desired function can be set to an arbitrary value. It does not have to be set to a valid function start, as the function call operation is essentially an indirect jump surrounded by some stack bookkeeping. On macOS, the attacker has a significant amount of control over the arguments of the function: they are the 64-bit pointer to the buffer, as given in `m_ext`, rather than the data pointer in the mbuf core; the 32-bit integer size of the buffer from the `m_ext`; and the optional 64-bit `ext_arg` field. Useful addresses can be found from a disassembly of the relevant version of the macOS kernel. These are freely available.

Breaking KASLR

There is one further level of complication to the attack imposed by macOS. It implements *Kernel Address Space Layout Randomisation* (KASLR), as detailed in Section 2.7, so the address of any given symbol in the kernel is not at a predictable address. On macOS, KASLR is implemented by adding a constant offset, known as the *KASLR slide* to every address within the kernel. The slide is an integer multiple of 2MiB, selected so as not to interfere with the relative positioning of symbols within a large page. This means that if we know the physical address of any symbol, we can simply subtract its 'unslid' location from its actual location, and discover the slide.

We have such a symbol. The symbol '`gIOBMDPageAllocator`' is revealed to peripherals at least twice within the kernel, from the USB and AHCI drivers. We are unsure why, as the drivers in question are closed source. The symbol is a '`struct iopa_t`' which presumably holds bookeeping information for IO Physical addresses. Its definition is given in Figure 5.2. The leaked pointer is actually at an offset of eight to the symbol's location in the disassembly, which means that it is

```
struct iopa_t
{
        IOLock          * lock;
        queue_head_t    list;
        vm_size_t       pagecount;
        vm_size_t       bytecount;
}
```

Figure 5.2: `struct iopa_t` Definition

actually the `list` structure that is leaked. This wraps a 'next' and 'prev' pointer, allowing the struct to be kept in a doubly linked list. It seems likely that this pointer is not exposed to the device deliberately. It is more likely that some metadata that needs to be visible to device is on the same page, and the IOMMU's 4KiB granularity means that this pointer is visible as well. The driver should be rewritten so that the data that needs to be made visible to the device is on its own 4KiB page.

Escalating the Attack

Being able to force one kernel-level function call is not obviously a catastrophic breach of security. It can be used to call the '`panic`' function for example, to induce a kernel crash, but a badly implemented device can cause the same thing. A more promising avenue of attack is a function called `KUNCExecute`, which can be called from the kernel, and which spawns a process as any user, including root.[1] It has been deprecated for a over a decade, but is still included in the kernel.

However, our ability to call it with useful arguments is restricted. The first argument to `KUNCExecute` is a C-style string – a pointer to a series of sequential `char`s, termindated by a zero or 'NUL' byte – containing the path of the program to execute. We have a 64 byte first argument that we have control over, but our knowledge and control over host memory is much more limited. Even though there are large areas of memory that we can write into, our scan only reveals their IOVAs, rather than their kernel addresses. The simplest approach is to use the unused data section of mbuf that we are using as our attack vector to contain the string. However, that restricts us to using a 23 character string. Probably the most useful application to start from the perspective of an attacker is the terminal, which has a path of `/Applications/Utilities/Terminal.app/Contents/Ma-`

[1] It is telling that if you Google for the function, the sixth entry is called 'Developing Mac OSX kernel rootkits'.

`cOS/Terminal`, which is 60 characters.

The simplest approach, which we used in the first version of the attack, was to create a shell script which then called the Terminal in turn, which had the desired effect. This allows an escalation of privilege, as most users of the system will be able to create an applicable shell script.

An alternative approach would be to use the 136 bytes normally occupied by the `pkthdr` struct to contain the string. This requires that the `M_PKTHDR` flag is unset, or the kernel will attempt to interpret the `pkthdr` information when freeing the packet, causing the system to crash.

Potential Escalations

However, given such privileged access to the kernel, and having seen the potential of ROP-style and return-into-libc attacks (see Section 2.7), we would like to do better. We have not implemented an attack of this nature, but have performed some initial investigatory work, which we describe here.

The IOMMU can be disabled on a per-device basis basis by setting a bit in the context entry – the second level IO page table entry – for that device (for more information on IOMMU configuration structures, see Section 2.6). MacOS does not have a public API for doing this, so it must be done by manually inserting the record in the relevant place. An attack would involve creating an entry for the device in a page that we controlled at a kernel addresses known to the attack platform. We can use an mbuf that carries its own kernel address as part of its data pointer to find some suitable memory. However, the kernel may try to free and mbuf that we have written over. As the mbuf will appear invalid this will cause a kernel panic. As mbufs are freed in chains, we can change the pointer to the next item in the chain and prevent the later mbufs from being reclaimed. With the fake context entry constructed, we can change the entry in the root table to point to our fake entry.

More recent versions of VT-d are able to cache invalid transactions in the IOMMU. If the attack platform has made any of these before disabling translation for itself, it must either wait for a TLB flush or trigger one manually.

This attack would require various snippets of code to allow the location of the IO page tables to be discovered and specific entries within them to be overwritten. If these snippets are not present in the system, it may still be possible to carry out a ROP attack. The attack platform would not be able to overwrite the stack directly, but it may be possible to create a fake stack in an area of memory the attack platform has access to, and then use a snippet of cope to overwrite the stack pointer to point to the new stack.

5.2 CUSTOM CONFIGURATION RESPONSES

The next step in complexity for the attack platform is programmable control over PCIe configuration headers. This allows an exploit that gives us full memory access to Linux platforms. To do so, we make use of PCIe's ATS feature, described in detail in Section 2.3. As soon as the device is given control of its own translation, it is fully trusted. The only protection mechanism built into ATS is that the host has to enable it explicitly. FreeBSD does not mention ATS anywhere in the codebase; the relevant parts of macOS are largely closed source, but we believe it does not support it. An examination of the Linux codebase seemed to reveal that Linux would automatically enable ATS for any device that displayed the relevant PCIe capability. We verified that this was indeed the case by creating the simplest possible model of an Intel 82574L NIC that Linux still recognised as valid. We then modified the model to include the ATS capability in its PCIe configuration space, and modified the code for making DMA requests to include the address translated bit. This allowed me to conduct arbitrary reads and writes to host memory, despite IOMMU protection still nominally being enabled.

A Minimal 82574L NIC Model

The minimal 82574L NIC model implements a fairly complete image of the NIC's configuration space using a 1KiB area of memory on the attack platform. Allowing the host to write to this memory was largely straightforward. We implemented a write-masking mechanism to allow regions of the configuration space to be marked read-only at a per-bit granularity. Adding the ATS capability to this model required just two lines of code, shown in Figure 5.3.

In addition to the configuration space, a further three dwords of state must be maintained by the attack platform. These emulate some device specific behaviours that are accessed via the host making memory request type TLPs. The registers that these emulate are:

EEPROM Read At memory location `0x14`. The EEPROM cannot simply be mapped to a BAR by the host, and must instead be read via a state-machine-based protocol through this memory register. To read a 16-bit portion of EEPROM,

```
/* ATS capability: dword [0] */
C(0x100, PCI_EXT_CAP(PCI_CAP_ID_ATS, PCI_ATS_VERSION, 0x0));
/* ATS capability: dword[1] */
C(0x104, PCI_ATS_PAGE_ALIGNED_REQUEST);
```

Figure 5.3: Code for adding an ATS capability to a basic 82547L model. The macro 'C' assigns its second argument to specific addresses in the configuration space given by its first argument. The address is given in bytes, as this is how addresses are presented in the specification, but the memory area is allocated at DWord granularity, which helps with endianness conversion. PCI_EXT_CAP is a macro provided with QEMU to help with construction and readability of PCIe capabilities. It could have been replaced with a constant integer.

the host writes an address to bits 15:2 of this word and a one in bit zero to indicate that it wants to start reading the location. It then reads the same register, and is presented with the data from the requested location in EEPROM in bits 31:16 with bit one set to indicate the read is complete. If bit one is not set, then FreeBSD will repeat the read until it times out.

By default, the model returns a value of zero in response to all EEPROM reads. This is sufficient for all values except the MAC address and the checksum. The MAC address can be any non-zero value. We use '0x2'. The checksum register then has to cause the sum of all 16 bit values in the EEPROM to be the value '0xBABA'.

MDI Control (MDIC) Register At memory location 0x20. This is the *Management Data Interface Control* register. It is used for communicating with the PHY, the chip used to interface with the physical layer of the Ethernet protocol. It has a similar protocol to the EEPROM register. Bits 15:0 of the DWord are used for data; bits 20:16 are used for the address of the register within the PHY; bits 25:21 for the address of the PHY itself, which is simply '0x1'; bits 27:26 are the *opcode* – read or write; bit 28 is the ready bit to indicate that the transaction has been completed by the PHY chip. The remaining bits are for various pieces of control and bookkeeping.

For the device to attach, we had to emulate two PHY registers. Register two is the *PHY identifier register*, which is set to 0x0141. Register three is the *Extended PHY ID*, which is set to 0x0CB1.

Extended Configuration Control At memory location 0xF00. This is a collection of somewhat miscellaneous functionality. The relevant use is governing access to the 'Management Data Input/Output'. The host sets various bits within it in order to request permission to write the MDIC register: if it can't read back the bit it has written, it assumes permission has not been granted, and keeps trying until it times out. We simply echo back what the host has written on a read.

Recieve Address Low At memory location `0x5400`. This is the one of two registers used to store the 48-bit Ethernet MAC address. We set it to '`0x2`' so that it agrees with the value read from the EEPROM.

For every other request that expects a response, the attack platform returns a default 'success' response; any data expected is returned as zero.

Carrying out the attack is straightforward. As soon as the host writes a zero into the position of the PCIe capability register that marks ATS as being enabled, the attack program begins reading eight byte sections of each page in the system by making read requests with the address translated bit set.

We found that this was sufficient to be able to read areas of system memory that were protected by the IOMMU if ATS were not enabled. We conducted further experiments to confirm that abuse of PCIe's ATS feature allowed full access to host memory, rather than the IOMMU being accidentally disabled for some other reason. With the ATS capability not present, the host marks requests with the address translated bit set as being unsupported requests, demonstrating that the host implements the ATS specification properly. With neither the ATS capability present, nor the address translated bit set, the host returns unsupported request completions as it there are no open windows into host memory for the attack platform.

This result is consistent from the understanding we derived from reading the Linux PCIe code. Even though no real 82574L NIC implements ATS, adding the relevant PCIe capability to a minimal model of one, and modifying the relevant bit in the memory requests it sent was sufficient to effectively disable the IOMMU.

5.3 FULL NIC EMULATION

The use of ATS is sufficient to allow attacks in the face of Linux's per-buffer mappings. However, as FreeBSD doesn't support this feature, this attack does not work against it. It does, however, implement mbufs in a similar way to macOS – including the use of a custom free function pointer – so if it were possible to gain access to an mbuf, a similar exploit should be possible. This realisation motivated the implementation of the fake NIC on the attack platform, in order to get FreeBSD to open windows containing the vulnerable data structure directly for the use of the attack platform. Details of the design of the NIC software stack on the attack platform are given in Section 4.4.

Subverting Control Flow

The control flow subversion vulnerability we exploit is almost identical to the one on macOS, although it does involve some differences as a result of differences in mbuf structure. We exploit a transmit mbuf, because these have their free function called by the host immediately after they are reported to leave the NIC. Receive mbufs may be freed at any time, depending on when the host processes the data they contain. A convenient place to exploit a transmit mbuf from QEMU was the function `start_xmit`. This loops through the descriptors in the card's transmit ring, and passes packet data into the selected QEMU network back end. We inserted an exploit function that is called with the IOVA of each buffer just before it is passed to the network back end. The exploit function operates as follows:

1 Return immediately if the buffer address is 2048 byte aligned, as this indicates an mbuf cluster, which is not suitable for exploiting.
2 Read the mbuf out of host memory into memory on the attack platform.
3 Convert the mbuf to big endian in order to be able to modify it on the BERI using standard struct syntax.
4 Modify the mbuf to have an appropriate reference count and set the appropriate flags to indicate that it has external data and a custom free function.
5 Set the address of the custom free function.
6 Convert the mbuf back to little endian.
7 Write the newly modified mbuf out over the old mbuf.

The biggest complication when attacking FreeBSD rather than macOS is that the attacker has slightly less control over the free function arguments on FreeBSD. On FreeBSD, the free function is always called with the address of the mbuf to be freed as its first argument, with the user able to specify custom arguments of 64-bits as the second or third arguments. It seems likely that this was adopted by chance, rather than as a deliberate security measure.

We were able to call the kernel panic routine on FreeBSD by subverting the function pointer. By replacing the first few bytes of the mbuf with the string 'BAD NIC!', we were able to confirm that the exploit was causing the function call. Writing over the start of the mbuf does not constitute a general purpose mechanism for passing arguments to the called function, as any value that isn't a kernel pointer to an mbuf causes the mbuf free routine to panic as it attempts to free the chain of mbufs. FreeBSD also lacks a function to call that is as compelling as the ‘KUNCExecute’ routine in macOS.

A sufficient number of packets necessary to trigger the kernel pointer subversion attack are found over the course of executing ‘dhclient’ on FreeBSD. In a simple setup, this does not happen by default, but it does when a desktop environment with a network manager widget is used.

We believe the escalation strategies that we presented for macOS are equally relevant to FreeBSD, although the snippets of code necessary to carry out the presented attacks may be less likely to occur due to the more specialised requirements on the function arguments. On the other hand, the NIC can ensure that the host does not try to free mbufs by simply reporting them as having not been sent. This simplifies attacks that require an area of memory that has a known kernel address.

KASLR on FreeBSD would thwart this attack, but it does not implement it. Even if FreeBSD did implement it, if it had a symbol leak in the same way as macOS, we would be able to exploit it. Even though the NIC-based attack platform would not necessarily be able to see the leaked symbol, we could impersonate another attack device which would be.

Reading Private Data

The problems we demonstrated on macOS and FreeBSD are severe. Being able to subvert a kernel function pointer can have extremely damaging effects. However, it does rely on an mechanism that is specific to these two operating systems: a function pointer that is close in memory to the data of an mbuf. The equivalent data type to an mbuf on Linux, an sk_buff, is sufficiently large that the data for the packet represented by the sk_buff is always external to it, so our attack does not work. The approach of placing all network data in clusters could be applied to macOS and FreeBSD too, although it may have a detrimental effect on network stack performance. It would lead to external fragmentation of mbufs, leading to a larger area of wasted cache space and require additional memory allocations.

In order to demonstrate that it was still possible to carry out attacks against an operating system that used current best practices, we created a memory reading attack against macOS that did not make use of any IOVAs that were not directly passed to the fake NIC based attack platform. From the fake NIC, we managed to read the plaintext of data that had been sent encrypted with IPsec over the real

NIC in the system.

The attack operated as follows:

1 Each time a buffer address is passed to the NIC, add the address of the page it is on to a linked list of open windows.

2 Periodically scan through the entirety of the mapped windows.

We set up an experiment where a Python script constantly sent a message consisting of repetitions of the character 'i' over IPsec to a remote host. The attack platform revealed many instances where the plaintext character 'i' was visible in pages that were exposed to the fake NIC. This is not because the host was trying to use the fake NIC to send the data: the fake NIC was just sitting on the system, while the Mac used its internal NIC to send the encrypted data. The plaintext ended up in the same page because mbufs are shared across devices, and encryption and decryption is performed in place.

The only way to foil this attack is by restricting the memory areas visible to the device, so mechanisms like KASLR are not sufficient. Mitigating it requires either new hardware in the form of a byte-granularity IOMMU, or for the OS internals to be restructured to have a separate pool of mbufs for each NIC. The latter change is not particularly conceptually complicated, but there is currently no binding between NICs and mbufs in the kernel. Worse, mbufs are used by drivers, so it is potentially the case that each driver would have to be modified individually.

5.4 EFFECTIVE DEFENCES

The attack presented in Section 5.1, with its use of memory scanning, is not sufficient to attack FreeBSD or Linux with per-device mappings switched on.

The attacks that rely on subverting control flow would similarly be thwarted by robust defences against control-flow subversion, as described in Section 2.7. This includes ASLR without leaked symbols, or a mechanism like CPI, as long as the data structure that contained the protected pointers was also protected by the IOMMU.

The eavesdropping attacks would require a more specialised defence mechanism. Using the IOMMU in a bounce-buffer configuration or the RIOMMU IOMMU design described in Section 3.2 would be able to achieve this.

The only current effective defence against ATS is simply not to enable it. We discuss policies for this, and other potential defence mechanisms in Chapter 6.

	Vulnerabilities Exploited				Attacks Achieved		
	ATS	*Shared Mappings*	*Granularity*	*Lifetime*	Control Flow Hijacking	Eavesdropping	Disables IOMMU
Basic Platform §5.1		✓		✓	✓	✓	
Custom Configuration §5.2	✓						✓ *
Full NIC §5.3			✓		✓	✓	

* Technically, we did not disable the IOMMU, but were able to access the system as though it were disabled from the attack platform.

Table 5.3: Summary of our novel attacks.

5.5 SUMMARY

In this chapter, we have presented four different attacks, using different vulnerabilities. These attacks are summarised in Table 5.3.

We have discussed several plausible attack scenarios that make IOMMU vulnerabilities a real and pressing concern. A cheap attack platform can be created that uses PCIe's DMA capabilities over a USB-C connector, which is used for tasks as ubiquitous as charging and connecting external storage and displays. Windows, Linux and FreeBSD make no attempt to use the IOMMU by default, leaving these platforms very vulnerable to DMA attacks. The protection used on macOS can be broken by an attacker without emulating any sort of real device, because macOS shares all of its IOMMU mappings with all DMA capable devices, the granularity of IOMMU windows is much larger than the data that has to be protected, and macOS stores kernel function pointers near network data. MacOS also leaks a kernel pointer that allows its KASLR protection to be broken. Linux can be fooled by a device that presents an extremely minimal approximation of a real device's behaviour into enabling ATS, a PCIe feature that allows a device full access to host memory, despite the simulated device not actually implementing this feature at all. Using a behavioural model of a NIC extracted from QEMU, we were able to exploit the same vulnerability we exploited on macOS on FreeBSD, allowing a kernel function pointer to be set to a known location. Finally, using nothing but information passed directly to a behavioural model of a NIC, we were able to read plaintext data that should have only been sent encrypted over a real NIC in a macOS system. This attack does not even require KASLR to be broken, and was possible due to MacOS's long lifetime mappings for its mbuf network data structures, and the lack of granularity of the IOMMU.

FUTURE DEFENCES 6

In this chapter, we suggest defences against DMA attacks that have not been presented in the existing literature and look at their likely efficacy if they were to be implemented. Not all of these directly involve the IOMMU. DMA is an attack vector, rather than an exploit by itself, so any DMA attack has a number of steps. If it is possible to prevent any of these, then the attack as a whole can be thwarted. This means that there are a number of non-IOMMU based mechanisms that can be used to prevent DMA attacks. This includes, for example, ASLR schemes, although these sometimes need modification in order to work against DMA attacks.

6.1 ROBUST DRIVERS

When a user-mode program encounters an unrecoverable error, it will typically display an error message and quit. When most drivers we have investigated encounter an unrecoverable error, they cause a kernel panic. This happened many, many times over the course of developing the malicious peripheral. Some of the previous work we have discussed has shown that a malicious driver running within a guest operating system can crash its host hypervisor by triggering a kernel panic [67].

This propensity towards kernel panics makes it easy to create malicious peripherals that carry out DOS attacks. There are many cases where a peripheral that appears to be displaying faulty behaviour could simply be disabled, rather than locking up the host entirely. This necessitates more complicated error handling than simply triggering a kernel panic. For example, a NIC may be in the process of communicating over the network. As there is no way to verify if a packet with a corrupted mbuf has been transmitted, it is probably desirable to kill the process that generated it, and allow the user to start it again from scratch if desired. This sort of behaviour would involve significant changes to the kernel and to the individual drivers, but these changes are beneficial if they can prevent an attacker from forcing shutdowns on vast numbers of shared compute resources at low cost.

6.2 SUSPICIOUS BEHAVIOUR DETECTION

When an invalid access is made by a device, the IOMMU reports this to the kernel. In response to this, current operating systems merely log that such a request has been made. On macOS, for example, this allows a crude, brute-force scan of memory. Changing this behaviour can provide better security relatively straightforwardly. In the course of normal operation, a device does not make any invalid requests. The OS could detect a certain number of invalid requests in a given time period, and, in response to this disable all requests from the device for a period of time. This could be achieved by setting the relevant bit on the IOMMU entries for the device, and performing the appropriate TLB flush. However, detecting invalid requests does not prevent attacks where a device's behaviour is impersonated in detail. There is also the potential for false positives to occur, although, in practice, we have not seen a legitimate device make even a single invalid request, and were a device to do so, it would indicate a fault with the device driver.

6.3 CRYPTOGRAPHIC HANDSHAKING

Operating systems load drivers for PCIe devices based on their manufacturer and device IDs. These are publicly available and easy to spoof. It is possible that some form of cryptographic handshaking is possible, so that only trusted devices are granted DMA. One possible scheme is based on public-key cryptography. When a company manufactures a peripheral, it creates a private key for that peripheral that is embedded securely onto the peripheral, perhaps on a separate secure processor. It then disseminates the public key for that peripheral over a trusted, public channel, which may be an OS update mechanism or similar. To confirm that the device is real, the host OS encrypts a random string with the public key, then passes it to the device, which must decrypt it with the private key. If the string that returns is the same as the plaintext, then the device must have the relevant private key.

However, this scheme has many potential problems. If the private key is leaked, then the device can be impersonated, and it is not possible to prevent devices with that key from working, because this prevents the use of many legitimate devices. The channel over which the keys are shared is another weakness, although it does not necessarily provide more vulnerabilities than the OS update mechanism itself. There is the need for some central agency to decide which devices are given keys at all. This raises cost, and may discourage innovation in peripherals. It may also be fooled by a very determined adversary. Finally, even if a scheme like this is introduced, it must still be possible to use legacy peripherals that require DMA. The use of cryptographic handshaking is not a convincing solution to the DMA attacks that exist today.

6.4 ENCRYPTED MEMORY

Encryption of main memory in commodity systems has been widely investigated [36]. Existing work has largely been in the context of protecting against, for example, cold boot attacks [34], where physical DRAM modules are chilled and removed from a computer so that they retain data which can then be read by an external machine. The level of protection against DMA attacks that is provided by memory encryption is unclear. Hardware encrypted memory schemes encrypt data as it leaves the cache for memory, and decrypt it as it re-enters. Attempting to encrypt memory without specialised hardware carries severe performance overheads, as the encryption must take place when register contents are written to cache. Intel report that with their AES instruction set extensions, a core can encrypt data at a rate of 4.2 cycles per byte, and decrypt at a rate of 1.3 cycles per byte [5]. This means that writing a 64-bit register to L1 cache takes an additional 34 cycles, an overhead of approximately 6×.

Crude memory encryption schemes that use an *Electronic Code Book* mode where each unit of encryption – typically a 128 bit chunk – are encrypted to the same result are vulnerable to frequency analysis, and potentially to a chosen plaintext attack. More sophisticated schemes use counter mode, which does not have this problem.

The interaction between an external device and computer memory complicates the implementation of memory encryption. A number of possible schemes are possible:

Encryption Unaware Devices Devices are used unmodified. This means the CPU must have a mode for bringing data into the encrypted domain. There would be a performance overhead due to additional encryption and decryption. It would also permit a malicious peripheral to read and write private data as it left or entered the system. With the symbol leaks we have seen, it is not clear the memory encryption by itself would prevent control flow attacks. In macOS, the leaked symbol is visible to all DMA-capable devices. For encryption to prevent it revealing the KASLR slide, it would have to operate at a fine enough granularity that the symbol's address was encrypted, while the data around it on the page was not.

A Shared Publically Available Key Devices are expected to encrypt and decrypt their data using a publicly available key different to that used by the CPU. This has the same security properties as the previous scheme, but requires new peripherals.

Per-device Keys Each device encrypts and decrypts its data using a unique key. The host has to support encrypting different areas of memory under different keys. At present, AMD's encrypted memory scheme uses a single bit of address to mark a physical page as encrypted or not [43]. This could be extended to multiple bits to refer to multiple keys. To work efficiently, it would be necessary to know which data was to be allocated to which device ahead of time. Use of such a scheme prevents a malicious peripheral from reading private data destined for a peer. However, its design and implementation would be complicated. It needs a mechanism for securely sharing a key between the CPU and each peripheral, and it is notoriously difficult to achieve this correctly. It also requires each peripheral to have hardware to enable the encryption. This has the same backwards compatibility problems as the use of a shared key.

6.5 FINER GRANULARITY MAPPINGS

Several of our attacks are possible because IOMMU mappings are made at the granularity of a 4KiB page, while device buffers are often much smaller than this. Probably the most conceptually simple solution to this problem is to design an IOMMU which has mappings of a sufficiently fine granularity to prevent these attacks. Creating an efficient design for this is not straightforward. Various schemes are possible, including increasing the depth of the page tables, associating one or more base and bounds with each page mapping, or associating a bit mask with each mapping. Of these, we believe the scheme that provides the best combination of security, performance and engineering effort is to associate a single base and bounds with each page, and to ensure that only a single buffer is mapped in each

IO virtual page.

The most obvious solution is to add another layer to the IOMMU page tables, resulting in 8 byte 'pages'. This level of granularity would be small enough to prevent the attacks we have demonstrated, but a number of problems prevent this from being a realistic solution. Primarily, a large number of mappings would have to be created for each buffer. The creation and revocation of these mappings would carry a performance overhead, and the effect on IOTLB usage would be enormous.

More promising alternatives involve using a system with the standard 4KIB page size, but with additional metadata to enhance security. Each mapping could have an associated base and bounds, so that accesses outside of these bounds are disallowed. This system works well in the case where only one buffer is mapped per page. However, this is extremely rare. For example, on FreeBSD, mbufs are allocated from a contiguous regions of identical data structures.

When allocating multiple buffers per page there are a number of options. Firstly, we could simply expand the base and bounds of the mapping to cover both buffers. This is unworkable. Any page that contains two mbufs will reveal the header of the second mbuf, as it is placed between the data sections of the two mbufs. Additionally, it is not possible to shrink mappings after they are no longer necessary without keeping track of why each region of the mapping is mapped. Consider the case of a mapping with three mbufs, a 'left', a 'middle' and a 'right' mbuf. Say the middle mbuf is freed first. Now there is a large hole in the mapping that the target device can access unnecessarily. Now say the left mbuf is freed. We can reduce the mapping to just include the right mbuf, without preventing any legitimate accesses from the device, but we cannot know that we need to do that without independently keeping track of the area of the page that should be mapped, because there is no information in the page mapping the specify that the mapping area for the middle mbuf is no longer necessary.

Secondly, we could add a separate mapping for each buffer. This would require modifications to the traditional page table tree. The most obvious solution is for each last level page table entry to point to a linked list of table entries. This would incur more overhead on page table access: a more efficient data structure than a linked list might be possible, but would still carry overhead relative to a single mapping per page. The IOTLB would require a redesign to be able to cope with the increased number of mappings and finer granularity. Potential problems with IOTLB design come from the fact that if we have high numbers of buffers mapped per page, we need to have high levels of associativity in the IOTLB to avoid churn.

We can avoid having to make significant changes to the IOTLB by requiring that only one buffer is permitted per each IO-virtual page, even if multiple of these can map into the same physical page. That only one buffer is allocated per IO-virtual page, rather than each physical page is crucial. It means that true spatial segregation, as discussed in Section 6.6 is not necessary. Instead the OS's IOVA allocator can be modified, which is a simpler change that has to happen only in one place, and does not have the effect of fragmenting system memory. In this scheme, the interpretation of IOVAs is unchanged. The bottom twelve bits of each IOVA are used as an offset into a mapped window. The high 52 bits are used to index a multi-level translation structure. The entries in the multi-level translation structure contain not only a physical page number, but a twelve-bit physical offset and length. The sum of the twelve-bit physical offset and length must be strictly less than the size of the page, to simplify the hardware needed to implement the scheme. The optimal definition for the encoding of the base and bounds depends on an analysis of common buffer sizes, and the trade-offs between security and increasing the size of the page tables.

The steps taken to translate an address for a request are:

1 The requester ID and high bits of the IOVA are checked in the IOTLB to see if a translation record is cached. If not, it is looked up in the translation structure and inserted into the IOTLB.

2 The bottom twelve bits of the IOVA to be translated are added to the physical offset. The carry out bit from this calculation is checked: if it is one, then the sum of the IOVA offset and physical offset puts the resulting address outside the bounds of the physical page, and the translation is not valid.

3 Simultaneously, the IOVA offset is checked against the physical length. If the IOVA offset is too large, the translation is not valid.

4 The returned physical address is the physical frame number returned as part of the translation record concatenated with the summed IOVA and physical offsets. Because we have confirmed that the carry out from the sum of the IOVA and physical offsets is zero, we can perform a concatenation, rather than a sum.

This process is presented diagrammatically in Figure 6.1. Compared with a conventional translation, it requires an additional twelve-bit comparison and twelve-bit add that can be carried out in parallel. This should require no more than a single additional cycle of latency. On a modern processor, a main memory access takes of the order of 200 cycles [2], excluding the latency due to first missing the cache. Assuming that is this is a reasonable lower bound on the latency between

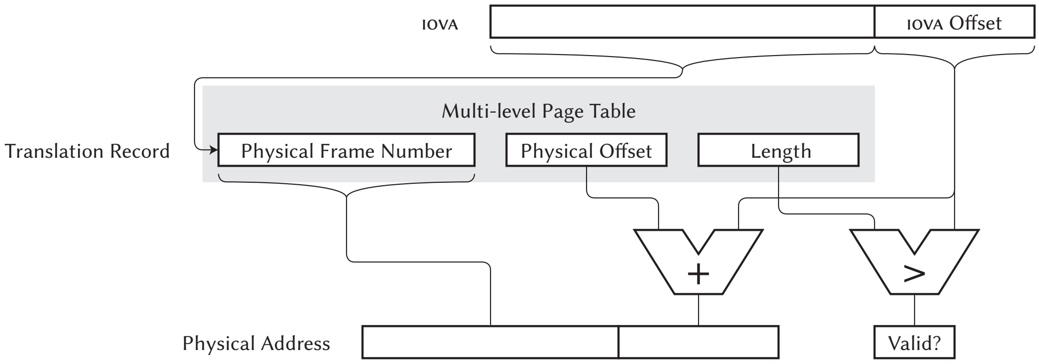


Figure 6.1: Diagram illustrating page table mapping with base and bounds. The use of the carry out bit as an additional validity check is not shown to simplify the diagram.

a PCIe peripheral and main memory, this means that the scheme should carry a latency overhead of no more than 0.5%. Some modern Intel processors allow DMA directly to last level cache. Information about the access latency from a peripheral to last level cache is not readily available. Taking the latency of access from the processor, 38 cycles [2], gives a latency overhead of 2.6%.

This scheme increases the size of the translation record that needs to be returned from the IOTLB, but by restricting the number of buffers to one per IO-virtual page, the same amount of data can be used to index the IOTLB. However, this restriction would lead to an increase in IOTLB pressure, as mappings that previously contained multiple buffers now contained only one. This overhead could be estimated using a modification to the allocation strategy on a system with a conventional IOMMU, and would be a sensible first step to take before implementing this scheme.

Various tweaks to the scheme are possible. For example, it would be possible to replace the offset and length with a minimum and maximum, and use comparisons with the offset, rather than an offset and comparison. This perhaps has more straightforward semantics when coming from standard page tables, at the cost of slightly more hardware. As the two comparisons can be carried out in parallel, it does not impose performance penalties when compared with the offset scheme. This allows the use of buffers that span multiple pages and do not start at the beginning of a physical page. With the addition-based scheme, this is not possible.

It is not possible to efficiently translate from a physical address to its IO virtual address with this scheme. However, this operation is not frequently used, so this is unlikely to contribute to a system-wide performance degradation. This scheme does inflate the number of IOTLB entries used, but would provide high granularity protection with a small hardware overhead and is worth more detailed investigation.

An alternative to a base and bounds based scheme is to use a bit mask, similar to AMD's DEV, as explained in Section 2.6. A 512-bit (64-byte) mask would be enough to cover a 4KiB page at a 64-bit granularity. This would inflate the size of the last level page tables: previously an entry was 64 bits; now it would have to be 576 bits. This would mean the last level page tables would increase in size from 4KiB to 36KiB, covering 9 pages rather than 1. For a 256 entry TLB, a mask for each entry would mean that the TLB becomes 16K larger, which is smaller than the size of the L1 caches on recent high-end Intel processors. The trade-offs of this scheme versus this base-and-bounds scheme are somewhat complicated. It requires significantly greater hardware changes, and provides slightly worse security. It also means that creating and modifying the page tables is more destructive to the processor's data cache. On the other hand, it requires less software modification, as the same IOVA allocator can be used, and allows multiple buffers per page, so does not cause an increase on IOTLB pressure. Further investigation is required to determine which scheme would be more appropriate.

Another approach is to use radically different design altogether like the 'rIOMMU', discussed in Section 3.2. As previously mentioned, this is not suitable for all workloads. Getting each device driver to select the IOMMU it uses is unappealing. It may be possible for the OS to select automatically based on device IDs or usage patterns.

It appears that in order to provide good security at high performance the design of IOMMUs must increasingly diverge from conventional MMUs. When we examine the differences in design goals between the conventional MMU and the IOMMU, this is perhaps not surprising. The MMU allows user processes to execute as though they are the only process on a virtualised instance of a host processor. The process' memory space is entirely self-contained, and communication with other processes is handled by the operating system. There are few parallels with the goal of the IOMMU, which must expose pre-existing areas of memory to peripherals so that data can be communicated with the outside world. While the similarity between requirements is arguably major – that they must translate different varieties of memory addresses for convenience and protection – the differences are significant enough to explain why practical MMU and IOMMU implementations could be very different.

It is not unreasonable to ask why the IOMMU is so similar to conventional MMUs in the first place. We suggest several explanations:

1 The IOMMU was not designed for security, but to allow real peripherals to be used from VMs. This puts the design goals for the IOMMU more in line with that of a conventional MMU. An OS provides its processes with a virtualised version of the running computer: a VMM provides each VM it manages with the same thing. The difference is that the resource being virtualised is not memory, but peripherals.

2 When the IOMMU was designed, IO performance was lower compared with memory and CPU performance, so the overhead from a slower IOMMU was relatively lower.

3 An expectation that OS designers would be willing to segregate data to be exposed into 4KiB pages.

4 A worry that OS designers would be unwilling to work with an unfamiliar mapping mechanism, and conversely a hope that expertise in programming conventional MMUs would be transferable to IOMMUs.

Changing the IOMMU at all is, of course, complicated as it requires the design and fabrication of new silicon. The IOMMU would need a mechanism to work the same way as the previous version in order to allow backwards compatibility, but this does not seem to be a particular problem, as both VT-d and AMD-V have many variants, and have evolved to include more advanced features. It also requires modifying the operating system to make use of it, but any improved protection mechanism will require OS modifications.

6.6 SPATIAL SEGREGATION

A conceptually simple alternative to finer-granularity mappings is to ensure that any page that is exposed to a device never contains data that the device should not be able to read. A free-pointer-override style attack similar to that we performed on FreeBSD and macOS was not possible on Linux because Linux allocates all of network packet data separately to its metadata.

It would be possible to push this approach across the kernel. On top of the initial engineering effort, this would require careful monitoring to ensure that no code that violated the principle ended up in the kernel. It could potentially also have a negative impact on performance, as being able to allocate data and metadata simultaneously reduces the total number of allocations that need to occur, and exhibits favourable caching behaviour. Certainly, Linux is widely regarded to have a less efficient network stack than FreeBSD [47].

6.7 BOUNCE BUFFERING

We have already discussed Markuze et al.'s proposed scheme that involves copying buffers into pre-existing mappings in Section 3.2. We believe that there are still possible performance improvements for a scheme similar to theirs. Marginal improvements could be achieved by using a CPU-based DMA engine, like that provided in 'Intel's IO Acceleration Engine' to perform the copying. However, such DMA engines are typically only included on high end workstations or servers, although, arguably, such systems are the only ones that need this extra performance.

It may be possible to get the benefits of system-life-time mappings and zero copy if the buffer can be allocated inside a pre-existing mapping, rather than needing to be copied there. This would require a lot of the effort of spatial segregation, as it would require special allocation calls for data that was going to be mapped, but provide higher performance. Additionally, it would require knowledge about which device the allocated buffers were destined for at the time of allocation. In many cases this shouldn't be a problem, because the data's destination is known at the point that it enters the kernel, and the relevant device can be determined based on this. If the development effort were made to implement this, we believe it would have many of the beneficial properties of bounce buffering and a lower performance cost.

6.8 ADDRESS SPACE LAYOUT RANDOMISATION

Address space layout randomisation (ASLR) was initially proposed to thwart ROP attacks, as presented in Section 2.7. It is naturally extended to the idea of 'IOASLR', where the layout of areas of device memory is randomised. This would thwart attacks that rely on portions of device memory being in deterministic ranges, but many of our attacks use device impersonation to cause the host to present IO addresses to the attacker. IOASLR would not be able to prevent these attacks.

6.9 SYMBOL HIDING

ASLR is vulnerable to attacks that manage to locate a symbol that has a known location within the binary. If such a symbol can be found, the slide can be found by a simple subtraction. We found that such a symbol is found on macOS, allowing us to break the KASLR implementation. This problem exists in more general purpose implementations of ASLR: for example, it is known that kernel pointers should not be revealed to userspace. Previously, peripherals have been implicitly trusted so it is not unexpected that symbols are revealed to them.

Drivers should be written so that pointers are not revealed to peripherals. If it is useful for a kernel pointer to be revealed to a peripheral, a table should be created in kernel memory, and an index into the table passed to the device instead. This requires modification of drivers, however, and there are many systems where drivers will remain outdated, and manufacturers who may be unwilling to make this change. Pointers leaked to devices are hard to check for at compile time. APIs for interacting with the IOMMU typically just reveal an area of memory to a device: there is no universal way to identify the semantics of what is revealed.

On macOS, IOASLR could make it implausible to find the leaked symbol address that allows us to break KASLR with a simple search. However, it would be possible to create a fake version of a peripheral that has a kernel pointer leaked to it in order to be able to quickly and deterministically find a kernel pointer. As PCIe allows multiple devices to be behind a bridge, this could be done from the same physical device as the main attack.

Hiding kernel symbols would prevent control flow subversion attacks, but would not prevent attackers from potentially being able to read private data.

6.10 FUNCTION POINTER INDIRECTION

As discussed in Section 2.7, ASLR-like techniques are one approach to preventing software control-flow-subversion attacks. Some other techniques are are Code-Pointer Integrity (CPI) [46] and ASLR-guard [49]. CPI aims to make ROP attacks impossible by making it impossible to change code pointers. ASLR-guard aims to prevent symbol leaks, so the ASLR slide can't be calculated. Both of these work in fundamentally the same way, by hiding the data to be protected in a special area of memory. This idea can equally be applied to pointers that are accessible by devices.

Such a scheme would involve the following: All function pointers used by the kernel are, rather than being used directly, stored in a *protected table*. This table must not modifiable by peripherals. This can be enforced with a number of compiler changes:

1 Every function pointer symbol is assigned an index in the table by the compiler.

For function pointers that do not cross function boundaries, a liveness analysis can be performed to keep the table compact. The size of the table can be bounded at compile time, and the table can be fixed in position.

2 The code sequence that would normally be emitted to make use of a function pointer is replaced with an indirect access through the table.

3 The functions for adding IOMMU mappings are modified to detect if any area of the mapping includes the table. If some part of the table is to be mapped, then either the program could fail, or the data could be copied to a bounce buffer. The table should be allocated on pages that contain no other data in order to prevent this from happening in practise.

The performance cost for such a scheme would require some amount of experimental work to accurately analyse. The use of function pointers is not particularly common, and this scheme would have little effect on code that did not use function pointers, apart from cache pollution due to the presence of the table.

Depending on implementation, indirect access of the function pointers through the table would take one to three instructions. The time taken for these instructions would vary widely based on how much of the table could be kept in the cache at any time.

These schemes are both part of an on-going arms race in computer security. A follow up paper to CPI found several vulnerabilities in the scheme as implemented [25]. ASLR-guard-like systems can be thwarted by timing-based attacks from userspace. These use high precision timing facilities and knowledge of cache structure to analyse the likely location of the kernel. A user with the ability to run a user-mode program on a host could then use this in tandem with a DMA-based attack. It may even be able to carry out the attack from a javascript enabled web page. These attacks are not necessarily relevant to the scheme we suggest here. We include them to illustrate the difficultly creating a wholly secure system.

6.11 NOVEL PERIPHERAL INTERCONNECTS

As a concept, DMA is quite low level. The contents of each TLP describe how an operation is carried out, not what that operation is. Rather than specifying ‘a packet is delivered’, they specify ‘this data should be written to this memory’, where the data represents a packet. This approach has its advantages. Primarily, it means that the circuit that processes the TLPs can be simple, and need only carry out a small variety of operations. However, as we have demonstrated, it has disadvantages. A lack of specificity about the operations each device performs leads to vulnerabilities, and can even impose artificial performance limitations. For example, recent work by Marinos et al., [54] attempts to minimise memory accesses while processing network data, making use of Intel's *Direct Data IO* (DDIO), which allows certain peripherals to perform DMA directly to last level cache. The necessity of using this technology, and the security vulnerabilities that arise due to DMA, could be prevented by the use of a higher-level, semantic protocol. Such a protocol would be less structured than PCIe, with much information being device specific. Data would be presented as continuous streams, rather than as instructions to modify memory. It may require special cores, able to receive the data at high speed, a reflection of the early channel processors we discuss in Section 2.6, or the use of increased on-device buffering. While a large departure from the design of the IOMMU, the idea makes some intuitive sense. There is no point filling DRAM with data faster than it can be processed by the CPU, or output to the disk or another NIC. Such a scheme would not require the use of an IOMMU in the same way as PCIe, because calculation of any relevant memory addresses would be the job of the core that received the data.

6.12 SUMMARY

We have presented many new techniques that would close some of the vulnerabilities that occur due to DMA attacks, and summarise them in Table 6.1. Each of the different defences that we present has involves different trade offs between security provided, performance overhead, and implementation overhead that require further investigative work to properly understand. Our attacks that do not rely on ATS make use of the *spatial* vulnerability, according to our taxonomy in Section 2.8. To be sure of security against these exploits, one of the protection mechanisms that counter it must be employed.

	Vulnerabilities Closed			Attack Steps Prevented	
	Fragility	*Granularity*	*Impersonation*	*Memory Scanning*	*Function Pointer Overwriting*
Robust Drivers §6.1	✓				
Suspicious Behaviour Detection §6.2			✓	✓	
Cryptographic Handshaking §6.3		✓	✓	✓	✓
Universally Encrypted Memory §6.4					
Per-device Encrypted Memory §6.4			✓	✓	
Finer Granularity Mappings §6.5		✓		✓	✓
Spatial Segration §6.6		✓		✓	✓
Bounce Buffering §6.7		✓		✓	✓
IOASLR §6.8			✓	◐ *	
Symbol Hiding §6.9					
Function Pointer Indirection §6.10					✓

* A good IOASLR should prevent attacks that rely on memory scanning, but in practice this is not necessarily the case.

Table 6.1: Summary of Potential Defences. The taxonomy of identified vulnerabilities is in Section 2.8. We do not show vulnerabilities protected by other, existing defence mechanisms. For example, mitigations against *shared mappings* and *lifetime* vulnerabilities are already found in Linux and FreeBSD. We do not include a row for novel interconnects, as we have not made a specific and detailed recommendation. It is unlikely that the same vulnerability categories would apply: hopefully, they would not.

CONCLUSION 7

Peripherals used to be simple devices, largely concerned with the translation of electrical signals. This is no longer the case. Even cheap peripherals may contain Turing-complete microcontrollers, and many of these can receive firmware updates. Coupled with the increasing presence of Thunderbolt 3, a small external port that allows DMA, it is no longer valid to include peripherals as part of the trusted computing base. They must be viewed as sophisticated adversaries. Protection from malicious peripherals is not a solved problem. We have demonstrated attacks against the best protections provided by Windows, macOS, Linux and FreeBSD. Some of these arise from policy decisions on the part of the developers, others from flaws in the algorithms used to operate the IOMMU. The IOMMU does not lend itself to the creation of protection mechanisms against DMA attacks, having been designed more to allow high-performance allocation of peripherals to virtual machines.

7.1 CONTRIBUTIONS

A Survey of Current Attacks and Protection

We have explored the literature of attacks that work in the presence of an IOMMU and have conducted a thorough survey of IOMMU use across a variety of modern operating systems. We have found that, by default, the use of the IOMMU is disabled in modern versions of Windows, and Linux and FreeBSD, and in many system firmwares. Of the operating systems we examined, only macOS enables the IOMMU by default.

We have found that there has been no systematic attempt to conduct attacks in the presence of an IOMMU. Many existing attacks target vulnerabilities that are not fundamental, including a time gap between system booting and the IOMMU being enabled, and early versions of Intel's VT-d IOMMU failing to remap interrupts. Some attacks target the use of the IOMMU within a virtual machine based system, rather than for intra-OS protection. Several attacks are carried out without a thorough understanding of the way in which the targeted OS actually uses the IOMMU [75, 74, 76].

A Platform for Probing PCIe

We have created a complete platform, detailed in Chapter 4, that allows rigorous analysis of PCIe. It is the first platform that runs a complete, conventional OS, supporting a complex software stack that can respond to requests from the host with respectable latency. This allowed us to be the first to truly demonstrate the effects that manipulating higher levels of the system stack can have on its security. We will open source the platform in order to allow the wider community to explore its applications in analysing PCIe correctness and performance. We created a clean, easy-to-use API for the platform, allowing experiments to be constructed quickly.

A Series of Novel Attacks

We have carried out attacks using vulnerabilities that have not previously been exploited:

ATS While ATS can improve performance, Linux's approach to it allows the IOMMU to be easily bypassed. The default policy of enabling ATS for any device that displays the relevant capability, whether or not real devices of that type support ATS is a severe vulnerability.

Granularity The drivers and device subsystems of the operating systems we examined were not written with IOMMU use in mind, and as a result, the IOMMU has been retrofitted to existing APIs. This has proven particularly damaging when combined with the 4KiB granularity of IOMMU mappings, and has allowed us to subvert control flow via a function pointer which was not deliberately made available to the attack device.

Shared and Lifetime Mappings In order to straightforwardly use the IOMMU with the network subsystem, macOS creates mappings for all mbufs, which are open for the lifetime of the system. This means that all data that exists in the mbuf page can be inspected and read by any device on the system. Tied with IPsec's use of in-place encryption and decryption, this allows a malicious peripheral to read plaintext.

Leak of a Kernel Symbol to a Device MacOS's KASLR would be enough to prevent the control flow hijacking attack if it worked as expected. Due to the lack of the clear threat model regarding devices, a small offset to a known symbol is leaked, allowing the slide to be reverse engineered, and the attack carried out.

7.2 RECOMMENDATIONS

Based on our analysis of IOMMU attacks carried out over Thunderbolt, we recommend a number of a measures that should be taken:

1 *IOMMU protection turned on by default.* This includes at the levels of firmware and operating system. Testing is vital in order to ensure that the system works properly. While this would carry a performance penalty, the evidence shows that this would only apply to users of very high performance infrastructure, rather than the ordinary users who most need protection.

2 *Better ATS policy.* Linux's policy of enabled ATS for any device that displays the capability leads to severe vulnerabilities. It should instead enable it only for devices with a driver that reports support for the feature, and then only when explicitly enabled by a system administrator.

3 *Improvements to OS support for the IOMMU.* Further work needs to be carried out to establish the most efficient system for using the IOMMU. We believe that with current IOMMU designs, a software usage pattern that sets up mappings for the lifetime of the system, and allocates relevant data structures inside those mappings may provide the best performance-security trade off. However, there are still many aspects of this algorithm and design that need further research.

4 *Improvements to IOMMU design.* It may be that it is not possible to achieve full security at acceptable performance with current IOMMU designs. The designs presented in Section 6.5 may alleviate this, although do not necessarily increase IOMMU performance. They at least should not negatively impact it, and would improve performance with limited software changes, as they only need the IOMMU driver to change, rather than peripheral drivers, or other parts of the OS stack. More radical designs, like the RIOMMU discussed in Section 3.2 may be necessary to achieve acceptable performance and security.

7.3 FUTURE WORK

A variety of research is possible to extend on the results we present here across a wide range of time frames. We have already suggested some further research in the previous work, but this is short-term mitigation research.

On the exploit side, there is scope to use the vulnerabilities we present here to attack virtual machine based systems. As this is an area that widely uses the IOMMU, this is worth carrying out, even though virtual machines are mostly deployed in servers without easy Thunderbolt access. Rather than using a malicious peripheral, it may be possible to carry out similar attacks by using a malicious driver for a benign peripheral, as seen in some attacks discussed in Section 2.11.

In the medium term, there are still attacks that can be carried out against systems that employ byte-granularity per-device mappings. These include ARP poisoning and webpage hijacking attacks from false NICs. There is also work to be done in a more thorough explanation of subverting device firmware: how may devices have vulnerabilities that allow their firmware to be changed remotely?

There are also questions arising about the influence of user-mode devices on protection, as seen with, for example, the Netmap framework [71]. A number of questions arise if we move to a world where each process has its own virtualised device interface. The include how the user process is supposed to make use of the IOMMU, and how likely user mode processes are to fall victim to user mode equivalents of the attacks we present here. At present, however, it seems unlikely that this will be a common use case for consumers as current hardware use patterns can provide good-enough performance for tasks that users seem interested in. On the other hand, this may be self-referential, and it could be that users are interested in this because that's what there is the hardware support for them to do.

In the long term, our protocols for interacting with devices need to be recreated with security as a first-order concern. A more channel-based and semantic approach would bring a variety of benefits. Rather than the device communicating by writing to an area of memory, it could explicitly send a message to the host that says 'this is the data for a packet', and the host could take care of where that data ends up being placed. It is hard to see how to handle this sort of approach while still keeping the performance of DMA, but, of course, research exists to answer hard questions.

GLOSSARY

ACPI Advanced Configuration and Power Interface. Open standard that operating systems can use to discover and configure system hardware, manage power states, and monitor system state.

ACS Access Control Services. PCIe specification providing security enhancements. See Section 2.3.

AES Advanced Encryption Standard. Symmetric-key cipher standard, adopted by the U.S. Government, and believed to be secure. It should be noted that this does not preclude vulnerabilities due to improper usage or implementation.

AGP Accelerated Graphics Port. Standard for a channel to connect a graphics card to a system. It has now been phased out in favour of PCIe. See Section 2.6.

AHCI Advanced Host Controller Interface. Specification for the operation of SATA host bus adapters.

ALM Adaptive Logic Module. Intel term for the basic building-block of an FPGA. See Section 4.2.

AMD Advanced Micro Devices. Designer and manufacturer of x86-compatible processors.

AMD-V AMD's term for their virtualisation features, including their IOMMU.

APU Accelerated Processing Unit. AMD's term for an SOC containing a CPU, cache, and GPU on the same chip.

ARM Ltd Formerly 'Advanced RISC Machines Ltd', but now the company is simply ARM Ltd. Designer of the most widely-used RISC instruction set, and processors implementing it.

ARP Address Resolution Protocol. Network protocol for discovering the link-layer (for example, Ethernet) address associated with a particular IPV4 address.

ASLR Address Space Layout Randomisation. Family of protection mechanisms that attempt to thwart ROP attacks by placing code and data at randomised locations in the system address space. See Section 2.7.

ATS Address Translation Services. PCIe specification allowing peripherals to translate their own IOVAs, bypassing the IOMMU. See Section 2.3.

BAR Base Address Register. Registers in the configuration space of every PCIe device that are written to in order to specify areas of address space that are allocated to that device. See Section 2.3.

BERI Bluespec Extensible RISC Implementation. CPU used for attack platform. See Section 4.2.

BIOS Basic Input/Output System. Firmware used to perform hardware initialisation, and provide limited runtime services. In modern PCs, deprecated in favour of UEFI.

BSV Bluespec System Verilog. Modern hardware description language.

CDC Control Data Corporation. Now-defunct manufacturer of mainframes and supercomputers.

CHERI Capability Hardware Enchanced RISC Instructions. Instruction set and implementation of a RISC processor, using capabilities to enhance performance and security of sandboxed applications.

CFG Control Flow Graph. A graph representing valid execution sequences of a program. See Section 2.7.

CFI Control Flow Integrity. Protection mechanism that aims to prevent control-flow attacks by ensuring that all branches a program takes are to a valid target. See Section 2.7.

CPI Code Pointer Integrity. Protection mechanism that aims to prevent ROP attacks by preventing changes to code pointers. See Section 6.10.

CPU Central Processing Unit. We use this to mean the processor that is responsible for running the OS. This is probably the 'most trusted' component of the system, because if it is not running faithfully executing the code provided for it, no security guarantees can be made. In our definition, the CPU may have multiple cores or threads. In systems with multiple processors, the definition spans the multiple processors.

CSV Command-separated Values. Very basic file format for tabular data. Each row of the data appears on a new line. Columns are separated by commas.

CTSRD Clean Slate Trustworthy Secure Research and Development. Research project revisiting the hardware-software interface to improve security. Responsible for the CHERI and BERI processors.

DDIO Data Direct I/O. Intel feature that allows NICs to DMA directly into processor cache.

DEV Device Exclusion Vector. AMD feature allowing a limited amount of PCIe isolation. See Section 2.6.

DHCP Dynamic Host Configuration Protocol. Protocol allowing a host to be automatically allocated an IP address, and furnished with related configuration information.

DMA Direct Memory Access. Permission for a peripheral to access host memory directly, without needing the CPU to transfer data. May also refer to a specific request from a peripheral.

DMAR DMA Remapping. Feature provided by an IOMMU allowing an IOVA to be translated into a physical memory request, or rejected.

DOS Denial of Service. Class of attack where an attacker prevents a service being used by legitimate users. This is often achieved by overwhelming the service with spurious requests.

DRAM Dynamic Random-Access Memory. Type of memory that stores each bit in a tiny capacitor. Used as PC 'main memory.'

DRHU DMA Remapping Hardware Unit. The component of an Intel IOMMU that carries out the remapping operation.

DWord Double Word. PCIe datatype 32-bits in size.

EAL Evaluation Assurance Level. Specification of the depth and rigour of an evaluation carried out to determine adherence to the 'Common Criteria for Information Technology Security Evaluation', an international standard for computer security security specification.

EEPROM Electrically Erasable Programmable Read-Only Memory. Read-only memory that can be erased and reprogrammed after application of an electrical signal. Commonly used to store firmware.

FPGA Field Programmable Gate Array. Integrated circuit that can be programmed to emulate the operation of other circuits. Largely made up of small RAM modules that can be programmed to embody simple binary functions, connected by a reprogrammable interconnect. See Chapter 4.

GART Graphics Address Remapping Table. IOMMU specified as part of AGP to simplify scatter-gather memory access. See Section 2.6.

GiB Gibibyte. 2^{30} bytes.

GDB GNU project DeBugger. Debugging tool, allowing a running program to be paused in response to certain conditions, and for its state to be investigated.

GPU Graphics Processor Unit. Or 'graphics card', a processor that is highly specialised for rendering 3D graphics. Over time, these have been become more programmable and general purpose, leading to the introduction of GPGPUs (*General Purpose GPUs*), which are capable of achieving high performance when executing a variety of applications that make heavy use of floating-point operations.

GUID Globally Unique Identifier. 128-bit number used as an identifier. Generated with a method that ensures that the chance of identical identifiers for different items is negligible.

IC Integrated Circuit. Electronic circuit implemented on a silicon wafer, allowing billions of components in a space less than 1000mm^2.

ICMP Internet Control Message Protocol. Protocol used by network devices to transmit operational information, including errors. The `ping` program, used to test network availability, uses the ICMP protocol.

IEEE Institute of Electrical and Electronics Engineers. Professional association responsible for many computer standards.

IO Input/Output. Parts of a computer responsible for receiving (input) and transmitting (output) data between a user and the system, or between systems.

IOASLR Input/Output Address Space Layout Randomisation. Security technique that randomises where a IO device appears in system address space. See Section 6.8.

IOMMU Input/Output Memory Management Unit. Device responsible for controlling and configuring the access an IO device has to system memory. See Section 2.6.

IOTLB Input/Output Translation Lookaside Buffer. Cache for recently-used IOVA translations. See Section 2.6.

IOVA Input/Output Virtual Address. The address that a particular IO device uses to access a region of system. This will be translated by an IOMMU into a physical memory address.

IP Internet Protocol. The communication protocol that allows datagrams to be transmitted between networks.

IPC InterProcess Communication. Mechanisms and protocols provided by an OS that allow running protocols to communicate between themselves.

IPSec Internet Protocol Security. A network protocol suite that authenticates and encrypts data packets.

KASLR Kernel Address Space Layout Randomisation. Randomising the location the kernel appears at in the system address space in order to make ROP attacks harder. See Section 2.7.

KiB Kibibyte. 2^{10} bytes.

LAN Local Area Network. A network that links devices generally within a radius of 1km.

MAC Media Access Control. The part of a network protocol that controls access to a shared medium, like a radio frequency or a shared bus.

MBR Master Boot Record. A type of sector at the start of a computer mass storage device that contains information about how the disks are partitioned, and executable code for a bootloader. The partition table of an MBR limits disk size to 2 TiB. As disks that exceed this size are beginning to appear, the MBR is being replaced with the GUID partition table scheme.

MDI Management Data Interface. The interface on the Intel 82574L NIC to the physical layer. See Section 5.2.

MHz Unit of frequency. A million cycles per second.

MiB Mibibyte. 2^{20} bytes.

MIPS Formerly 'Microprocessor without Interlocked Pipeline Stages': now a brand. Family of RISC instruction sets.

MMU Memory Management Unit. Translates from process-specific virtual addresses to physical addresses, allowing multiple processes to execute on a processor without knowledge of other running processes.

MSI Message Signaled Interrupts. Modern interrupt mechanism in PCI and PCIe. See Section 2.3.

MSI-X Not defined as an acronym. Extension to MSI that still works in the same fundamental way. See Section 2.3.

NIC Network Interface Controller. IO device that connects a system to a network. See Section 2.5.

NMI Non-Maskable Interrupt. Hardware interrupt that normal techniques for allowing interrupts to be ignored do not apply to. Usually used to report non-recoverable hardware errors.

OS Operating System. Program that provides a virtualised, uniform view of a computer to a number of running processes. It will provides abstractions of the hardware in the system to its running programs.

OSI Open Systems Interconnection. Model of the components of a network as seven stacked layers, each consuming services of the layer below, and providing services to the layer above.

PASID Process Address Space ID. Extension to PCIe. By tagging transaction with a PASID, a PCIe device can be shared by multiple processes (or guest OSs) while providing each process with a complete 64-bit virtual address space.

PC Personal Computer. Computer built around processor and interconnect standards that are direct successors to those found in the 1980's IBM PC. We use this definition in order to draw a distinction from, for example, phones and tablets.

PHP Recursive acronym. 'PHP Hypertext Preprocessor'. Widely-used programming language for server-side programming.

PHY Shorthand for a component that handles the physical layer in a NIC.

PPP Point-to-point protocol. A network protocol used to establish a direct connection between nodes, most famous for its use in dial-up internet access.

QEMU It is unclear if this is an acronym[1]. Full system emulator for a variety of architectures.

QWord PCIe data type of 64-bytes.

RAM Random Access Memory. Any memory technology where any memory location can be accessed in approximately equal time. Colloquially used as shorthand for system main memory.

ROP Return Oriented Programming. Exploit technique that circumvents protections against running code in a data section. See Section 2.7.

RISC Reduced Instruction Set Computing. Instruction set design philosophy that focusses on maximising instruction throughput, working as a good output for compilers.

RS-232 Recommended Standard 232. A standard for transmission of data, commonly used in computer serial ports. On PCs it is largely obsolete, in favour of more modern standards like USB.

SATA Serial ATA. ATA refers generically to mass storage connections, for historical reasons. SATA is a modern standard for this.

SD Secure Digital. Flash memory card format.

SGX Software Guard Extensions. Intel technology to protect certain pieces of code from disclosure or modification. See Section 6.4.

SLIRP Not an abbreviation. Originally a program that emulated a PPP internet connection over a shell account; now used as part of the QEMU network stack. See Section 4.4.

SMMU System Memory Management Unit. ARM MMU specification that includes support for IO remapping.

SOC System-on-Chip. An single IC that includes multiple components of a full system.

SODIMM Small Outline Dual In-line Memory Module. Specification for computer memory modules.

SPEC Standard Performance Evaluation Corporation. Set of widely-used benchmarks in computer architecture.

TAP Network Tap. Virtual network device that interacts at the link layer to send or receive, for example, Ethernet frames.

TCB Trusted Computing Base. The components of a system that have to be trusted

[1] The official website does not seem to say anything on the matter; Wikipedia thinks 'Quick EMUlator', as do some third-party sites, but they may have just taken this from Wikipedia. At least one site says that the original developer has not said anything about it.

in order to fulfil security guarentees.

TCP Transmission Control Protocol. Internet protocol that provides a reliable byte-stream when layered on top of IP.

TLB Translation Lookaside Buffer. Cache for translations from process virtual addresses to physical addresses.

TLP Transaction Layer Packet. Conceptual unit of PCIe data transfer.

TXT Trusted Execution Technology. Intel technology that attempts to mitigate hypervisor attacks, firmware attacks and root kits.

UDP User Datagram Protocol. Thin layer over IP that provides unreliable, packet-based networking.

UEFI Unified Extensible Firmware Interface. Specification defining an interface between an OS and system firmware. Successor to the BIOS.

USB Universal Serial Bus. Standard defining cable, connectors and protocols for computer peripherals.

USB-C Port that supports the USB protocol, but also Thunderbolt, Display Port, and power transmission. See 2.4.

VBS Virtualization-based Security. Microsoft feature that used hardware virtualisation features to create a secure enclave.

VGA Video Graphics Array. Connector carrying analog video, usually used for computer displays. Mostly, but not entirely, replaced in favour of more modern standards.

VM Virtual Machine. An emulation of a computer system.

VMM Virtual Machine Monitor. Alternative term for hypervisor. Computer technology for running virtual machines. May be type-1, running on bare metal or type-2, running on conventional OS.

VT-d Intel Virtualization Technology for Directed IO. Intel technology for virtualising IO devices. Include Intel's IO specification.

BIBLIOGRAPHY

[1] *250+ New Features*. Apple Inc. 2011. URL: `https://web.archive.org/web/20110606235315/http://www.apple.com/macosx/whats-new/features.html#security` (visited on 08/08/2017).

[2] 7-cpu. *Intel Skylake*. 20th July 2017. URL: `http://www.7-cpu.com/cpu/Skylake.html` (visited on 05/01/2018).

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson and Jay Ligatti. "Control-flow Integrity". In: CCS '05. ACM, 2005. (Visited on 29/01/2018).

[4] *About the security content of OS X Lion v10.7.2 and Security Update 2011-006*. Apple Inc. 20th Jan. 2017. URL: `https://support.apple.com/en-gb/HT202348` (visited on 25/05/2017).

[5] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich and Ronen Zohar. *Breakthrough AES Performance with Intel AES New instructions*. Whitepaper. June 2010. URL: `https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf` (visited on 04/01/2018).

[6] *AMD FX Series Processors*. Advanced Micro Devices. URL: `http://www.amd.com/en-us/products/processors/desktop/fx` (visited on 19/06/2017).

[7] *AMD64 Architecture Programmer's Manual. Volume 2: System Programming*. Advanced Micro Devices. Dec. 2016. URL: `http://developer.amd.com/resources/developer-guides-manuals/` (visited on 05/01/2018).

[8] Gene M. Amdahl, Gerrit A. Blaauw and Frederick P. Brooks. "Architecture of the IBM System/360". In: *IBM Journal of Research and Development* 8.2 (1964), pp. 87–101. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5392210` (visited on 26/06/2014).

[9] Nadav Amit, Muli Ben-Yehuda and Ben-Ami Yassour. "IOMMU: Strategies for Mitigating the IOTLB Bottleneck". In: *Proceedings of the Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2010)*. June 2010.

[10] *An Introduction to the Intel® QuickPath Interconnect*. Whitepaper. Intel Corporation, Jan. 2009. URL: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html (visited on 27/09/2017).

[11] Michael Becher, Maximillian Dornseif and Christian N Klein. "FireWire: all your memory are belong to us". In: *Proceedings of CanSecWest*. 2005. URL: https://cansecwest.com/core05/2005-firewire-cansecwest.pdf (visited on 05/01/2018).

[12] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer and Leendert van Doorn. "The Price of Safety: Evaluating IOMMU Performance". In: *OLS '07: The 2007 Ottawa Linux Symposium*. July 2007, pp. 9–20.

[13] Adam Boileau. "Hit by a Bus: Physical Access Attacks with Firewire". In: *Ruxcon 2006*. May 2006. URL: https://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf (visited on 05/01/2018).

[14] Jeff Bonwick and Jonathan Adams. "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources." In: *USENIX Annual Technical Conference*. 2nd Sept. 2002, pp. 15–33. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix01/bonwick.html (visited on 06/02/2018).

[15] Diego Callega. *Linux 2.6.12*. Kernel Newbies. 17th July 2017. URL: https://kernelnewbies.org/Linux_2_6_12 (visited on 08/08/2017).

[16] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. 2014. URL: http://dl.acm.org/citation.cfm?id=2671225.2671250 (visited on 29/01/2018).

[17] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham and Marcel Winandy. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 559–572.

[18] David Chisnall, Colin Rothwell, Brooks Davis, Robert N.M. Watson, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann and Michael Roe. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, Istanbul (ASPLOS 2015)*. Mar. 2015.

[19] Control Data Corporation. *Control Data 6400/6500/6600 Computer Systems: FORTRAN Extended Reference Manual*. 1969. URL: `Control%20Data%206400/6500/6600%20Computer%20Systems:%20FORTRAN%20Extended%20Reference%20Manual` (visited on 06/02/2018).

[20] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86. URL: `https://eprint.iacr.org/2016/086.pdf` (visited on 05/01/2018).

[21] Crispin Cowan, F. Wagle, Calton Pu, Steve Beattie and Jonathan Walpole. "Buffer overflows: Attacks and defenses for the vulnerability of the decade". In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*. Vol. 2. IEEE. 2000, pp. 119–129.

[22] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann and Fabian Monrose. *Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection*. 2014.

[23] Solar Designer. *Getting around non-executable stack (and fix)*. Aug. 1997. URL: `http://seclists.org/bugtraq/1997/Aug/63` (visited on 18/07/2017).

[24] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon and Olivier Levillain. "Can you still trust your network card?" In: *CanSecWest* (2010). URL: `https://www.ssi.gouv.fr/uploads/IMG/pdf/csw-trustnetworkcard.pdf` (visited on 05/01/2018).

[25] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard and Hamed Okhravi. "Missing the Point(er): On the Effectiveness of Code

Pointer Integrity". In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 781–796.

[26] Ulf Frisk. "Direct Memory Attack the KERNEL". In: *Proceedings of DEFCON'24*. Aug. 2016. URL: `https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Ulf-Frisk-Direct-Memory-Attack-the-Kernel.pdf` (visited on 05/01/2018).

[27] *GeForce NOW System Requirements*. NVIDIA LTD. 2018. URL: `https://www.nvidia.co.uk/geforce/products/geforce-now/mac-pc/system-reqs/` (visited on 01/02/2018).

[28] Cristiano Giuffrida, Anton Kuijsten and Andrew S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. 2012. URL: `http://dl.acm.org/citation.cfm?id=2362793.2362833`.

[29] Pavel Gladyshev and Afrah Almansoori. "Reliable acquisition of RAM dumps from Intel-based Apple Mac computers over FireWire". In: *Digital Forensics and Cyber Crime*. 2011, pp. 55–64.

[30] E. Göktas, E. Athanasopoulos, H. Bos and G. Portokalidis. "Out of Control: Overcoming Control-Flow Integrity". In: *2014 IEEE Symposium on Security and Privacy*. May 2014.

[31] Siobhan Gorman. "Fraud Ring Funnels Data From Cards to Pakistan". In: *The Wall Street Journal* (11th Oct. 2008). URL: `https://www.wsj.com/articles/SB122366999999723871` (visited on 26/09/2017).

[32] James Greene. *Intel® Trusted Execution Technology*. Whitepaper. Intel Corporation, 2010. URL: `https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html` (visited on 04/01/2018).

[33] Matthew P Grosvenor. "uvNIC: rapid prototyping network interface controller device drivers". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, pp. 307–308.

[34] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum and Edward W. Felten. "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[35] Norm Hardy. "The Confused Deputy:(or why capabilities might have been invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[36] Michael Henson and Stephen Taylor. "Memory Encryption: A Survey of Existing Techniques". In: *ACM Computing Surveys* 46.4 (Mar. 2014), 53:1–53:26.

[37] Michael Howard, Matt Miller, John Lambert and Matt Thomlinson. *Windows ISV Software Security Defenses*. Microsoft Corporation. Dec. 2010. URL: `https://msdn.microsoft.com/en-us/library/bb430720.aspx` (visited on 08/08/2017).

[38] Ralf Hund, Carsten Willems and Thorsten Holz. "Practical timing side channel attacks against kernel space ASLR". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 191–205.

[39] "IEEE Standard for a High Performance Serial Bus". In: *IEEE Std 1394-1995* (1996).

[40] *Intel® Virtualization Technology for Directed I/O Architecture Specification*. Intel Corporation. Nov. 2017. URL: `https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf` (visited on 04/01/2018).

[41] *Intel's New Mesh Architecture: The 'Superhighway' of the Data Center*. Intel Corporation. 15th June 2017. URL: `https://itpeernetwork.intel.com/intel-mesh-architecture-data-center/` (visited on 27/09/2017).

[42] Gamma International. "Tactical IT Intrusion Portfolio: FINFIREWIRE". In: (2011). URL: `https://wikileaks.org/spyfiles/files/0/293_GAMMA-201110-FinFireWire.pdf` (visited on 05/01/2018).

[43] David Kaplan, Jeremy Powell and Tom Woller. *AMD Memory Encryption*. Whitepaper. Advanced Micro Devices Inc, 21st Apr. 2016. URL: `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf` (visited on 04/01/2018).

[44] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu and Peng Ning. "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software". In: *Proceedings of the 22nd Annual Computer Security Applications Conference*. 2006, pp. 339–348. URL: `http://dx.doi.org/10.1109/ACSAC.2006.9` (visited on 06/02/2018).

[45] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News*. IEEE Press. 2014.

[46] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar and Dawn Song. "Code-Pointer Integrity". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 147–163.

[47] Michael Larabel. *Linux Distributions vs. BSDs With netperf & iperf3 Network Performance*. Phoronix. 7th Dec. 2016. URL: `http://www.phoronix.com/scan.php?page=article%5C&item=netperf-bsd-linux` (visited on 20/07/2017).

[48] Brian Lich. *Introduction to Device Guard: virtualization-based security and code integrity policies*. Microsoft Corporation. 5th Apr. 2017. URL: `https://docs.microsoft.com/en-gb/windows/device-security/device-guard/introduction-to-device-guard-virtualization-based-security-and-code-integrity-policies` (visited on 09/08/2017).

[49] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim and Wenke Lee. "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 280–291. URL: `http://doi.acm.org/10.1145/2810103.2813694` (visited on 06/02/2018).

[50] Carsten Maartmann-Moe. *Inception*. 2011. URL: `http://www.breaknenter.org/projects/inception/` (visited on 05/01/2018).

[51] *Mac OS X has you covered.* Apple Inc. 2011. URL: `https://web.archive.org/web/20110525190329/http://www.apple.com/macosx/security/#sixtyfour` (visited on 08/08/2017).

[52] Moshe Malka, Nadav Amit, Muli Ben-Yehuda and Dan Tsafrir. "rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015, pp. 355–368. URL: `http://doi.acm.org/10.1145/2694344.2694355` (visited on 06/02/2018).

[53] Moshe Malka, Nadav Amit and Dan Tsafrir. "Efficient Intra-Operating System Protection Against Harmful DMAs". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Feb. 2015, pp. 29–44. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/malka` (visited on 06/02/2018).

[54] Ilias Marinos, Robert N.M. Watson, Mark Handley and Randall R. Stewart. "Disk, Crypt, Net: Rethinking the Stack for High-performance Video Streaming". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 211–224. URL: `http://doi.acm.org/10.1145/3098822.3098844` (visited on 06/02/2018).

[55] Alex Markuze, Adam Morrison and Dan Tsafrir. "True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 2016, pp. 249–262.

[56] John McDonald. *Defeating Solaris/SPARC Non-Executable Stack Protection*. Mar. 1999. URL: `http://www.ouah.org/non-exec-stack-sol.html` (visited on 18/07/2017).

[57] Robert M. Metcalfe and David R. Boggs. "Ethernet: Distributed packet switching for local computer networks". In: *Communications of the ACM* 19.7 (1976), pp. 395–404. URL: `https://dl.acm.org/citation.cfm?id=360253` (visited on 04/01/2018).

[58] *Microsoft Windows Kernel Address Space Layout Randomization Security Bypass Vulnerability*. Cisco. 13th Aug. 2013. URL: `https://tools.cisco.com/security/center/viewAlert.x?alertId=30296` (visited on 08/08/2017).

[59] Benoît Morgan, Guillaume Averlant, Éric Alata and Vincent Nicomette. "Bypassing DMA remapping with DMA". In: *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications*. June 2016.

[60] Benoît Morgan, Eric Alata, Vincent Nicomette and Mohamed Kaâniche. "Bypassing IOMMU Protection against I/O Attacks". In: *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. Oct. 2016, pp. 145–150.

[61] Roger M Needham. "Systems aspects of The Cambridge Ring". In: *Proceedings of the Seventh ACM symposium on Operating Systems Principles*. ACM. 1979, pp. 82–85.

[62] Nergal. *The advanced return-into-lib(c) exploits: PaX case study*. Dec. 2001. URL: `http://phrack.org/issues/58/4.html` (visited on 18/07/2017).

[63] *OS X Mountain Lion Core Technologies Overview*. Apple Inc. June 2012. URL: `http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf` (visited on 08/08/2017).

[64] *PC OEM requirements for Device Guard and Credential Guard*. Microsoft Corporation. 2nd May 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/design/minimum/device-guard-and-credential-guard` (visited on 09/08/2017).

[65] PCI-SIG. *PCI Express Base Specification Revision 3.0*. Nov. 2010.

[66] Allison M. Pearce. "Exploring I/OMMU Vulnerabilities in Direct Memory Access Attacks". MPhil Thesis. University of Cambridge, 2015.

[67] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon and Christoph Neumann. "On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. 2014, pp. 305–316. URL: `http://doi.acm.org/10.1145/2590296.2590299` (visited on 06/02/2018).

[68] Omer Peleg, Adam Morrison, Benjamin Serebrin and Dan Tsafrir. "Utilizing the IOMMU Scalably". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. July 2015, pp. 549–562.

[69] David R. Piegdon and L. Pimenidis. "Hacking in physically addressable memory". In: *Proceedings of 4th International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment, DIMVA*. 2007.

[70] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack." In: *USENIX Security Symposium*. 2016.

[71] Luigi Rizzo. "Netmap: a novel framework for fast packet I/O". In: *21st USENIX Security Symposium*. 2012, pp. 101–112.

[72] Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* (2012).

[73] Henry Samuel. "Chip and pin scam 'has netted millions from British shoppers'". In: *The Telegraph* (10th Oct. 2008). URL: `http://www.telegraph.co.uk/news/uknews/law-and-order/3173346/Chip-and-pin-scam-has-netted-millions-from-British-shoppers.html` (visited on 26/09/2017).

[74] Fernand Lone Sang, Éric Lacombe, Vincent Nicomette and Yves Deswarte. "Analyse de l'ecacité du service fourni par une IOMMU". In: *Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications*. June 2010.

[75] Fernand Lone Sang, Éric Lacombe, Vincent Nicomette and Yves Deswarte. "Exploiting an I/OMMU vulnerability". In: *Proceedings of 5th International Conference on Malicious and Unwanted Software (MALWARE), 2010*. Oct. 2010, pp. 7–14.

[76] Fernand Lone Sang, Vincent Nicomette and Yves Deswarte. "A Tool to Analyze Potential I/O Attacks against PCs". In: *IEEE Security & Privacy* 12.2 (Mar. 2014), pp. 60–66.

[77] Bruce Schneier. *Sony's DRM Rootkit: The Real Story*. 17th Nov. 2005. URL: `https://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html` (visited on 25/08/2017).

[78] Russ Sevinsky. "Funderbolt: Adventures in Thunderbolt DMA Attacks". In: *Black Hat USA* (2013). URL: `https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf` (visited on 06/02/2018).

[79] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 552–561.

[80] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu and Dan Boneh. "On the effectiveness of address-space randomization". In: *Proceedings of the 11th ACM conference on Computer and communications security*. ACM. 2004, pp. 298–307.

[81] J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, W. Zwaenepoel and P. Willmann. "Concurrent Direct Network Access for Virtual Machine Monitors". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Feb. 2007, pp. 306–317.

[82] The Linux Kernel Team. *Linux KConfig*. July 2017.

[83] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein and Michael Bailey. "Users Really Do Plug in USB Drives They Find". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). May 2016, pp. 306–319.

[84] Fujita Tomonori. "DMA Representations: SG_table vs SG_ring IOMMUs and LLD's Restrictions". In: *Linux Storage and Filesystem Workshop*. Feb. 2008. URL: `https://www.usenix.org/legacy/publications/login/2008-06/openpdfs/lsf08reports.pdf` (visited on 19/06/2017).

[85] Jim Turley. *Introduction to Intel(R) Architecture*. Whitepaper. Intel Corporation, Jan. 2014. URL: `https://www.intel.com/content/www/us/en/intelligent-systems/embedded-systems-training/ia-introduction-basics-paper.html` (visited on 27/09/2017).

[86] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi and Cristiano Giuffrida. "Drammer: Deterministic rowhammer attacks on mobile platforms". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[87] Robert N. M. Watson, David Chisnall, Brooks Davis, Wojciech Koszek, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann and Jonathan Woodruff. *Bluespec Extensible RISC Implementation: BERI Software reference (UCAM-CL-TR-853)*. Tech. rep. University of Cambridge, Apr. 2015. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-853.pdf (visited on 06/02/2018).

[88] Robert N. M. Watson, Robert Norton, Jonathan Woodruff, Alexandre Joannou, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Michael Roe, Colin Rothwell, Stacey Son and Munraj Vadera. "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture". In: *IEEE Micro* 36.5 (Sept. 2016), pp. 38–49.

[89] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton and Michael Roe. *Bluespec Extensible RISC Implementation: BERI Hardware reference (UCAM-CL-TR-852)*. Tech. rep. University of Cambridge, Apr. 2015. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-852.pdf (visited on 04/01/2018).

[90] Paul Willmann, Scott Rixner and Alan L. Cox. "Protection Strategies for Direct Access to Virtualized I/O Devices". In: *USENIX 2008 Annual Technical Conference*. 2008, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1404014.1404016 (visited on 06/02/2018).

[91] Rafal Wojtczuk. *Defeating Solar Designer's Non-executable Stack Patch*. Jan. 1998. URL: http://insecure.org/sploits/non-executable.stack.problems.html (visited on 18/07/2017).

[92] Rafal Wojtczuk and Joanna Rutkowska. *Following the White Rabbit: Software attacks against Intel VT-d technology*. 2011. URL: http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf (visited on 05/01/2018).

[93] Rafal Wojtczuk, Joanna Rutkowska and Alexander Tereshkin. "Another way to circumvent Intel trusted execution technology". In: *Invisible Things Lab* (2009). URL: http://soft-club.ucoz.ru/newsfail/metalnews/intelvpro/Another_TXT_Attack.pdf (visited on 04/01/2018).

[94] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song and Wei Zou. "Practical Control Flow Integrity and Randomization for Binary Executables". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. 2013. URL: http://dx.doi.org/10.1109/SP.2013.44 (visited on 29/01/2018).

[95] Mingwei Zhang and R. Sekar. "Control Flow Integrity for COTS Binaries". In: *Proceedings of the 22Nd USENIX Conference on Security*. 2013. URL: http://dl.acm.org/citation.cfm?id=2534766.2534796 (visited on 29/01/2018).

[96] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel and Mark Silberstein. "Understanding The Security of Discrete GPUs". In: *Proceedings of the General Purpose GPUs*. 2017, pp. 1–11. URL: http://doi.acm.org/10.1145/3038228.3038233 (visited on 06/02/2018).

LIST OF FIGURES

2.1 Interconnect Protocols in a Single Processor Intel System 20

2.2 PCIe ID Bit Layout 26

2.3 ATS Extended Capability Bit Layout 29

2.4 Descriptor Ring Diagram 32

2.5 Intel VT-d Page Table Diagram 36

4.1 Photograph of macOS experimental setup 67

4.2 Photograph of Windows, Linux and FreeBSD experimental setup . . 68

4.3 Attack Platform Diagram 69

4.4 Intel PCIe Hardcore Header and Data Alignment 74

4.5 `RawTLP` definition 75

4.6 Request Transaction Bit Layout and Corresponding Struct 76

4.7 Postgres field access macro 85

4.8 Code to construct TLP. 86

5.1 MacOS Mbuf Definition 92

5.2 `struct iopa_t` Definition 95

5.3 Code for Adding an ATS Capability to a NIC Model 98

6.1 Page Table Mapping with Base and Bounds 111

LIST OF TABLES

2.1 Meanings of the AT field in a PCIe header. 29
2.2 Taxonomy of Vulnerabilities. 43
2.3 Summary of published DMA attacks. 51
3.1 Performance Improvement Provided by [53] 62
3.2 IOMMU Usage in Current OSs 66
3.3 IOMMU-based Mitigation Research 66
5.1 Fields Common to Every MacOS Mbuf. 92
5.2 Header Data in MacOS Mbufs with an External Payload. 93
5.3 Summary of our novel attacks. 103
6.1 Potential Defences 118