Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Using the IOMMU for Safe and Secure User Space Network Drivers

Stefan Huber

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Using the IOMMU for Safe and Secure User Space Network Drivers

Nutzen der IOMMU für sichere User Space Netzwerktreiber

Author:	Stefan Huber
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich, M. Sc.
Date:	March 26, 2019

Abstract

Commonly used user space network drivers such as DPDK or Snabb currently have effectively full access to the main memory via the unrestricted Direct Memory Access (DMA) capabilities of the PCI Express (PCIe) device they are controlling. This can be a security issue, as the driver can use the PCIe devices DMA access to read and / or write to main memory . But malicious hardware, or hardware with malicious or corrupted firmware can be an even bigger security risk, as most devices and their firmware are closed source. Attacks with malicious NICs have been shown as a proof of concept in the past. All modern CPUs feature an I/O Memory Management Unit (IOMMU) as part of their virtualization capabilities: It is required to pass PCIe devices through to virtual machines and is currently used almost exclusively for that. But it can also be used to restrict memory access of DMA devices, thus reducing the risk of malicious or simply badly implemented devices and code.

In this thesis, support for using the IOMMU via the `vfio-pci` driver from the Linux kernel for the user space network driver `ixy` was implemented in C and Rust and the IOMMU and its impact on the drivers were investigated. In the course of this, a model of the IOMMU on the investigated servers was developed to enable the usage of it in further work and other drivers, as well as minimize the performance impact from using it. Reasonable specifications of the IOMMU that are not widely known or documented, as the number of page entries in the IOMMU's IOTLB or its insufficiently documented capability of using huge pages for memory management were found and used. Additionally, the C library `libixy-vfio` was developed to make it easy to use the IOMMU in any driver. When properly implemented, i.e. using 2 MiB hugepages and putting all NICs in the same IOMMU container, using the IOMMU has no significant impact on the performance of the ixy driver in C or Rust and does not introduce any latency to most packets, but effectively isolates the NICs and restricts access to memory effectively. Since the performance impact is negligible and the security risk when not using the IOMMU is high, using it should always be a priority for researchers and developers when writing all kind of drivers. Using the library `libixy-vfio` or following the patches for `ixy` or `ixy.rs`, implementing usage of the IOMMU is simple, safe and secure for user space network drivers.

Contents

List of Figures

Chapter 1

Introduction

User space drivers for NICs have several advantages over the kernel drivers. For specific tasks, like simple forwarding or packet generation, there is a speed improvement, since the packets do not have to traverse the kernel from the NIC to the application. Kernel drivers follow the kernel coding rules and use a lot of code and macros that can be complicated and thus badly readable. They implement every feature of a NIC when often only a very small subset is really needed, whereas user space drivers can implement only exactly what they need to be small and readable, as for example ixy [12] does. Since most widely used kernels (Windows, Mac OS, Linux) are written in C, most kernel drivers are too. This can be cumbersome and, since writing correct C code is hard, often leads to memory management problems [8], sometimes resulting in the fatal `segfault` error and often leading to severe security risks [29]. User space drivers like ixy.rs [11] solve this problem by using a "safe" language for their implementation of the driver, reducing the risk of such mistakes to a bare minimum, while only sacrificing little to no speed [46].

One other, not so-often discussed disadvantage of kernel drivers is that they automatically run with root privileges, making them inherently unsafe and potentially dangerous. Unfortunately, most user space still need to run as root, since the device files do not belong to any user other than root, or the driver needs to make use of huge pages or mlock capabilities, which, by default, also need root privileges. While all of these problems might be solvable on their own, the `vfio-pci` driver [26] for Linux solves them all at once in a very elegant way, exposing exactly one file for each device bound to it which can be `chown`ed to the non-privileged user the driver will run as.

But the most useful feature of the IOMMU for user space drivers is that it can control the memory regions a PCIe device can access. By default, PCIe devices can unrestrictedly access the whole system memory and other PCIe devices memory via DMA, reading as well as writing. Since most drivers are extremely hard to read and thus most computer users do not know what their NIC driver does, it would be very simple to attack a user with a malware driver that conveniently also has inherent internet access. And even if the malware is in the firmware itself, virtually impossible for users to detect, DMA access enables the misbehaving firmware to read

and write all system memory. As Markettos et al. proved in their *Thunderclap* paper, this is not just a theoretical risk, but computers can actively be attacked by malicious hardware [28]. The IOMMU can restrict the access of the device to specific memory regions, protecting the user from malicious firmware, and the programmers of said user space drivers from accidentally overwriting important memory regions, e.g., their file system.

Currently, the IOMMU is mostly used for virtualization, passing through PCIe devices like NICs to virtual machines in professional environments (e.g., server hosting) or graphics cards in consumer environments (*GPU passthrough*) [1]. While some user space drivers use the IOMMU for some level of isolation [10], others dropped the support for it again or have never begun to implement it [19]. Also, most operating systems do not enable the IOMMU by default, or even have support for doing so [21, 39, 28].

How big are the changes to a driver that add IOMMU usage for better security? What is the performance impact of using the IOMMU, and is it feasible to use it? And what are the non-published specifications of the IOMMU on a modern CPU? In this thesis, the usage of the IOMMU will be implemented in the educational user space network drivers `ixy` and `ixy.rs`. Additionally, a C library for easy extension of drivers to use the IOMMU will be written. Using the ixy driver, by changing various variables, the IOMMU will be investigated and new key data about it estimated. It will be shown how to implement the fundamental changes to device and memory management, and the method will be explained in detail. The performance of the IOMMU enabled driver will be measured with `ixy-fwd` and `MoonGen` [14] based on packet forwarding capacity and latency. In the course of this, the IOMMU of the investigated CPU will be modeled and the optimal use case and constants for its constrains researched and explained.

Chapter 2

Background

2.1 User Space Network Drivers

User Space Network Drivers, like the *Data Plane Development Kit* (*DPDK*) [9], *Snabb* [40] and *ixy* [12] try to circumvent some of the stated problems with kernel drivers, like high latency and low performance by minimizing the number of kernel code and syscalls needed. As a result, they can provide a higher level of security, as they are isolated from the kernel space and could be run by a non-privileged user. They mostly are highly specialized. DPDK is the largest user space driver kit with most applications. It can be used for software packet routing (with Open vSwitch [34]), dumping packets (tcpdump-like [9]) and most other Network Functions Virtualization (NFV) [17] applications. MoonGen can be used for packet generation and ixy has exactly two applications right now, namely packet forwarding and packet generation. Additionally, most user space network drivers only work on a very limited set of NICs, with DPDK supporting about 30 NICs at the time of writing. With kernel drivers, any application in the user space can use any NIC that has a kernel driver available (the kernel contains currently over 1000 different ethernet network drivers [45]), but has to use syscalls for every interaction with the hardware, thus adding layers of function calls and privilege separation and impacting performance.

Most network drivers are implemented very similar at a ground level [18], see Figure 2.1 for an overview of how ixy works. Most modern NICs have the option to use multiple *queues* for receiving (RX) and sending (TX) packets, thus the ability to balance load on multiple CPUs (or CPU cores). The driver can enable either one, several or all hardware queues. For each such enabled hardware queue, some kind of queue structure is generated in the main memory. The NIC of course needs memory access to these regions, thus there exists a mapping from the virtual memory address of this queue (for the driver) to the physical memory address (for the hardware). Since actual packet data is large, and the queue needs to be accessed regularly (to check if a new packet was received or needs to be sent), most NICs don't save actual packet data

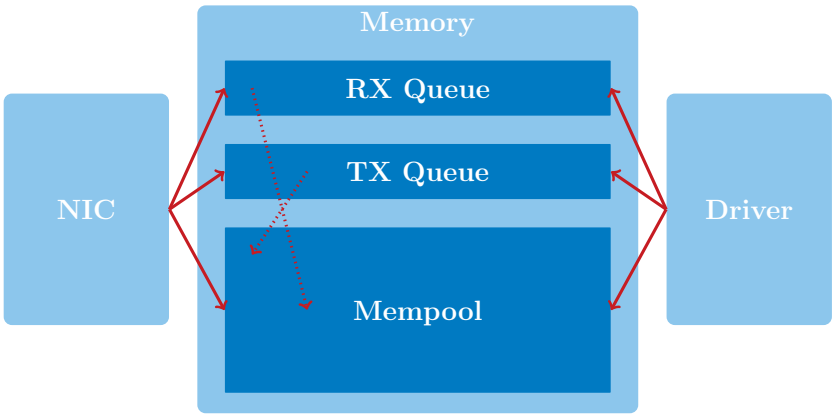

Figure 2.1: Model of the user space network driver ixy. Every NIC hardware queue (only one depicted here) features one RX and TX queue in the main memory. Every RX and TX queue entry point to one packet buffer in the mempool (dotted lines). Both the NIC and the the driver application have to access both queues and the mempool regularly (solid lines).

in these queues, but only packet metadata, or *descriptors*. The packet data itself is then stored in a separate memory space. Depending on the implementation, the driver allocates the memory for each packet individually (which produces overhead for mapping and unmapping each packet memory space), or allocates only a single, large memory space often called *mempool*. Each entry in every packet descriptor queue must then be given a fixed *packet buffer* address from this mempool, and the NIC needs access to this mempool, too, again needing a mapping of each packet in virtual (driver) memory space to physical space. Depending on the implementation, a mempool is allocated for every NIC or for every driver. NICs normally have unrestricted memory access to the main memory via DMA, thus access to each queue and mempool is guaranteed. This means, that, if the address given to the NIC in a packet descriptor in one of the queues is wrong, the NIC will write to or read from unwanted memory space, with the ability to destroy important information. The same of course is true if the NIC is given wrong memory addresses for the queues.

The device is set up by issuing certain Peripheral Component Interconnect (PCI) commands to certain device files, e.g., in Linux the files `config`, and `resource0` and other Base Address Registers (BAR) files are located in `/sys/bus/pci/devices/$PCI_ADDRESS/`. These commands differ with every NIC, but must always contain the address to the queues. Most devices also have other commands, like device resets, enabling and disabling of queues, or for certain cosmetic abilities like turning the LEDs of a NIC on and off. Other commands that are issued via these files might tell the NIC at which place in a queue to place new packets, telling the device that new packets have been put in a TX queue, or simply to stop transmitting or receiving packets.

On the application side, the RX queues must be read and cleaned (*received*) regularly or after an Interrupt Request (IRQ) from the NIC, i.e., for each received packet, the descriptor entry must be given a new packet buffer from the mempool. The received packet must then be processed. In the case of a forwarding application, the packet in the mempool must be edited (e.g., the TTL must be decreased), the destination in a forwarding table found and then added to a (possibly

different) NIC's TX queue, i.e., the packet buffer memory address written to an empty TX queue descriptor. Other applications might process the packet in other ways, e.g., simply discarding the packet, displaying its content or dumping it to hard disk memory. Sending a packet happens similar. The application needs to allocate a packet buffer in the mempool for the packet, write the contents into the buffer and set the buffer address to an empty TX queue entry. Whenever the driver tries to *send* a packet, but the TX queue is full (i.e., the NIC has not sent any packets from the queue), it has to decide if the packet is getting queued in another, internal queue (increasing packet delay), deleting unsent packets from the TX queue and substituting them with new packets, or dropping the packet to be sent (both resulting in packet loss).

All driver operations normally happen in so called *batches*, i.e., the driver will receive a certain amount (the *batch size*) of packets at once, process and send packets and clean queues (clearing hardware flags or assigning new packet buffers) in batches. Since accessing main memory is relatively slow, a small batch size will bottleneck the application. E.g., for the ixy forwarding application, a batch size of at least 32 will not bottleneck the application [12] for sufficiently fast CPUs. It is worth noting, that the usage of IRQs adds additional packet delay, thus simple user space drivers like ixy run the above mentioned steps (batched receiving, processing and sending) in a continuous loop and thus fully occupying one CPU core. Using IRQs, the CPU usage could be decreased, but this would also increase the packet delay due to the added overhead of processing IRQs.

2.2 The (IO)MMU

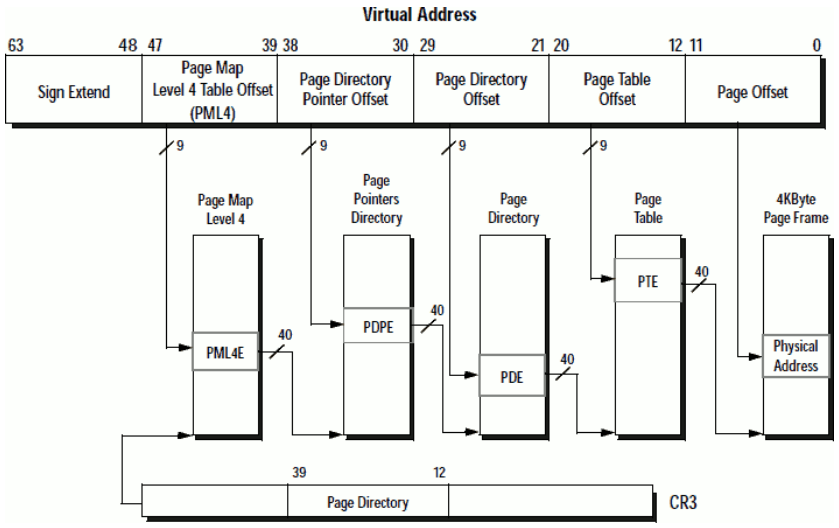

Figure 2.2: 4-Level memory page translation in x86-64 CPUs. Page mapping happens via four 9 Bit wide layered page tables. From *O. Lawlor, 2012 CS 301 - Assembly Language* [25].

The *MMU* in modern CPU architectures is part of the CPU itself, though it started as an external component, e.g., the Motorola 68851 in 1984. Its main purpose is to translate virtual memory

addresses that are given to any process requesting memory to physical memory addresses, i.e., it knows where a processes virtual memory is stored in the physical domain. This has some advantages. Processes are kind of backward-compatible (i.e., 32-bit processes can make use of memory beyond the 32-bit barrier at 4 GiB) when using virtual memory. They are also isolated by the MMU. Every process gets his own virtual memory, such that non-privileged processes can not access memory that was not explicitly assigned to them.

To address physical memory, the MMU works with so-called *pages*. On most x86-64 processors, the standard page size is 4 KiB, i.e., every 4 KiB requested by a process might be contiguous in virtual address space, but separated in physical address space. The locality of each virtual page to its physical page is saved in a translation table in the main memory. As seen in Figure 2.2, the translation of these virtual to physical address spaces happens over several *levels* (in case of 4 KiB pages there are 4 levels [7] on modern CPUs, though 5 levels are already implemented in the Linux kernel [6]).

The *Translation Lookaside Buffer (TLB)* is a fast cache that saves some of these translation table entries. In modern CPUs, a typical MMU TLB can hold up to 4096 of these 4 KiB page entries [3], accelerating access to up to $4096 \times 4\,\text{KiB} = 16\,\text{MiB}$.

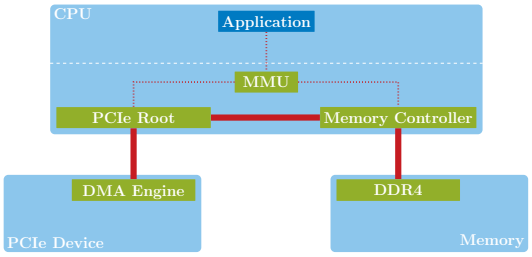

Figure 2.3: Every PCI(e) device has *unrestricted* memory access via DMA ...
Image source: [15]

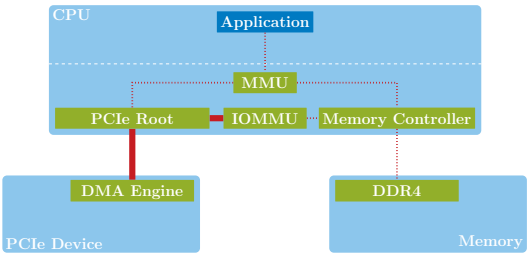

Figure 2.4: ... but not with the IOMMU enabled!

The *IOMMU* on the other hand does for I/O devices (hence the name), what the MMU does for processes. It can control a PCI(e) devices' memory access and maps physical memory to I/O Virtual Addresses (IOVA), which is called DMA Remapping (DMAR). The IOMMU descends from the Graphics Address Remapping Table (GART), which was originally used by graphics cards. An IOMMU was first implemented in Sun's UltraSPARC processors PCI interface as "Device Virtual Memory Access" [42], AMD implemented its version of an IOMMU as "AMD I/O Virtualization Technology (IOMMU)" [2] (and gave the IOMMU its name), and Intel implemented its own version of it as "Intel Virtual Technology for Directed I/O (VT-d)" [23]. In modern CPUs, the IOMMU is part of the CPU itself, like the MMU. Other IOMMU implementations exist for other architectures and manufacturers, but in this thesis Intel's VT-d will be investigated because the hardware is readily available and probably used more often in server environments than AMD's processors. For the sake of simplicity, this thesis will always speak of *IOMMU*, even when Intel's VD-t is actually in use. Nevertheless, most findings should be applicable to at least AMD's very similar IOMMU, and probably other IOMMU implementations. An IOMMU also features paging (e.g., the Intel VT-d features a two-level paging [23]),

and has its own TLB, called *IOTLB*. There are no official data on how many entries make up the IOTLB, though.

2.2.1 Huge Pages

Since modern computers have much more than 16 MiB of main memory, and modern processes need much more of it, accessing main memory gets very inefficient when many accesses to large spaces of main memory are made (as in network drivers), since the TLB cache runs out of enough entries quickly. For comparison, the unmodified *ixy* driver uses 8 MiB of main memory for every NICs mempool (which stores the actual sent / received packets) and two additional 8 KiB queues (RX and TX), adding up to 16.032 MiB main memory for the simple forwarder application. These 16 MiB need to be accessed regularly and thus the forwarder needs all of the MMUs TLB entries (and even gets some misses).

For this, larger page sizes have been introduced. Most modern systems feature 2 MiB pages, and even 1 GiB pages are supported by newer CPUs, all of them called *huge pages*. Each such entry only uses one entry in the (IO)TLB. The only hardware restriction is, that these pages must then be contiguous in physical memory. With this approach, much more memory can be accessed in a very fast way, as the (IO)MMU does not need to load the translation table entries for many small pages from the main memory and traverse all the (IO)MMU page levels, but instead can read relatively few entries from the fast (IO)TLB cache. E.g., the *ixy forwarder*, only needs 12 TLB entries when using 2 MiB huge pages, removing all TLB cache misses, thus making the driver faster and reducing package latency [12].

The *Intel Virtualization Technology for Directed I/O Architecture Specification* [23] states that Intel's IOMMU features page sizes of 4 KiB, 2 MiB and 1 GiB, but the number of entries in its IOTLB is not discussed in that document. Neugebauer et al. concluded from their tests, that the IOTLB of one of their test systems (Intel Xeon processors of the Ivy Bridge to Broadwell generation) has 64 entries [33].

2.2.2 Usage of the IOMMU in Other Operating Systems

Other operating systems than the one investigated in this thesis (Linux) might write their own drivers for the IOMMU. *Microsoft Windows*, with the exception of Windows 10 Enterprise, does not support using the IOMMU, and even in Windows 10 Enterprise, it is not enabled by default. In fact it is only possible to use it when both UEFI boot and secure boot are enabled. And even then, devices share mappings, i.e., every device can read every other devices memory, and up to build 1803, the host operating system was not fully protected. Limited protection was implemented in build 1803, though [28]. *Mac OS*, on the other hand, seems to be the only widely used operating system that by default enables the IOMMU, thus protecting the user from malicious devices [28].

Especially for microkernel and unikernel operating systems using the IOMMU seems interesting, as they aim to be very secure. *Minix*, the free microkernel operating system, has a driver for

AMD's IOMMU since at least 2011, but does not use it by default for any kind of device [43]. *Singularity*, the discontinued microkernel operating system from Microsoft, apparently recognized the problem in 2005 already, but did never receive an IOMMU driver [21]. *MirageOS* and *IncludeOS*, unikernel operating systems, rely on the virtualization technology of the host to isolate devices. Since those unikernel operating systems are mostly compiled for exactly one purpose, the need for IOMMU is very small as every OS has a very limited number of PCI devices. Still, it would be good to have the additional security to be protected from malicious devices. *Redox*, the free Rust operating system, does not currently have an IOMMU driver [37]. Also *Biscuit*, a POSIX kernel written in the Go language, does not implement an IOMMU driver right now [35]. The *seL4* microkernel also does not have an IOMMU driver currently, but the developers are working on it [39]. Both big free virtualization technologies *KVM* [24] and *XEN* [48] have options to use the IOMMU, but that is only on a per-VM based isolation of PCI devices, so a VM can not read other VMs' or the host's memory. The guest operating system is then in charge of separating its devices itself, e.g., via a virtual IOMMU exposed by the virtualization solution [5].

2.2.3 Usage of the IOMMU in Other Projects

DPDK, the user space network driver framework, implements usage of isolated network interfaces, when VT-d is activated in the host kernel and BIOS, and the VFIO driver is used for NICs [10]. So, when having the IOMMU enabled in the BIOS and in the operating system, it will be used to restrict devices, though all devices will belong to the same container, thus not isolating them from each other and giving them full access to each other's DMA memory. Still, the DPDK framework can not be run as non-root user, since access to the address translation from virtual address space to physical address space needs root permissions under Linux, and this is needed for most drivers. *Snabb*, another user space network driver framework, does currently support the IOMMU only if the host system has the IOMMU enabled in "passthrough" mode, i.e., enabled, but not operating at all. An issue to make Snabb work with enabled IOMMU exists, though [19].

2.2.4 IOMMU in Linux: VFIO

In Linux, the usage of the IOMMU is implemented in the `vfio-pci` driver. VFIO originally meant "**V**irtual **F**unction **I/O**", but since this name was meaningless at best and misleading at worst, other names have been found: "**V**ery **F**ast **I/O**" or "**V**irtual **F**abric **I/O**". The most fitting name would be "**V**ersatile **F**ramework for user space **I/O**", though [47] .

Have a look at Figure 2.5 for the following explanation of the partition into IOMMU devices, groups and containers. In respect to the IOMMU, a *function* of a PCI(e) device (e.g., a single interface on a NIC, a single USB controller of an USB controller card, or the audio or graphics controller of a graphics card) is an *IOMMU device*. Mind the difference between PCI(e) device and IOMMU devices: One might call a single PCIe add-in card a "device", but for the IOMMU,

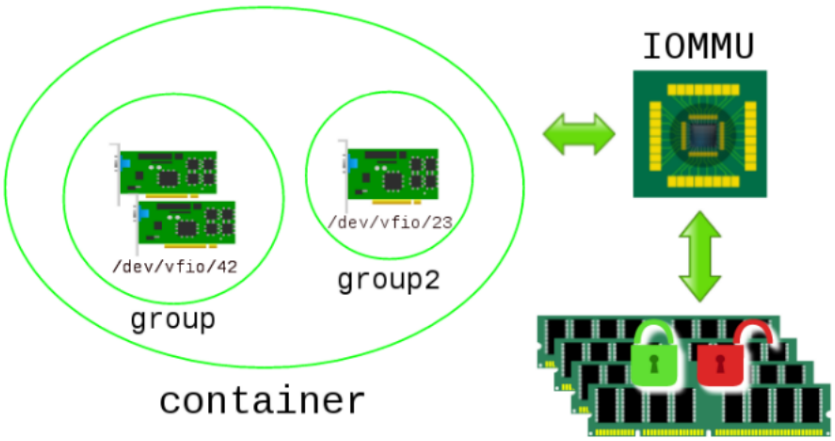

Figure 2.5: Overview over the grouping of PCI devices into groups and containers, and the way they access main memory. From *An Introduction to PCI Device Assignment with VFIO* [47].

this card might be several "IOMMU devices", e.g., a PCIe card might have two network interfaces, but every one of those interfaces is a single "IOMMU device". IOMMU devices (= hardware functions) that can not be further separated on hardware level, e.g., the audio and graphics chip on a graphics card, form an *iommu group*. Thus, an IOMMU group might contain one or more IOMMU devices. It is possible, that one single PCI device with two functions (e.g., a NIC with two interfaces) adds two or more IOMMU groups (one for every interface). The exact grouping of hardware into VFIO groups can change with hardware (e.g., a different mainboard), firmware (e.g., mainboard BIOS) or the version of the `vfio-pci` driver, where most of the times newer versions will give a finer isolation of devices. With some mainboards, even several PCIe cards might be grouped together in regards of the IOMMU. This means that for optimal performance a cutting edge firmware, BIOS and kernel are necessary.

Though each of the IOMMU groups can be separated by the IOMMU, this is sometimes not desirable. In a simple forwarding application of a network driver, both NICs have to access the other NIC's mempool. For such scenarios, one or more IOMMU groups can be put in *IOMMU containers*. IOMMU memory is mapped to containers, such that every device in every group of one container can access all the memory that was mapped to this container, but is fully isolated from other containers and the rest of the system main memory.

The `vfio-pci` driver maps IOMMU devices, groups and containers to their software counterparts, VFIO devices, groups and containers [47]. Thus, full control over how to group devices (as far as the hardware allows) and put them into containers is given to a user by the `vfio` driver.

Chapter 3

Related Work

Of course, a lot of literate was published on the IOMMU, and the comparison of performance between different languages, compilers and interpreters. This chapter will give a short overview over some of the work that is related to this thesis.

3.1 Related Work on the IOMMU

Since Oracle applied for its "Device Virtual Memory Access" patent in 1993 [44], the isolation of DMA devices has found concern in the scientific community. In 1996, when the patent was granted and the IOMMU implemented in the SPARC architecture, Linux was ported to SPARC, with full usage of the new IOMMU [22]. But only with AMD's own implementation of the IOMMU in 2007 [2] and Intel's implementation of VT-d in 2011, IOMMUs became widely available on many devices, starting a real flood of publications, dealing with breaking, fixing, improving and, finally, actually using the IOMMU.

3.1.1 Impact of the IOMMU on PCIe Performance

At Sigcomm '18, a group of researchers presented their findings on the impact on NICs through the PCIe bus and various of its features, including the IOMMU [33]. Using Netronome and NetFPGA NICs, they wrote a PCIe benchmarking suite called `pcie-bench` that measures among other things bandwidth and latency of the PCIe bus, and were able to compare different architectures and processors. Neugebauer et al. were also able to investigate the impact for Non-Uniform Memory Access (NUMA) systems and the IOMMU. Though PCIe apparently does not bottleneck 10 GBit/s NICs (with exceptions), apparently the bus itself must be considered as a possible bottleneck for NIC performance. In that work, the impact of the IOMMU int IOTLB was also investigated, concluding a IOTLB size of 64 entries. Additionally, they found that the effective bandwidth of a PCIe 3.0 device drops as far as 70 percent, depending on the PCIe bus packet

size. This could be circumvented by using two network cards on separate PCIe lanes, but for 40 GBit/s NICs more than four PCIe lanes would be needed for further improvement.

3.1.2 DMA Attacks

Markettos et al. wrote an article called *Thunderclap* [28] in which they describe a hardware attack on hosts exposing PCIe via Thunderbolt, that can be mitigated in parts by using the IOMMU in the host system. By programming an FPGA that was attached to the host's Thunderbolt via Universal Serial Bus (USB) C, such that it registered as an Intel 82574L Gigabit Ethernet Controller to the host system - a standard gigabit ethernet adapter that can be connected to hosts via USB 3. It then acted as an ethernet adapter, sending and receiving packets through the attached ethernet port to the host system, and thus was almost indistinguishable from a normal adapter. It used its DMA access though, to read system memory for a proof of concept. It was found, that many operating systems do not protect from such attacks by default, as described in Chapter 2.

Stewin et al. wrote in 2012 the DMA malware *DAGGER* (DmA based keystroke loGGER) [41]. It was written for the *Intel Management Engine (IME)*. The IME has full DMA access to the main memory via the *Management Engine Interface (MEI)* PCI interface. The DMA malware DAGGER was implemented in the IME Firmware and it uses its DMA capabilities to read system memory and find the keyboard buffer address in the memory of the host operating systems, Linux and Windows. The malware infiltrates the host systems during runtime with a known bug in the IME, and then logs keyboard input of the user and sends the information via the IME's network connection (an out-of-band network connection that uses the same physical hardware than the host operating system). Using the IOMMU can mitigate this malware attack, since the IME is not able to use DMA to read memory when isolated. It still can use the network interface, though, and as such is an inherent security risk in all unpatched systems.

3.1.3 Circumventing the IOMMU as an Attacker

Just by using the IOMMU, a certain basic protection can be assumed, as long as it is implemented and set up correctly. In 2010, Sang et al. analyzed two attacks on the IOMMU [38]. The first attack was set against a misconfigured Intel VT-d, showing that it is hard to correctly configure the IOMMU and thus be protected from DMA attacks. For this attack, either hardware access is needed to the machine to disable the IOMMU all together, (e.g., in the unprotected Basic Input/Output System (BIOS)), or root access is needed to set up the kernel to not use the IOMMU or group several devices together. The second attack was set against a correctly set up IOMMU, abusing a second I/O controller to perform attacks on the user. In this attack, a machine with known IOMMU grouping (see Chapter 2), can be attacked. When two PCIe devices are in the same IOMMU group, they can still read each other's memory. In the case of the attack described in the paper, an USB ethernet card was connected to the same PCI bus as a FireWire device, thus both shared DMA capabilities and were not isolated from each other. A malevolent FireWire device (a modified iPod) was used to inject malicious ethernet frames

to the USB Network Interface Card (NIC). The attacker then could use ARP cache poisoning to intercept all traffic between the attacked host and any other machine, when in the same network. Generally, the FireWire and Thunderbolt connections are prone to such attacks, since they expose DMA access to the attached device.

In 2016, based on the DAGGER attacks, Morgan et al. used a bug in the IOMMU and the Linux kernel to bypass IOMMU protection [31]. By circumventing the IOMMU very early in the system boot process (it was demonstrated during the GRUB bootloader), a modified FPGA device was able to inject a rootkit kernel module into the main memory, that was executed later and provided the attacker with physical access to the device effectively full root access.

3.1.4 IOMMU Limitations and Their Mitigation

As stated earlier, the IOMMU features an IOTLB that caches recently accessed pages. When many DMA accesses are made in short time, the IOTLB needs to be refreshed, and pages that are no longer in the cache have to be read from the system main memory, by walking the page table. This can seriously impact performance. Amit et al. wrote in *IOMMU: Strategies for Mitigating the IOTLB Bottleneck* about how the IOTLB bottleneck could be mitigated, and proposed hard- and software solutions for that [4]. They begin with stating that single-use mapping of DMA memory is unfeasible and increases CPU load, thus multiple-use mappings should be created that are not unmapped after each access. This would decrease security only minimally, since the device could read all its mapped memory all the time. Several devices were investigated in regard to their DMA access patterns and different IOTLB miss rates. In their investigated CPU (an Intel Xeon X5570 on a X58 chipset), the IOTLB seems to have 32 entries. To investigate the access patterns and find hardware fixes for the IOMMU, they implemented the *vIOMMU* [5], a virtual IOMMU that can also be used by virtual machines that do not have an own IOMMU. Based on the access patterns they recognized with several kinds of hardware, e.g., network cards and SCSI disk controllers, they found ways to mitigate the IOTLB bottleneck. One of their main proposals is the increase of pages the IOTLB can hold, and the use of huge pages. Additionally, they propose prefetching of adjacent IOTLB entries and explicit caching of newly mapped pages. With their proposed changes, they were able to reduce the number of IOTLB misses by up to two thirds, depending on the application.

Malka et al. designed a ring IOMMU (rIOMMU) [27] to mitigate some performance impacts of the IOMMU and IOTLB on high performance networking cards and PCIe SSD controllers. Using this could further improve performance for high-throughput 40 GBit/s networking cards, but needs hardware implementation and can not be implemented in software only. The throughput gain is achieved by implementing the IOMMU as a ring buffer, thus minimizing the cost of IOTLB invalidations. Since memory accesses are already queued in a ring queue (RX and TX queue in case of network cards), access to memory spaces also happens in a ring kind of way. When DMA memory is unmapped after every operation, the IOTLB entry has to be invalidated, which is a costly procedure and might thus be stalled, to later invalidate the whole IOTLB at once [27]. Since the ixy driver does currently not unmap DMA addresses, such invalidations do not happen, and thus ixy already is optimized for the use of IOMMU without needing an

rIOMMU. Drivers for other devices, or drivers with a different use case than ixy, might need to unmap DMA memory, though (e.g., solid state drive drivers), and in these cases employing an rIOMMU would improve throughput and reduce latency.

3.2 Performance Comparison of Different Languages

Many attempts have been made to create a reasonable, fair comparison of the different programming languages, compilers and interpreters. Kernels and operating systems exist written in many different languages (Linux, Windows and Mac OS are written in C, Redox OS in Rust, Biscuit and GopherOS in Go, and even Java has its own JavaOS). Ixy right now comes in flavors of eight programming languages, namely the standard implementation in C (`ixy`), `ixy.rs` in Rust, `ixy.go` in go, `ixy.cs` in C#, `ixy.hs` in Haskell, `ixy.swift` in swift, `ixy.ml` in OCaml and `ixy.py` in Python [13]. Since all of the language variants of ixy have been compared to each other [13], a comparison of the languages for other tasks seems reasonable. A short overview over some comparisons between languages, compilers and interpreters will be given here.

In 2000, Prechelt compared C, C++, Java, Perl, Python, Rexx and Tcl with each other, using the Phonecode programming problem [36]. In the Phonecode problem, every letter of the alphabet is given a digit. The algorithm is provided an input file of words, and an input file of telephone numbers, and must match words to the letters of the telephone numbers. In this comparison, C is the fastest implementation, followed by Perl (time factor of 4), Python, Tcl, Java and C+ (all about a factor of 10), and finally Rexx (factor of about 100). The memory footprints of the running programs range from 10 MBytes (C++) to 45 MBytes(Java). Also, the coding time per program has been evaluated, and the length of the program in Lines Of Code (LOC). It is concluded, that C and C++ are comparably fast, and the scripting languages fall behind in performance.

In 2015, the codebase Rosetta Code [30] was used to compare eight widely used programming languages ("procedural: C and Go, object-oriented: C# and Java, functional: F# and Haskell, scripting: Python and Ruby") in *A Comparative Study of Programming Languages in Rosetta Code* [32]. Rosetta Code is a website that features solutions for different problems in as many programming languages as possible. For example, the *Caesar cipher* problem, where each letter in an input is shifted by a certain number of letters in the alphabet, has a solution in 124 different programming languages right now. Investigated were the length of source code, size of executables, runtime and memory performance and susceptibility to failures. Length of code is important, as programmers are prone to produce more errors in more LOC. Python solutions to the investigated programs needed the least LOC, followed by Haskell, Ruby, F#, Java, Go, C and C#. The size of the executable files is mostly irrelevant, since most computers have more than enough main memory and cache to keep most binary code in at least level 3 cache, but it is noteworthy that most languages have a similar compiled size (bytecode), whereas Haskell and C# compile to native code, with a larger executable. Speed results are more interesting: C comes ahead of the other languages, followed by Go. Then Java and F#, C#, Haskell and Python follow in that order, with Ruby needing the most time for most tasks. A similar pattern

can be seen for memory usage of the programs, where the only major difference is that Haskell solutions need more memory than Ruby. Last, the susceptibility to errors is investigated. The strictly typed languages produce less errors in most cases, making Go the most secure of the investigated languages, and Python and Ruby, very loosely typed languages, the most error prone.

In 2019, Felix Leitner called upon the readers of Fefes Blog to send idiomatic or strongly optimized code for a single word processing problem, in which words in an input file should be counted and output in order of their occurrence counts [46]. Solutions for the languages C, Rust, Go, Python, Perl and others have been tested with the source code of llvm 8.0.0 as input file. While the highly optimized C code (compiled with *gcc*) is the fastest competitor, Rust follows (with a working time of about double that of C, but almost 10 times as much memory needed) on second place. C++, Go and java follow C and Rust, with up to 3 times more time needed for the task, and Ruby comes in last taking almost 12 times as long.

With all of the above results and since the ixy application is CPU bound most of the time in case of forwarding 64 Byte packets, it comes to no surprise that the original ixy and ixy.rs implementation feature the best performance.

Chapter 4

Implementation

This chapter will describe how to activate the IOMMU on a machine and how to implement using it in C and Rust.

4.1 Activating the IOMMU

On most systems, the IOMMU is deactivated by default, and must be activated in both the BIOS and the operating system. Exact steps on how to activate the IOMMU in a server's BIOS can not be given here, as too many different mainboards and BIOSes exist, but on Intel CPUs the option is most likely called *VT-d*, and on AMD CPUs *IOMMU*. Most of the time it is essential to also activate all virtualization options (i.e., all other VT options like *VT-x*). The IOMMU must then be enabled for the operating system. To the Linux kernel, the command line argument `intel_iommu=on` must be given for Intel CPUs, and for many distributions (i.e., kernel build configurations), `iommu=force` must also be given, since it is not set by default in all distributions configurations. Only now, devices will be visible in their respective IOMMU groups in `/sys/kernel/iommu_groups`, and the IOMMU can be used.

4.2 IOMMU Grouping and Device Acquisition

As stated in Subsection 2.2.4, IOMMU devices are grouped and then put in containers. Any memory that is mapped to a container with one or more devices will be available by all devices in all groups in this container, but not to any devices in other containers. In a best-case scenario, every device would have its own container, being completely isolated from all other devices.

The usability of a network driver where each device is completely isolated would be limited by raw CPU and memory speed (e.g., packets would have to be copied by the process from the memory space of one NIC to the memory space of the other NIC). Thus, in the IOMMU enabled

implementation ixy [20] all NICs are put in one container, making it possible for each NIC to read the packets in the mempool of any other NIC.

Binding a device to the `vfio-pci` driver in Linux exposes the devices group in `/dev/vfio/$GROUP_NUMBER`. This file can be chowned by any user, thus eliminating the need to be root to communicate with the device. This file combines all devices in one group.

4.3 Every System Programmer's Friend: ioctl(2)

To do useful stuff with the group file, the VFIO syscalls in `ioctl` are needed. `ioctl(2)` is the Linux kernel interface for everything that does not get its own syscall. Since it combines many use cases, it is not that simple to use, syscalls to it are not that easy to read and the return values and `errno`s can be very ambiguous. In the Linux documentation to VFIO [26] and in `linux/vfio.h`, all needed syscalls are described and explained. Luckily, there is an even better introduction to VFIO than the kernel documentation by Alex Williamson [47], a Red Hat developer.

Summarized, the following `ioctl` calls are needed in roughly that order:

```
1  ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);
2  ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);
3  ioctl(container, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU);
4  ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);
5  ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);
6  ioctl(group, VFIO_GROUP_GET_DEVICE_FD, $DEVICE_PCI_ADDR);
7  ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);
8  ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);
9  ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &irq);
10 ioctl(container, VFIO_IOMMU_UNMAP_DMA, &dma_map);
11 ioctl(device, VFIO_DEVICE_RESET);
```

Source code 1: All `ioctl` calls needed for basic IOMMU support

`VFIO_GROUP_GET_STATUS` sets the status to two flags. `VFIO_GROUP_FLAGS_VIABLE` is set to true, when the group is viable, i.e., all devices in the group are either unbound by any Linux driver or bound to the `vfio-pci` driver. `VFIO_GROUP_FLAGS_CONTAINER_SET` is set to true when the group already is in a container. For this driver it is assumed that a group's devices are bound to `vfio-pci` before the driver is started, but as a precaution the viable flag is checked anyways. If a group is not viable, the driver exits with an error message. As the `ixgbe` driver in ixy works only for NICs that are each in their own group, a check for `VFIO_GROUP_FLAGS_CONTAINER_SET` is not needed.
`VFIO_GROUP_SET_CONTAINER` adds a group (in this case with only one device) to the container, that was previously opened with `open("/dev/vfio/vfio", O_RDWR)`.
`VFIO_SET_IOMMU` enables the IOMMU model for this container to Type 1. Other possible types are `VFIO_SPAPR_TCE_IOMMU` for SPARC processors or `VFIO_TYPE1v2_IOMMU` for version 2 of the IOMMU. Since `Type1v2` does not offer any improvements regarding ixy, Type 1 is used to be

compatible with most modern x86-64 CPUs, including Intel and AMD processors.
`VFIO_IOMMU_GET_INFO` then fills a `struct vfio_iommu_type1_info`, which contains information about supported memory sizes. For the ixgbe devices on the investigated servers, the IOMMU supports 4 KiB, 2 MiB and 1 GiB page sizes. Since on all the ixy supported NICs and systems 2 MiB huge pages are supported, this `ioctl` is not used in ixy, and the driver always uses 2 MiB huge pages.
`VFIO_IOMMU_MAP_DMA` is the critical one, it maps memory for the device according to the given `struct vfio_iommu_type1_dma_map`. This also programs the IOMMU to allow I/O memory access for this container only to the mapped region(s). Such mapped regions can be reset later via `VFIO_IOMMU_UNMAP_DMA`, which is not needed in ixy, since this happens automatically when the application is closed again. This also prevents the higher CPU usage by remapping DMA space every other time, as explained in the work by Amit et al. [4].
Up to now, all interactions have been with the group. `VFIO_GROUP_GET_DEVICE_FD` recovers the device file descriptor for following `ioctl` syscalls that need access to the individual device.
`VFIO_DEVICE_GET_INFO` fills a `struct vfio_device_info` with information about several functions supported by the device (via `VFIO_DEVICE_FLAGS_*`), the number of regions the device exposes and the IRQs the device supports. Since the ixgbe NIC is well-known, this call is not needed in ixy.
`VFIO_DEVICE_GET_REGION_INFO` is needed in ixy, as the region info for the `config` region is needed to enable DMA in `vfio_enable_dma` and the `resource0` region to configure the device. The information on the requested region is put in a provided `struct vfio_region_info`.
`VFIO_DEVICE_GET_IRQ_INFO` is not needed in ixy, since ixy does not yet make use of IRQs.
Finally, `VFIO_DEVICE_RESET` resets the device. This syscall is not needed, either, as this happens automatically when exiting the process, and the device is reset by writes to the config region when initializing an ixgbe device.

4.4 LIBIXY-VFIO

The main part of the work regarding VFIO and the IOMMU happens in the newly written `libixy-vfio` library. The `libixy-vfio` header file `libixy-vfio.h` describes all exposed functions:

```
1 int vfio_init(const char* pci_addr);
2 void vfio_enable_dma(int device_fd);
3 uint8_t* vfio_map_region(int vfio_fd, int region_index);
4 uint64_t vfio_map_dma(void* vaddr, uint32_t size);
5 uint64_t vfio_unmap_dma(int fd, uint64_t iova, uint32_t size);
```

Source code 2: libixy.vfio functions exposed in `libixy-vfio.h`

`vfio_init()` initializes the IOMMU for the given device by issuing the above mentioned `ioctl`s after checking the binding of the device to the `vfio-pci` by looking for an existing group file. It returns the device file descriptor that is later needed by other calls to the `libixy-vfio` library.

`vfio_enable_dma()` enables DMA access for the device with file descriptor `device_fd` via the config region exposed by the device file descriptor returned by `vfio_init()`.
`vfio_map_region()` maps one of the device regions, either the config or one of the resource regions, in the memory and returns a pointer to the mmapped region. This is needed in ixy for mapping the config region to enable DMA (see above), or to map the `resource0` BAR, to communicate with the NIC.
`vfio_map_dma()` and `vfio_unmap_dma()` map and unmap DMA-able memory for the device. The unmap function is not used in ixy, as all regions are unmapped when the process exits. `vfio_map_dma()` is the function that actually programs the IOMMU and restricts the device to the mapped memory.

4.5 Changes to ixy

Changes to ixy include adding two fields to the `struct ixy_device` `bool vfio` and `int vfio_fd`. The boolean value `vfio` is needed to check if initialization of the IOMMU is needed. The `int vfio_fd` is the pointer to the aforementioned device file descriptor. It is needed to access the config space and BAR of the device, as the `vfio-pci` driver exposes them in the `/dev/vfio/$GROUP_NUMBER` file as stated earlier. The access is thus different than the regular access via the files in `/sys/bus/pci/devices/$PCI_BUS_NUMBER/`. In the non-IOMMU enabled version of ixy, device configuration happens by mapping `/sys/bus/pci/devices/$PCI_BUS_NUMBER/config` into memory and reading / writing to the mapped memory. In the IOMMU enabled version, the config space is mapped in the `vfio_fd` with a certain offset, which is described in the `struct vfio_region_info` filled by the `VFIO_DEVICE_GET_REGION_INFO` `ioctl` syscall. Same is true for the `resource0` and every other resource space, though only the `config` and `resource0` memory spaces are needed in ixy.

The file `ixgbe.c` had to be changed to determine if VFIO initialization is needed, and initialize it. Also, the `resource0` file has to be `mmap`ped via above mentioned file descriptor `vfio_fd` instead of the standard `/sys/bus/pci/devices/$PCI_BUS_NUMBER/resource0` file. The same two changes had to be made to the VirtIO driver in `virtio.c`.

The `memory.c` file also needed several adjustments. Since the memory mapping for the IOMMU happens on a per-container basis, a global variable for the container file descriptor is needed, which is set once an IOMMU enabled device gets initialized. In accordance with that, the `memory_allocate_dma()` function needs to change the way memory is allocated for VFIO devices, calling `libixy-vfio`'s `vfio_map_dma()` function instead of `mmap`ing huge pages by itself. The memory mapping to the devices I/O Virtual Addresses (IOVA) is chosen to be explicitly simple: Despite the fact that an own memory mapping could be used, the implementation maps the same virtual addresses for the device (IOVA) as for the process (VA).

The full, working implementation into ixy (C) can be found at GitHub [20].

4.6 Implementation Using Rust

Since the Linux kernel is written in C, for any application also written in C (like the original ixy driver) it is relatively easy to implement the needed `ioctl` syscalls, memory mapping and file descriptor handling. Implementing IOMMU support in other languages, however, holds its own problems.

At time of writing this thesis, ixy implementations exist for eight programming languages, namely the standard implementation in C (`ixy`), `ixy.rs` in Rust, `ixy.go` in go, `ixy.cs` in C#, `ixy.hs` in Haskell, `ixy.swift` in swift, `ixy.ml` in OCaml and `ixy.py` in Python [13]. Of these, `ixy.rs` has the best performance (after the original `ixy` implementation), and Rust is a "systems programming language", and such the Rust implementation was chosen to receive an IOMMU upgrade, too.

While the basic steps behind the IOMMU setup are the same as with C (see above), the most implementation intensive part is actually calling `ioctl` in rust. Modules for `ioctl` usage exist (e.g., `nix::sys::ioctl`), but those lack other needed C syscalls (like `pread` or `pwrite`). So, with use of the crate `libc`, IOMMU support was implemented in `ixy.rs`. Some of Rusts most appraised advantages had to be circumvented, though, as C library calls expect pedantic C behavior. Rust supports structs, but the memory layout of a Rust struct is not the same as in a C struct by default, instead the compiler option `#[repr(C)]` must be given if the C memory structure is needed:

```
1  //Rust code                                  //C code
2  #[repr(C)]
3  struct vfio_group_status {                  struct vfio_group_status {
4      argsz: u32,                                 __u32   argsz;
5      flags: u32,                                 __u32   flags;
6  }                                           };
```

Source code 3: Different declaration of structs in C and Rust. Mind the `#[repr(C)]` in the Rust code.

Since most of the VFIO structs from `linux/vfio.h` are used as arguments for the needed `ioctl` calls, all those struct definitions had to be implemented as `#[repr(C)]` structs in ixy.rs.

Rust is designed to be as memory safe as possible, and this is implemented by very good measures: Files, for example, are closed as soon as the program moves out of the context where the file was opened. This is a very good idea, since most programmers are lazy and often forget to `close()` files. That behavior is unwanted, though, if a file handle needs to be held open as long as the program runs, as is the case in the network driver `ixy.rs`. In that case, a global variable has to be defined, which is not only very "un-rust-y", but also `unsafe` (more on that below).

Also, some constants from the C header file `linux/vfio.h` had to be hard coded in `ixy.rs`, as they were not exposed by any rust crate.

4.6.1 Unsafe Code

Rust is a safe programming language, as long as the *Unsafe Superpowers* are not used. These include dereferencing a raw pointer, calling an unsafe function, accessing a mutable static variable, and implementing an unsafe trait. All four of these are needed in the user space network driver `ixy.rs`:
Dereferencing raw pointers is needed, since `ioctl` sometimes only gives back pointers, which have to be dereferenced then.
As stated above, *calling unsafe functions* from the `libc` is needed, which are all together `unsafe`. Most of all, of course, `ioctl` itself is unsafe.
The file descriptor for the container file needs to be *mutable static*, but must accessed at least once per application run.
Lastly, the `RawFD` type for raw file pointers is needed, and `RawFD` *implements an unsafe trait.*

All together, 29 of 552 added lines of code use the `unsafe` keyword. This does not invalidate the use of Rust for usage as a systems programming language, as the basic safeties that Rust guarantees are still given outside of the unsafe scope, e.g., when handling packet memory. By design, above operations need to be unsafe as they can not be checked at compile time. The usage of the `unsafe` keyword clearly marks regions in the code, where basic programming mistakes could still be made. Other mistakes, like logical ones, can still be made outside of the scope of `unsafe` regions, but here the compiler can at least give you some guarantees of not running into memory errors like access violations.

The full, working implementation into ixy.rs can be found at GitHub [11].

Chapter 5

Modeling the IOMMU

All the measurements in this thesis were made on an Intel Xeon E5-2620 v3 with an Intel 82599ES NIC. The load (2×10 GBit/s at 64 Byte packets, i.e., 14.88 Mpps) was generated on a server with an Intel Xeon E5-2630 v4 CPU and another Intel 82599ES NIC with `MoonGen` [14]. Similar results were achieved with Intel X540-2 NICs on both sides. Results may vary with different NICs, processors and even with different mainboards due to different PCIe configuration.

5.1 Performance Impact of 4 KiB Pages

When using the IOMMU was first implemented in this thesis, it was done without use of 2 MiB huge pages, which resulted in a performance drop of up to 75 percent, as can be seen in Figure 5.1. This led to the question, how to optimize for this apparent bottleneck, and for this, it is important to know the number of page entries, the IOTLB can hold. As stated earlier, this

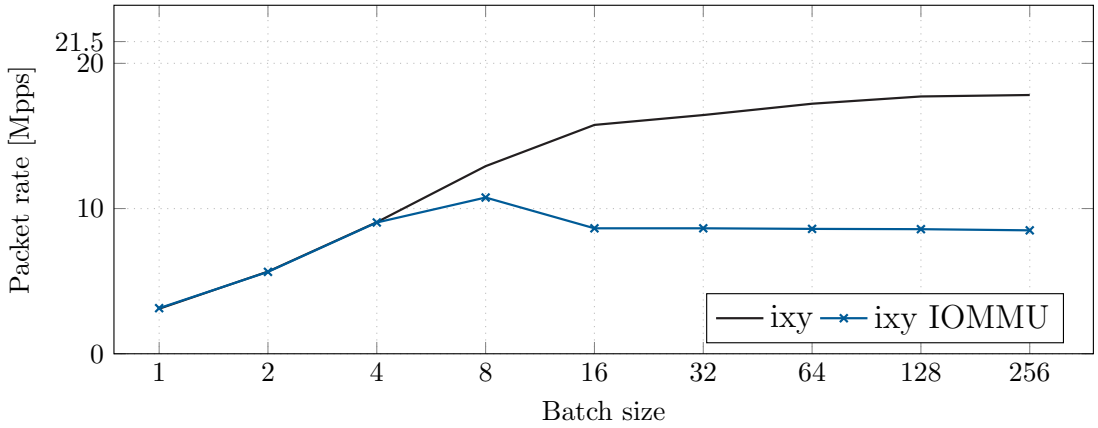

Figure 5.1: Impact of 4 KiB pages on the IOMMU at 1.60 GHz

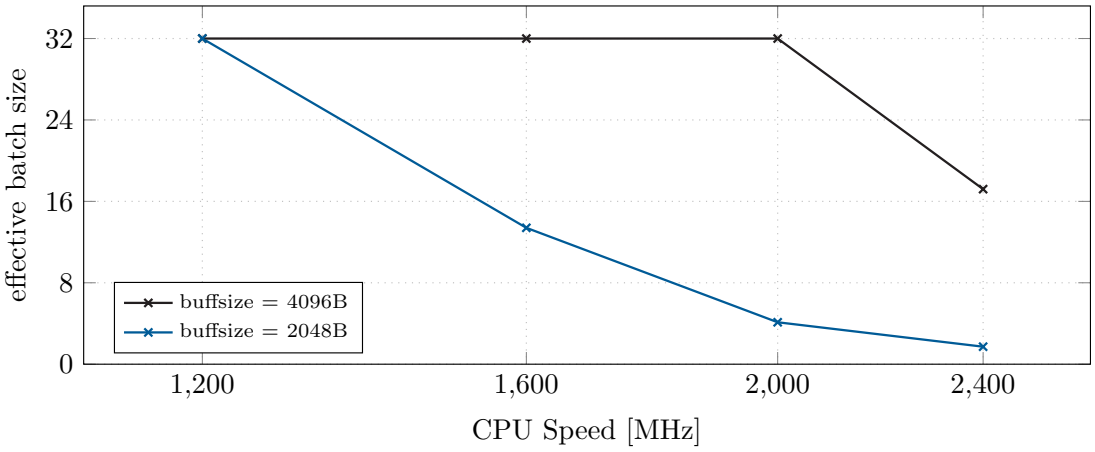

Figure 5.2: Impact of the CPU frequency on the effective batch size (ebs) with 4 KiB pages. ebs is capped at 32 by the driver constant `BATCH_SIZE`.

is not stated in the Intel specifications [23], but because of the severe performance drop, it can safely be assumed that the IOTLB features significantly less entries than the TLB.

The performance impact of the TLB of the MMU when using 4 KiB pages has been investigated by Emmerich et al. in the original ixy implementation and is very small [12]. For comparison with the IOTLB impact, it can be seen by inspecting the Effective Batch Size (ebs), i.e., the actual number of packets the `forward` application reads and writes from and to the RX and TX queues per loop cycle. In the unmodified ixy driver, the (maximum) batch size is set to 32, so 32 packets would be taken from the RX queue of every NIC, the packets touched (so that they go through all levels of the CPU cache), and then written to the RX queue of the other device. If the NICs can write / read more packets to / from the queues than the CPU in a certain amount of time, the ebs will always be this set batch size. Increasing the batch size can help with this issue to a certain point, as shown by Emmerich et al. [16]. This ebs bottlenecking can be observed by intentionally creating a CPU bottleneck, i.e., clocking the CPU at a very low speed. When increasing the CPU speed again, the process will be able to read and write from the queues faster, up to a certain point, where the NICs will be slower in refilling the RX queue than the CPU is in clearing it. Then the ebs will drop from the set batch size of 32, indicating that the CPU is faster than the NICs in the forwarding application.

As Figure 5.2 shows, with a modified packet buffer size (the size of one memory pool entry, i.e., the memory footprint of one packet in the main memory) of 4096 Byte, the application is CPU-bound up to 2.00 GHz. Only when reaching 2.40 GHz, the CPU is "faster" than the NIC, emptying the RX queue faster than the NIC can fill it. In this scenario, one packet needs a whole 4 KiB memory page, and both the MMU and the IOMMU are bound by TLB and IOTLB cache misses: The memory footprint of the whole DMA memory is $4 \times 8\,\text{KiB} + 2 \times (4096 \times 4096\,\text{B}) = 32.032\text{MiB}$, which is way to large for even the MMU's 4096 pages. The second line depicts the same load with a unmodified packet buffer size of 2048 Byte. Now, two packets fit into one 4 KiB memory page. Then, the application is CPU-bound only

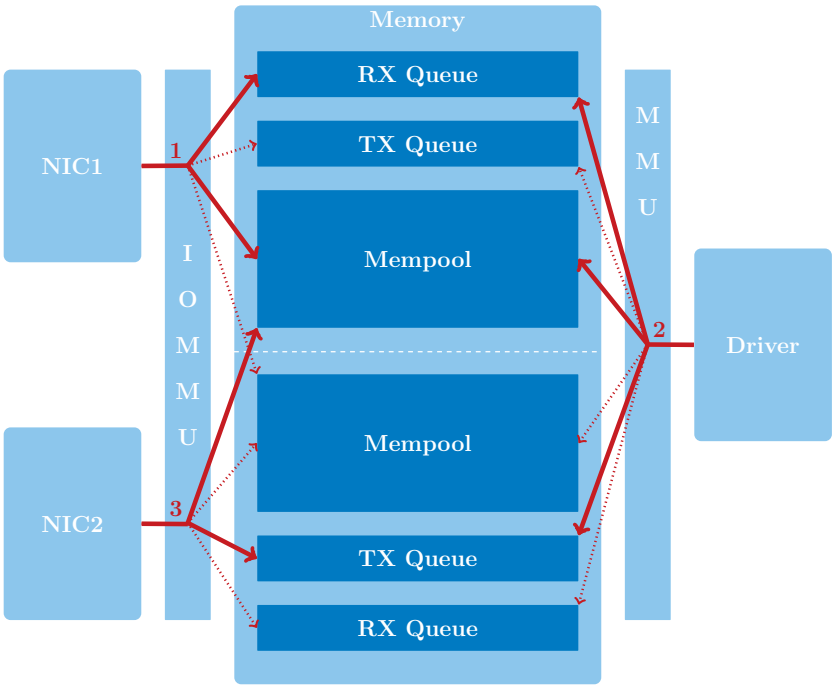

Figure 5.3: Model of the ixy forwarder with the IOMMU. Both the IOMMU and the MMU need to hold the pages for all four queues and and all accessed packet buffers in both mempools.
The simplified path for a single packet forwarded from NIC1 to NIC2 is marked in thick lines: The packet is received by NIC1, which then writes the descriptor to the RX queue and the packet into the packet buffer in its mempool. The driver checks the RX queue for new packets, touches the packet in the mempool and moves the descriptor to NIC2's TX queue. NIC2 then reads the descriptor from its TX queue, reads the packet from NIC1's mempool and sends it.

up to 1.20GHz, any faster than that the CPU out-paces the NIC. The memory footprint of the DMA memory is about half the size from before: $4 \times 8\,\text{KiB} + 2 \times (4096 \times 2048\,\text{B}) = 16.032\,\text{MiB}$, which fit (almost) in the MMU's TLB. This shows, that the IOTLB has fewer entries than the TLB, and thus more frequent misses, so the NICs can not push new packets fast enough into the memory, and the CPU will wait for new packets most of the time, decreasing the ebs.

5.2 Measuring the Number of Pages in the IOTLB

To find out the number of entries in the IOTLB, one needs to understand in which way the NICs access the main memory. Figure 5.3 depicts a simplified version of the process of forwarding one packet.

The following section will use the unmodified values that are hard coded in the ixy driver. The forwarder processes packets in batches of 32 (*batch size*) packets at once. The buffer for one packet in main memory is 2048 Byte (*packet buffer size*). The number of entries in one NICs mempool is 4096 of these packet buffers (*mempool entries*). The length of the RX and TX queues is 512 entries each (*queue entries*).

This leads to 8 KiB memory size for each queue, since the queue itself contains only the packet descriptors with 16 Byte each. Additionally, each NIC gets a mempool, that is large enough for 4096 entries of 2048 Byte packets each, adding up to 8 MiB of main memory for every mempool. All in all this means that 8.016 MiB of main memory are reserved for each NIC.

Since the main application of ixy right now is the `ixy-fwd` forwarder application, that application is used in this work for performance testing. This means, that DMA-able memory for the forwarder application is 16.032 MiB all together.

When running, every NIC accesses all its RX and TX queue entries regularly, pushing the memory mapping table entries containing these ($2 \times 2 \times 8\,\text{KiB} = 32\,\text{KiB}$) into the IOTLB. Since they are accessed rather often, it is safe to assume, that they do not leave the IOTLB, which evicts the least recently used lines first. Then, for each packet that is received by the NIC (and a free entry in the RX queue of this NIC exists), the packet is written to the NICs mempool, thus 2 KiB of memory are accessed - ideally one 4 KiB page per two packet buffers. Because of the way the mempool is used in ixy with a *free stack*, two consequently accessed packet buffers will most likely be consequent in physical memory.

If no free entry is found in the RX queue, the packet is discarded by the NIC. This happens only if the application is CPU bound, which should not be the case at high CPU speeds. An access to the mempool is thus only made whenever the CPU cleared an entry in the RX queue. The application tries to clear the RX queue regularly, copying the descriptors in the RX queue in a *forwarder descriptor buffer* it holds for packets to be sent out, and then assigning a new packet buffer from the mempool to this RX queue entry. This buffer comes from the top of the *free stack*, in a manner that recently used buffers that are not needed anymore (because the other NIC sent the packet out and the driver cleaned them from the RX queue, see below) gets reassigned, effectively re-using only a small subset of the packet buffers in the mempool. The driver then cleans up descriptors in the TX queue, freeing packet buffers in the mempool of that NIC. The newly freed buffers are inserted into the free-stack, where new buffer are taken from in the RX step in the next `forward` loop. After the driver sends a packet into the TX queue of a NIC, the NIC reads the packet from the mempool (of the other NIC), i.e., accessing the memory that was just accessed by the receiving NIC. Because of the way the *TX cleaning* is implemented, a maximum of two full *clean batches*, which is the same as the batch size, of buffers is added to the regularly used buffers, so the number of buffers that are regularly used is described by the following formula:

$$\#\text{buffers} = \text{rx queue length} + 2 \times \text{batchsize}$$

This number, multiplied with the packet buffer size (2048 by default) plus the size of both queues (RX and TX each 8 KiB by default) gives us the memory that is accessed *by each NIC* regularly. For simplicity, the size of the TX queue is always set to the same size as the RX queue. Dividing this number by the page size (4096 Byte) will give the number of pages that are regularly accessed, as calculated by Equation 5.1.

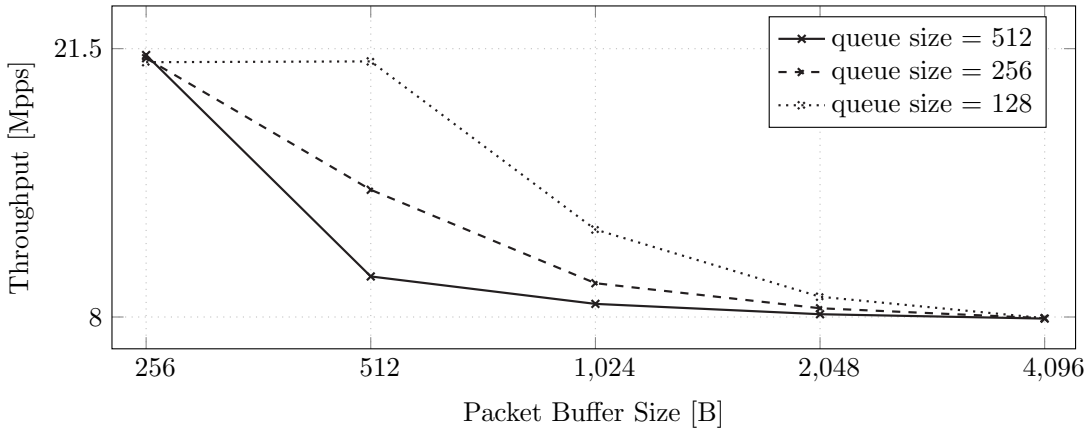

Figure 5.4: Forwarding throughput at variable RX queue size and packet buffer size, CPU clocked at 2400 MHz. The NIC is capped at 21.5 Mpps by the PCIe bus. Minimum throughput of about 8 Mpps is achievable even with very frequent IOTLB cache misses. The data for this figure can be seen in Table 5.1.

$$\begin{aligned}&\#\text{pages} = 2 \times (\#\text{buffers} + 2 \times \text{queue size})/\text{pagesize} = \\ &2 \times ((\text{queue length} + 2 \times \text{batchsize}) \times \text{packetbuffer size} + 2 \times \text{queue length} \times 16\,\text{Byte})/4096\,\text{Byte}\end{aligned} \tag{5.1}$$

Whenever this number is larger than the number of IOTLB entries, a drop in throughput will be noticeable. When the application is CPU bound, e.g., because of a very low CPU speed, the number of regularly accessed buffers and thus pages will differ, as the RX queues might not be cleared right away and significantly more packet buffers will be needed.

5.3 Results of the IOMMU Measurement

Looking at the results in Table 5.1 and Figure 5.4, the IOTLB of the IOMMU in the investigated CPU should feature between 80 and 84 pages. Since the application is not CPU bound in above scenarios, the *effective batch size* differs from the set batch size: The CPU can clear the RX and TX queue much faster than the NICs can refill them, thus the number of regularly accessed pages might be smaller than the one calculated above. Thus, a **IOTLB size of 64 entries** is suspected, as also concluded by Neugebauer et al. [33]. Since the ixy driver forwards 21 Mpps at this point, any output of the driver regarding the number of regularly accessed buffers is just so imperformant, that it makes the application CPU bound again, and the number of Mpps forwarded drops as the number of accessed buffers increases. A working method to really *count* the number of regularly accessed buffers, and thus pages, with ixy could not be found.

batch size	packet buffer size	queue entries	accessed pages	forwarded Mpps
32	4096	512	1160	7.92
32	2048	512	584	8.14
32	1024	512	296	8.66
32	512	512	152	10.04
32	256	512	80	21.18
32	4096	256	644	7.94
32	2048	256	324	8.44
32	1024	256	164	9.70
32	512	256	84	14.40
32	256	256	44	21.02
32	4096	128	388	7.94
32	2048	128	196	9.02
32	1024	128	100	12.40
32	512	128	52	20.86
32	256	128	28	20.82

Table 5.1: Pages accessed at variable RX queue size and packet buffer size, CPU clocked at 2400 MHz. This data was used for Figure 5.4. Especially relevant are the accessed pages, calculated with Equation 5.1.

5.4 Limits of the PCIe Bus

Additionally it can be noted that the NICs are apparently capped at pushing 21.5 Mpps over the PCIe bus. The Intel 82599ES NIC is a dual port network card that runs on 8 PCIe 2.0 lanes. In the paper *understanding PCIe performance for end host networking* [33], the PCIe read/write bandwidth was found by Neugebauer et al. to be capped at small transfer sizes (i.e., small packets) because of the packetizing that happens on the PCIe bus. This means, that the PCIe bus really caps the performance of the ixy drivers at about 21.5 Mpps with small packets. This performance problem could be solved by increasing the transfer size of the card, either by telling the NIC to batch PCIe transfers, or simply generating larger packets with MoonGen (not forwarding more Mpps, but at least saturating the two connected 10 GBit/s network lines). This PCI bottleneck can easily be bypassed by using NICs on separate PCI lanes, which leads to higher throughput, being bound by CPU clock.

Chapter 6

Results

The investigated server features an Intel Xeon E5-2620 v3 6-core CPU with a base frequency of 2400 MHz, and a turbo frequency of up to 3200 MHz. The 32 GiB (4×8 GiB) of DDR4 main memory on this server were clocked at 1866 MHz. For most performance measurements, the CPU was clocked at 2400 MHz, with Turbo Boost disabled. The load generating server is a Intel Xeon E5-2630 v4 10-core CPU with a base frequency of 2200 MHz. The load is generated with the software packet generator MoonGen [14]. The network cards used are Intel X520-T2s, two-port SFP+ 10 GBit/s network adapters attached to PCIe 2.0×8 ports. Comparable results were achieved with two Intel X540-T2s, copper Ethernet siblings to the aforementioned X520-T2s. Both servers were running Debian stretch on the Linux 4.9 kernel. The C code for ixy was compiled with the ixy `Makefile` and `gcc` 6.3.0. The Rust code for ixy.rs was compiled with `rustc` 1.33.0. Both `ixy` and `ixy.rs` were pinned to a single CPU core using `taskset -c`. PTI was used for Meltdown, `__user pointer sanitization` for Spectre v1 and `full generic retpoline` for Spectre v2 mitigation. Since ixy is a user space driver, and there are not many kernel syscalls while actually running the application, neither Meltdown nor Spectre mitigation influence the performance or latency significantly.

6.1 Performance

For performance comparison with the non-IOMMU enabled ixy and ixy.rs drivers, forwarding capacity in Mpps (million packets per second) was measured depending on forwarding batch sizes. Each test ran 2 minutes, and the average Mpps over the whole test time was used. The packet size was always set to the worst-case of 64 Byte. The size of the RX and TX queues was set to the default 512 entries, the packet buffer size to the default 2048 Byte and the size of the mempool to the default 4096 packet buffers. TX clean batch size was also set to the default size of 32 entries. Also, a memory page size of 2 MiB was set, as is by default. The impact of using a smaller page size is investigated in the previous chapter of this thesis.

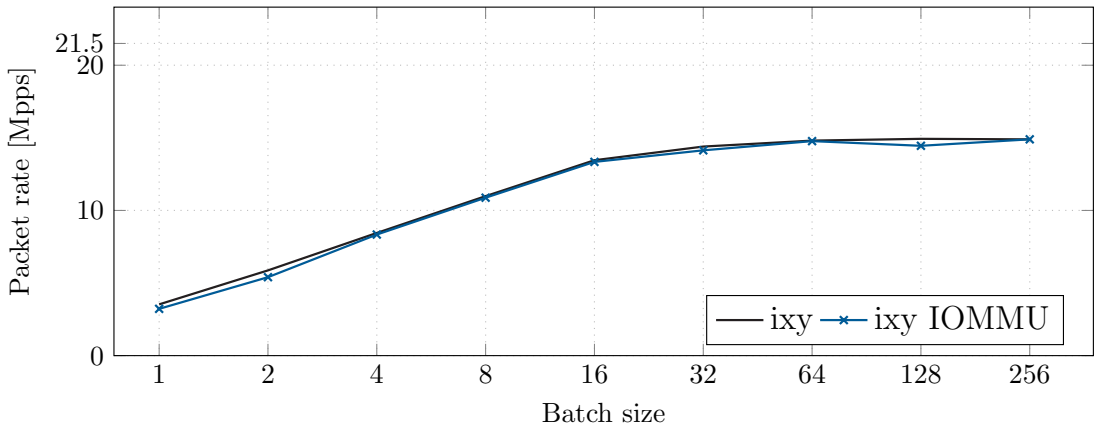

Figure 6.1: Forwarding capacity of the ixy driver variants at CPU speed of 1600 MHz. The lines are mostly overlapping.

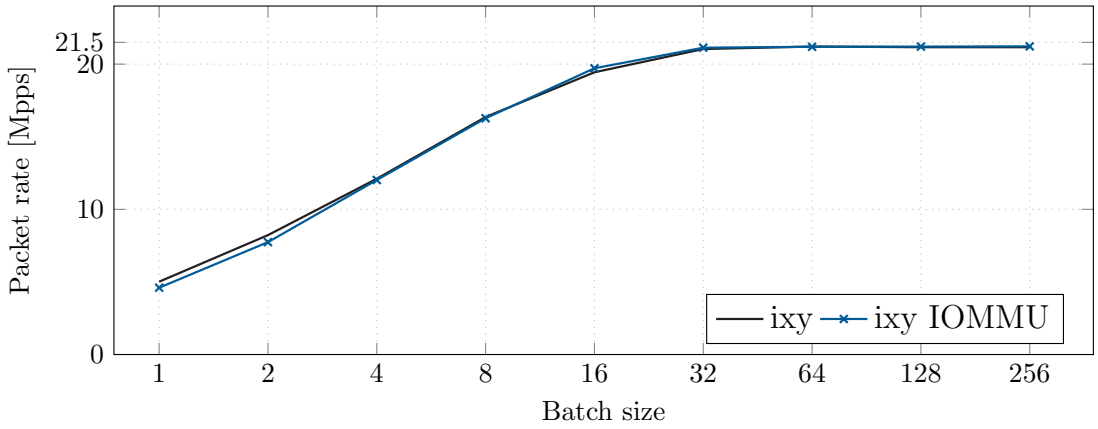

Figure 6.2: Forwarding capacity of the ixy driver variants at CPU speed of 2400 MHz. The lines are mostly overlapping.

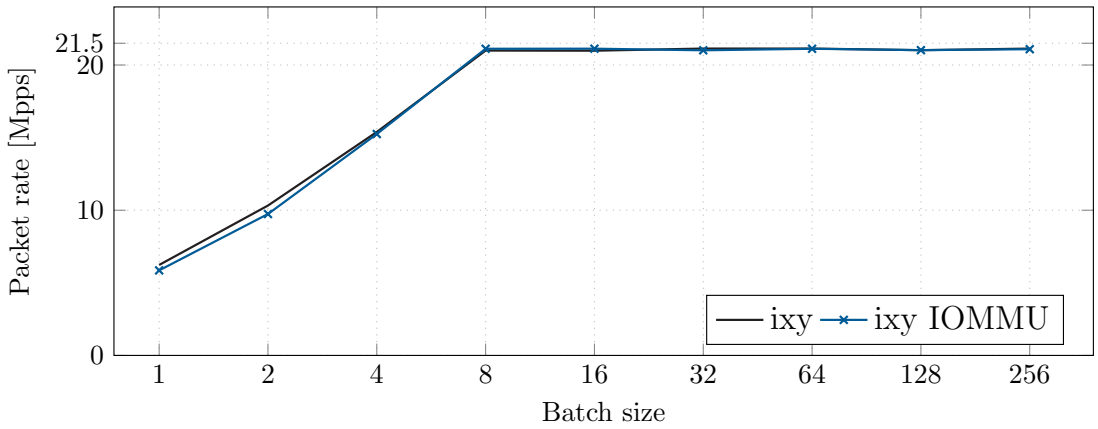

Figure 6.3: Forwarding capacity of the ixy driver variants at CPU speed of 3200 MHz turbo frequency. The lines are mostly overlapping.

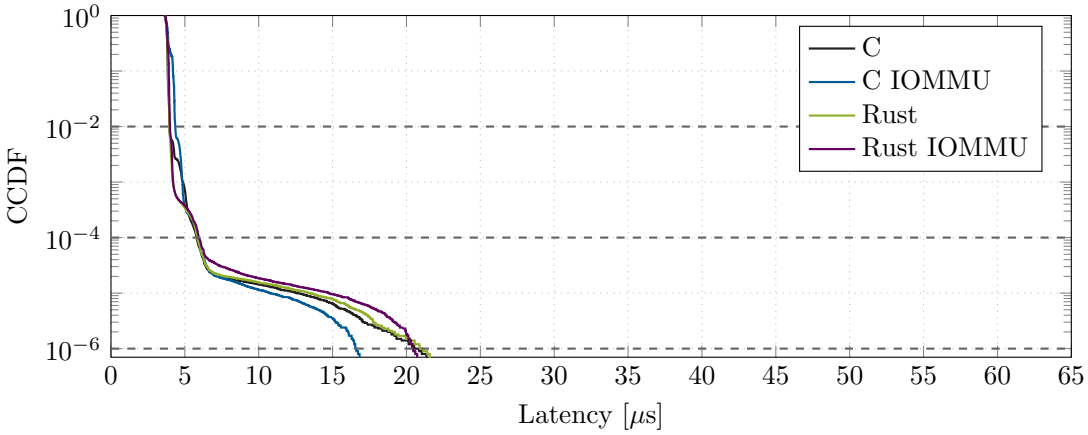

Figure 6.4: Latency distribution of packets when forwarding 1 Mpps

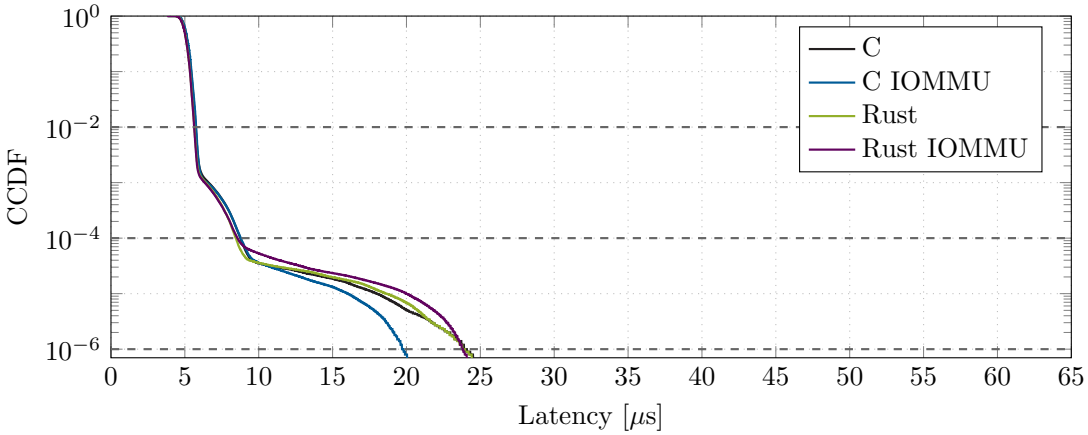

Figure 6.5: Latency distribution of packets when forwarding 10 Mpps

As can be seen in Figure 6.1, Figure 6.2 and Figure 6.3, the usage of the IOMMU does not affect the forwarding capacity of the ixy driver significantly. Forwarding rates for different CPU speeds and batch sizes are the same for the IOMMU enabled and the vanilla variant of `ixy`. The same holds true for `ixy.rs` and its IOMMU counterpart.

6.2 Latency

For latency measurements, all packets were also captured before and after the device under test with fiber optic taps, and the MoonSniff framework was used to acquire hardware timestamps (with 25.6 nanosecond precision). All these tests were run for about 160 million packets, when generating a forwarding load of 1, 10 and 20 Mpps, at 64 Byte per packet. For the latency tests, the CPU was clocked permanently at 2400 MHz.

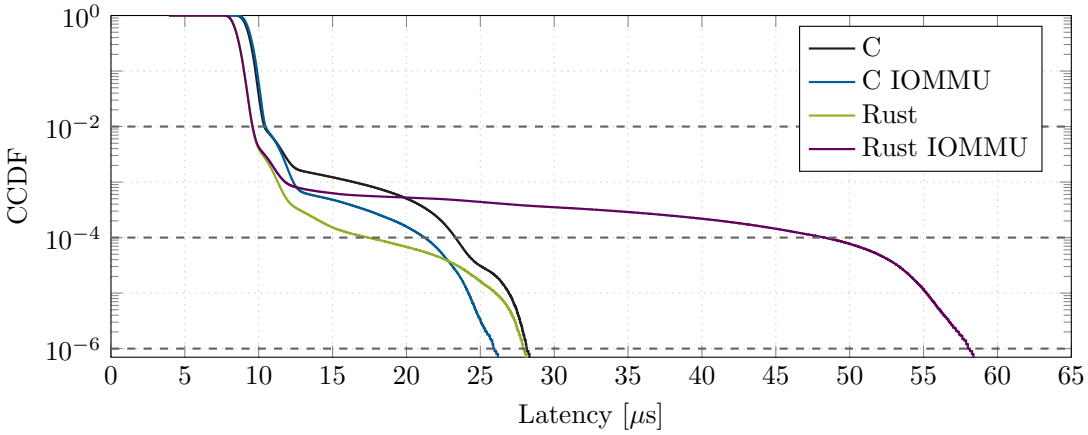

Figure 6.6: Latency distribution of packets when forwarding 20 Mpps

Figure 6.4, Figure 6.5 and Figure 6.6 show the complementary cumulative distribution function of the packet latencies. When forwarding 1, 5 and 10 Mpps, the latency distribution is relatively the same, with more than 99.9999 percent of all packets accumulating under 10μs of latency by getting forwarded by any implementation of ixy or ixy.rs. The maximum latency introduced in these tests seem to be about 25 μs. Curiously, the IOMMU implementation of ixy in C introduces clearly less latency than the other three implementations.

Figure 6.6 shows the latency of packets forwarded when approaching the CPU bottleneck at 20 Mpps. Then, only 99.99 percent of all packets have a latency of less than 10 μs introduced, with a major latency introduction of 20 to (in the case of the IOMMU implementation of ixy.rs) 50 μs more (adding up to 30 or 60 μs) for the rest of all forwarded packets. At this graph it is clearly visible, that at least on this CPU the Rust implementation of ixy takes less time per packet than the C version. Only for a very small percentage of packets, the Rust implementation takes more time than the C implementation.

6.3 C and Rust on Different CPUs

For the investigated CPU (Intel Xeon E5-2620 v3), the C implementation is significantly and reproducible slower than the Rust implementation. Tests on other servers indicate, that this is not the case for all CPUs: All other investigated machines show the same behavior as was measured by S. Ellman and P. Emmerich in [12] and [11], where the C implementation is up to 10 percent faster than the Rust implementation. This holds true for the IOMMU enabled implementation, too, on all but the one investigated CPU.

Why this is the case, could not be established in the course of this thesis. Measurements with the `perf(1)` tool revealed, that the instructions per packet are comparable for `ixy` and `ixy.rs`, but with `ixy.rs` less cycles per packet were needed than with `ixy`, effectively meaning than with Rust, more instructions per cycle were possible than with C. It needs to be stated again,

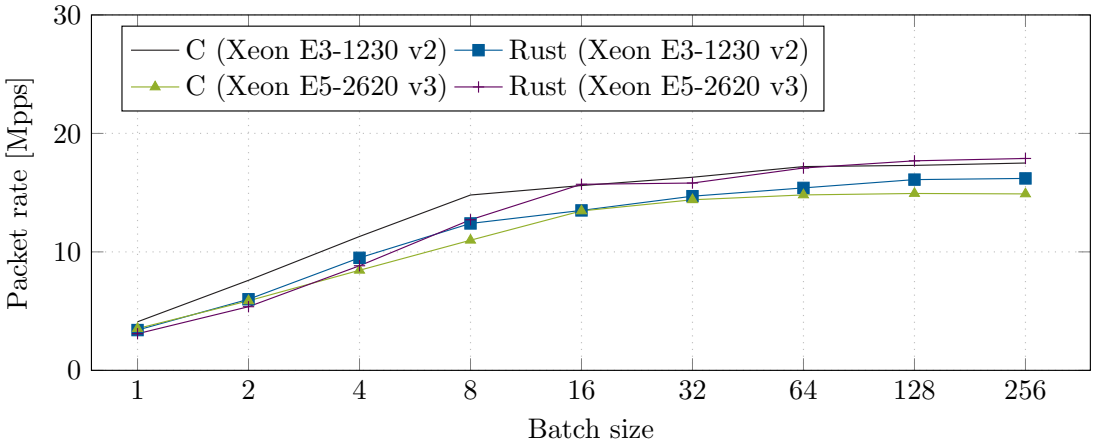

Figure 6.7: Speed comparison of ixy and ixy.rs on two different Intel Xeon processors

that this was only measured on this one Server and CPU, and all other investigated CPUs behaved as stated above and C was significantly faster than Rust. Also, this holds true for several versions of `rustc` (1.30 to 1.33 were tested) and the `gcc` (6.3.0 and 8.2.0 were tested). Compiling with `llvm` on the other hand produced slower results than compiling with `gcc` on all tested platforms.

Chapter 7

Conclusion: Safe and Secure User Space Drivers

In the course of this thesis, the educational user space network drivers `ixy` and `ixy.rs` were fitted with the ability to take use of the IOMMU to isolate PCIe NICs from the main memory and the rest of the host system. The `libixy-vfio` C library was written to make it easy to use the IOMMU in other drivers. The IOMMU was modeled after the findings in this thesis, which provides additional insight into the inner workings of it and makes it easier to understand what goes on and how it works.

The addition of IOMMU support to the investigated ixy driver in C and Rust is - though up to now it was not well documented - relatively easily to do. The full source code for ixy, ixy.rs and the patches that introduced IOMMU support into both as well as the `libixy-vfio` library are freely available at Github [12, 11].

The usage of the `libixy-vfio` library for C could speed up the implementation of IOMMU into other user space frameworks significantly, and the implementation in other programming languages is also possible, as proven by this implementation of IOMMU support into ixy.rs. Adding IOMMU support should thus be possible to any language that has support for the `ioctl` syscall (e.g., the Go language, Haskell, or Python), or can take advantage of the `libixy-vfio` library written in C.

It was found, that the IOMMU has 64 IOTLB entries for 4 KiB memory pages, at this page size being not suitable for a network driver with a memory impact of more than 16 MiB. Using 2 MiB pages solves that problem, though. The performance impact of using the IOMMU is, for 10 GBit/s NICs and the investigated CPU, negligible, as is the latency introduction by using the IOMMU.

So, *Using the IOMMU for Safe and Secure User Space Network Drivers* is not only possible, but also does not introduce any additional performance issues. By using the patches for `ixy` and

`ixy.rs` or the `libixy-vfio` library, NICs with a potentially questionable and dangerous closed-sourced firmware can securely be used, and fully isolated from the rest of the system. The possibility for a programmer to introduce memory errors into his machine is greatly reduced by restricting the NICs DMA access to certain areas, making the above mentioned user space network drivers a safe choice.

7.1 Limitations and Remaining Questions

Although this thesis gave an overview over the IOMMU and its history, implemented its usage and discussed some features of it in detail, not all details of it could be investigated. Also, since ixy and its IOMMU implementation are relatively young, there might still be bugs to be fixed and features to be implemented. Further questions regarding (user space) network drivers, the IOMMU and even the PCI(e) bus come to mind easily:

7.1.1 Bottlenecking of the PCIe Bus

Visible on Figure 6.2 and Figure 6.3 is the hard limit of ixy at about 21.5 Mpps. This behavior is odd, but can be seen in all implementations of ixy. One explanation for this behavior could be, that the PCIe bus is bottlenecking: Data on the PCIe bus is packetized, and at small packet sizes, the full throughput of PCIe 2.0 can not be achieved [33]. Most transferred data on the PCIe bus will be either the size of the queues (8 KiB) or the size of actual ethernet packets (64 Byte). For a lot of small PCIe packets, the headers will take an increasing amount of bandwidth, thus reducing the effective bandwidth. At 21.5 Mpps, the `ixy` driver is probably capped at this PCIe effective bandwidth. PCIe bus bottlenecking was already discussed in [33].

7.1.2 40 GBit/s and Faster NICs

For data and switching centers, 10 GBit/s cards are most of the time not fast enough. 40 GBit/s and faster NICs are used more and more often, and user space drivers for such appliances could be interesting as well. Since those new NICs are even less well known than Intel's ixgbe NICs, educational user space drivers for and isolation of this hardware could bring many new insights. The bottlenecking through the IOMMU and the PCIe bus was discussed [33], but expansion of ixy to 40 GBit/s and the isolation of 40 GBit/s NICs with the help of `libixy-vfio` might return further interesting results.

7.1.3 Different IOMMUs

Further investigation of the IOMMU on different CPUs or architectures can easily be done with FPGA NICs [33] or the `ixy` driver. Differences of the IOMMUs in different generations of Intel processors have not yet been investigated, especially server CPUs before the Sandy Bridge generation and very new processors could reveal interesting facts about the implementation of

Intel's VT-d in newer generations and Intel's will to improve it. Also, with AMD's Zen server processors, their feasibility for servers have greatly risen, and AMD's IOMMU might as well be investigated. Although implementation of ixy on SPARC micro architecture might not be feasible for production, it would make the comparison of x86-64 and SPARC implementations of the IOMMU possible.

7.1.4 Improvements on ixy

Since ixy is very young, many improvements to it and its other language siblings can still be made. Other languages, with the exception of the herein discussed Rust version, do not employ using the IOMMU up to now. With the help of the `libixy-vfio`, this could be retrofitted to most variants. Also, ixy does not yet feature a driver for 40 GBit/s NICs, which would improve the usability of ixy in real use-case relevant environments. Using a 40 GBit/s connection, the influence of the PCIe bus on networking could further be investigated, as discussed for Netronome NFP and NetFPGA NICs by Neugebauer et al. [33]. Using a 40 GBit/s connection, the influence of the PCIe bus on networking could further be investigated, as discussed for Netronome NFP and NetFPGA NICs by Neugebauer et al. [33]. For high loads, burst packet sending should be implemented to remove the PCIe bus bottleneck that occurs when transferring a lot of small packets over the PCIe bus. This would reduce overall packet latency and improve the maximum achievable overall throughout.

7.2 Future Work

The findings of this work suggest, that implementing IOMMU usage in every driver is highly advisable. Thus, retrofitting other user space drivers, e.g., the other language implementations of ixy or DPDK, might yield good results and make the usage of such drivers even safer for research work. Since Thunderbolt, and with it, external PCIe accessibility in consumer end devices like laptops is on the rise, implementing IOMMU isolation for a real use driver, like Linux' e1000 driver, would not only find a huge user base, but also make those end devices much safer.

Starting with enabling the IOMMU in more operating systems by default (as stated earlier, only Mac OS does this by default right now), and then rewriting the drivers to use it reasonable or retrofitting them with `libixy-vfio` will lead to a much safer environment for most regular users, by simply using a piece hardware of that existed in end user devices for a long time already anyways.

List of acronyms

BAR	Base Address Registers
BIOS	Basic Input/Output System
DMA	Direct Memory Access
DMAR	DMA Remapping
DPDK	DataPlane Development Kit
ebs	Effective Batch Size
GART	Graphics Address Remapping Table
GRO	Generic Receive Offload
IME	Intel Management Engine
IOMMU	I/O Memory Management Unit
IOTLB	I/O Translation Lookaside Buffer
IOVA	I/O Virtual Addresses
IRQ	Interrupt Request
LOC	Lines Of Code
MEI	Management Engine Interface
MMU	Memory Management Unit
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PCI	Peripheral Component Interconnect
PCIe	PCI Express
TLB	Translation Lookaside Buffer
USB	Universal Serial Bus
VFIO	Virtual Function I/O

Bibliography

[1] Darren Abramson. "Intel Virtualization Technology for Directed I/O". In: *Intel Technology Journal* 10 (Aug. 2006). DOI: 10.1535/itj.1003.02.

[2] Advanced Micro Devices Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*. [Online, accessed 2019-03-24]. 2016. URL: https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf.

[3] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. [Online, accessed 2019-03-24]. 2018. URL: https://www.amd.com/system/files/TechDocs/24593.pdf.

[4] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. "IOMMU: Strategies for Mitigating the IOTLB Bottleneck". In: *Computer Architecture*. Ed. by Ana Lucia Varbanescu, Anca Molnos, and Rob van Nieuwpoort. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 256–274. ISBN: 978-3-642-24322-6.

[5] Nadav Amit et al. "vIOMMU: efficient IOMMU emulation". In: *USENIX Annual Technical Conference (ATC)*. 2011, pp. 73–86.

[6] Jonathan Corbet. *Five-level page tables*. 2017. URL: https://lwn.net/Articles/717293/.

[7] Jonathan Corbet. *Four-level page tables*. 2004. URL: https://lwn.net/Articles/106177/.

[8] Cody Cutler, M Frans Kaashoek, and Robert T Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 89–105.

[9] DPDK developers. *DPDK homepage*. [Online; accessed 2019-03-24]. 2018. URL: https://www.dpdk.org.

[10] DPDK developers. *Using Linux IOMMU Pass-Through to Run DPDK with Intel VT-d*. [Online; accessed 2019-03-24]. 2014. URL: http://doc.dpdk.org/guides/linux_gsg/enable_func.html.

[11] Simon Ellman. "Writing Network Drivers in Rust". B.Sc. Thesis. Technical University of Munich, 2018.

[12] Paul Emmerich. *ixy github page*. [Online; accessed 2019-03-24]. 2018. URL: https://github.com/emmericp/ixy.

[13] Paul Emmerich et al. *ixy-languages github page*. [Online; accessed 2019-03-24]. 2018. URL: https://github.com/ixy-languages/ixy-languages.

[14] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference 2015 (IMC'15)*. 2015.

[15] Paul Emmerich et al. *Safe and Secure Drivers in High-Level Languages*. [online, accessed 2019-03-24]. 2018. URL: https://media.ccc.de/v/35c3-9670-safe_and_secure_drivers_in_high-level_languages.

[16] Paul Emmerich et al. *Writing Network Drivers in High-Level Languages*. [Online; accessed 2019-03-24]. 2019. URL: https://fosdem.org/2019/schedule/event/writing_network_drivers_in_high_level_languages/.

[17] ETSI. *Network Functions Virtualization Introduction*. URL: https://www.etsi.org/technologies/nfv.

[18] Sebastian Gallenmüller et al. "Comparison of Frameworks for High-Performance Packet IO". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2015)*. Oakland, CA, USA, May 2015.

[19] Luke Gorrie. *IOMMU requirements are too strict*. [Online; accessed 2019-03-24]. May 2015. URL: https://github.com/snabbco/snabb/issues/455.

[20] Stefan Huber. *ixy.iommu github page*. [Online; accessed 2019-03-24]. 2019. URL: https://github.com/huberste/ixy.

[21] Galen Hunt et al. *An overview of the Singularity project*. Tech. rep. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[22] David S Miller Miguel de Icaza. "The Linux/SPARC port". In: (1996).

[23] Intel Corporation. *Intel Virtualization Technology for Directed I/O Architecture Specification*. [Online, accessed 2019-03-24]. 2018. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[24] KVM developers. *How to assign devices with VT-d in KVM*. [Online; accessed 2019-03-24]. July 2016. URL: https://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM.

[25] O. Lawlor. *2012 CS 301 - Assembly Language - Memory Map Manipulation*. [Online, accessed 2019-03-24]. 2012. URL: https://www.cs.uaf.edu/2012/fall/cs301/lecture/11_05_mmap.html.

[26] Linux Kernel Source Documentation. *VFIO - Virtual Function I/O*. [Online; accessed 2019-03-24]. 2018. URL: https://www.kernel.org/doc/Documentation/vfio.txt.

[27] Moshe Malka et al. “rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers”. In: *ACM SIGPLAN Notices* 50.4 (2015), pp. 355–368.

[28] Athanasios Markettos et al. “Thunderclap: Exploring vulnerabilities in Operating System IOMMU protection via DMA from untrustworthy peripherals”. In: Internet Society, 2019.

[29] MITRE Corporation. *CVE Linux Kernel Vulnerability Statistics*. 2018. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[30] Mike Mol. *Rosetta Code Web Page*. [Online; accessed 2019-03-24]. 2007. URL: https://wwwrosettacode.org.

[31] B. Morgan et al. “Bypassing IOMMU Protection against I/O Attacks”. In: *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. 2016, pp. 145–150.

[32] S. Nanz and C. A. Furia. “A Comparative Study of Programming Languages in Rosetta Code”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 778–788.

[33] Rolf Neugebauer et al. “Understanding PCIe Performance for End Host Networking”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’18. Budapest, Hungary: ACM, 2018, pp. 327–341. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230560.

[34] Open vSwitch Developers. *Open vSwitch Website*. URL: http://www.openvswitch.org.

[35] Parallel and Distributed Operating Systems group at MIT CSAIL. *Biscuit research OS Github page*. [Online; accessed 2019-03-24]. 2018. URL: https://github.com/mit-pdos/biscuit.

[36] L. Prechelt. “An empirical comparison of seven programming languages”. In: *Computer* 33.10 (2000), pp. 23–29.

[37] Redox developers. *Redox Github page*. [Online; accessed 2019-03-24]. 2019. URL: https://gitlab.redox-os.org/redox-os/redox/.

[38] Fernand Lone Sang et al. “Exploiting an I/OMMU vulnerability”. In: *2010 5th International Conference on Malicious and Unwanted Software*. IEEE. 2010, pp. 7–14.

[39] seL4 developers. *Frequently Asked Questions on seL4*. [Online; accessed 2019-03-24]. Jan. 2019. URL: https://docs.sel4.systems/FrequentlyAskedQuestions.html.

[40] Snabb developers. *Snabb github page*. [Online; accessed 2019-03-24]. 2018. URL: https://github.com/snabbco/snabb.

[41] Patrick Stewin and Iurii Bystrov. “Understanding DMA malware”. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2012, pp. 21–41.

[42] Sun Microsystems. *Legacy IOMMU limitation and solution in SPARC sun4v*. [Online, accessed 2019-03-24]. 2017. URL: https://blogs.oracle.com/linux/legacy-iommu-limitation-and-solution-in-sparc-sun4v-v2.

[43] Adriana Szekeres. *IOMMU driver for MINIX 3*. [Online, accessed 2019-03-24]. 2011. URL: http://www.minix3.org/docs/szekeres-iommu.pdf.

[44] Juay K Tan and Robert W Kwong. *I/O cache with dual tag arrays*. US Patent 5,551,000. 1996.

[45] Linus Torvalds. *Linux kernel page*. [Online; accessed 2019-03-24]. 2018. URL: https://www.kernel.org.

[46] Felix von Leitner. *Fefes Blog: Performancevergleich mehrerer Sprachen und Implementationen eines simplen Probelms*. [Online; accessed 2019-03-24]. 2019. URL: https://blog.fefe.de/?ts=a2689de5.

[47] Alex Williamson. *An Introduction to PCI Device Assignment with VFIO*. [Online; accessed 2019-03-24]. Aug. 2016. URL: http://events17.linuxfoundation.org/sites/events/files/slides/AnIntroductiontoPCIDeviceAssignmentwithVFIO-Williamson-2016-08-30_0.pdf.

[48] XEN developers. *VTd HowTo*. [Online; accessed 2019-03-24]. Aug. 2017. URL: https://wiki.xen.org/wiki/VTd_HowTo.