DMA Security in the Presence of IOMMUs

Christian Schwarz
TU Dresden
Germany

Viktor Reusch
TU Dresden
Germany

Maksym Planeta
TU Dresden
Germany

ABSTRACT

Faulty, vulnerable or malicious PCIe devices can harm a system through DMA. IOMMUs can act as a security mechanism to protect against this problem by restricting the memory that is accessible via DMA. Unfortunately, there are methods to bypass the IOMMU restrictions. This paper is a survey over the currently existing bypasses and their feasibility. Current systems might be exploited from any untrusted source of DMA, which includes peripheral PCIe devices, virtual machines using SR-IOV, and even RDMA network cards, which enable remote attacks. Key strategies for the attacks presented here are Rowhammer, cache side-channels, and the exploitation of weaknesses in device drivers, e.g., for network cards, or protocols like PCIe or Ethernet OAM. An attacker can potentially achieve denial of service, the reading of confidential data, and even arbitrary code execution. Fortunately, there are some precautions to reduce the risks for affected systems.

KEYWORDS

IOMMU, DMA, PCIe, SR-IOV, RDMA, Thunderbolt, Rowhammer

1 INTRODUCTION

Attackers can use direct access to PCIe devices like NICs, GPUs or NVMe drives, to issue malicious DMA requests and take over the system. To protect against such attacks, an I/O memory management unit (IOMMU) [1, 10] limits what DRAM memory a device can access. IOMMUs enable secure device pass-through from host to guest in hardware virtualized environments with little runtime overhead. This paper presents a survey of known vulnerabilities and attack vectors which violate guarantees promised by IOMMUs.

To protect the system, the host OS programs the IOMMU to grant a device restricted access to DRAM. Now, an untrusted application or a guest VM can be allowed to access the device directly, bypassing the host. The host OS does not run a risk of having its sensitive data being leaked or manipulated, because the IOMMU denies requests that target non-mapped addresses.

For the aforementioned concept to remain valid, the IOMMU must rely on certain assumptions, like the PCIe devices themselves being trustworthy [1, 10], which do not always hold in practice. These broken assumptions create a window of opportunity for a potential attacker. The attacks differ in the required attacker power ranging from physically installing a malicious device down to connecting as a remote client. Correspondingly, a more powerful attacker can go as far as controlling the whole system, whereas some weaker ones may only be able to affect a system's availability or confidentiality.

We identify three major attack vectors aiming to bypass an IOMMU (see figure 1): a malicious device (1), a malicious device driver (2), and a malicious remote system (3). A malicious device is the most potent attack vector, whereas malicious remote systems are the least potent. An attacker may install a malicious device through a supply chain or evil maid attack. A device may be installed inside a system or connected to an externally accessible port that enables PCIe access through protocols like *Thunderbolt*. To be recognized by the host system, the malicious device pretends to be a known benevolent one, like a PCIe network card.

In certain settings, an attacker can take control of the device driver, without getting immediate control over the host system. Such situation may happen if the device driver is run by a microkernel [13], is part of an OS-bypass architecture [6, 22], or runs inside a guest VM. In this case, the attacker employs the device as a confused deputy to escalate their privileges.

A malicious remote system may abuse a device in a similar way as a confused deputy, when it gains network access to the vulnerable device. Such a situation occurs in *Remote Direct Memory Access (RDMA)* networks, which become more and more widespread in multi-tenant environments and public clouds. Acting as a confused deputy, such a device may leak sensitive information to a remote system. We foresee this attack vector becoming more prominent with the growing number of programmable devices (smart NICs, storage, etc.).

This paper proceeds by outlining the important details of IOMMU functionality (see section 2), followed by the description of the actual attacks. We conclude with our recommendations for building secure IOMMU-enabled systems.

FGBS '22, März 17–18, 2022, Hamburg, Germany

https://doi.org/10.18420/fgbs2022f-04

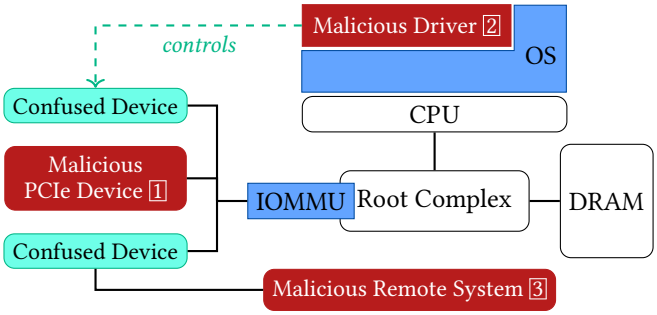

Figure 1: Different attack vectors through DMA. Components which might host malicious code are marked in red. Additionally, attacks can use bugs in *confused* devices to access (otherwise restricted) memory.

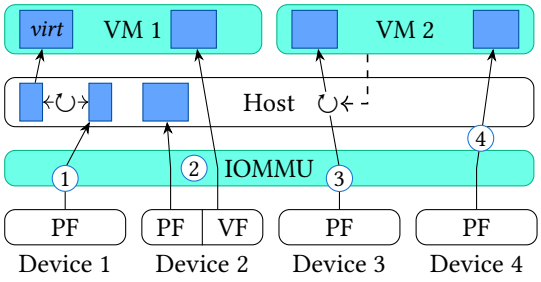

Figure 2: Types of device accesses for VM guests. The guest can only access a virtualized device provided by the host (①). The guest accesses an SR-IOV virtual function (VF, ②). The host passes part of the device MMIO, but controls the way it is accessed (③). The guest has full access to the device (④).

2 BACKGROUND

With the IOMMU enabled, PCIe devices are assigned to virtual address spaces, similar to the way the MMU manages address spaces for user-space processes. The OS preconfigures the IOMMU to translate device virtual addresses inside each DMA request to physical DRAM addresses. Additionally, IOMMUs enable device pass-through to VMs with low overhead by creating matching device virtual and guest physical address spaces.

DMA Address Remapping. The process of DMA address translation, begins with a PCIe device emitting a DMA request. It sends this request in the form of a Transaction Layer Packet (TLP) on the PCIe bus, where it is received by the *root complex* (RC) inside the CPU. Once such a packet is received, the RC-attached IOMMU uses fields from the TLP header to determine the *source-id* [10, p. 29] of the request. This id tells the IOMMU which address space to use for the address remapping. It then works out the address mappings by consulting the *context table* entry of this source-id for the corresponding *I/O page tables* [10], which are formatted similarly to the page tables of regular OS processes. The context and I/O page tables are set up by the operating system and reside in RAM. Only once this address translation has finished successfully, a request will be issued from the RC to main memory.

Device Sharing and Forwarding. To bypass an IOMMU, an attacker requires direct device access, which is not always granted, if the attacker controls only a part of the system. Figure 2 outlines several virtualization configurations used in practice to give a VM device access.

With a traditional virtualization technique (①), the host system passes a virtual device to a VM, instead of mapping the device Memory Mapped I/O (MMIO) buffers to the guest [27]. The host system must then translate the requests to the virtual device into requests to the physical device. This indirection protects a potentially vulnerable device well, but has high performance overhead. One could achieve higher performance by mapping device memory into the guest address space (③), even if some requests [8, 29] require explicit calls to the VM host. Still, such virtualization incurs significant CPU and latency increase.

The host may choose to pass the device either completely (④) as a *physical function* (PF), or, if the device supports hardware virtualization (SR-IOV [23]), partially (②) as a *virtual function* (VF). Most commonly, a device offers PFs with full device capabilities, and VFs with limited capabilities. When using a VF, a physical PCIe device can appear as multiple PCIe endpoints (and therefore IOMMU address spaces) by using different values for the *function-id* field inside TLP headers. Device pass-through offers the lowest overhead, but puts more trust on the device to be resilient against a malicious guest system or user-space driver.

3 MALICIOUS DEVICE

Malicious devices attacking the system are very hard to detect, and thus enable covert attacks. These attacks exploit shortcomings in the IOMMU specifications or vulnerabilities in architectures of actual systems.

IOMMU Bypass. The easiest practical way to introduce a malicious device into a system is Thunderbolt, which exposes the internal PCIe bus on USB-C, allowing for external peripherals to emit DMA requests. Additionally, older Thunderbolt controllers might bypass the IOMMU even if it is enabled [18]. This means that attached devices have unprotected

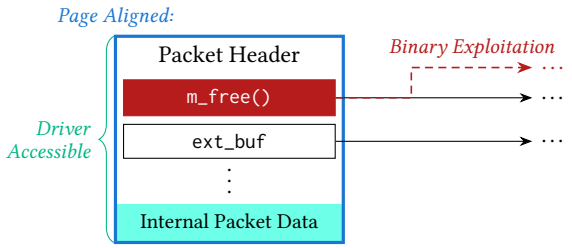

Figure 3: Exemplary network-driver data structure from OSX and BSD [14]. The malicious driver may overwrite the `m_free()` function pointer (which is originally meant to free `ext_buf`), gaining arbitrary code execution.

DMA access, which can be used to take over the system [18]. Intel treats using the IOMMU for Thunderbolt as a feature and calls it "Kernel DMA Protection" (kDMAp), which is only available on some systems shipped after 2019 [18, 20]. The kDMAp feature is configurable through UEFI, but known to cause compatability issues with some Thunderbolt devices [20]. For some older systems it is possible to enable it using third party patching [19]. MacOS on Apple hardware also has it always enabled [2, 18]. Otherwise, the safest option is to completely disable the Thunderbolt protocol from the BIOS.

IOMMU Translation Bypass. IOMMU specifications [1, 10] allow for disabling IOMMU checks upon the device request as part of the ATS feature of PCIe. The intent of this feature is to enable performance optimizations by letting the device handle IOMMU translations by itself [14, p. 9]. This way, a malicious device may create a DMA request for accessing any physical address, instead of faithfully following the host-OS instructions. Starting from Linux 4.20 [26], ATS became unavailable for "untrusted" devices, a category that, e.g., Thunderbolt devices fall into by default. Alternatively, one can disable the feature in the OS [25].

Subpage Granularity. Another class of exploits abuses the fact that virtual address mappings are performed on the granularity of pages. Most operating systems store, e.g., incoming network packets in buffers that contain the raw packet data as well as some headers with metadata. When the metadata memory is placed directly in front of the data, and a driver decides to map this buffer to the device, the device can access the metadata which might contain sensitive data. An attacker can spoof their device id to control the used driver, so drivers for any device, present or not, can be exploited. Older MacOS versions and current FreeBSD versions are still vulnerable to this attack [14], because the mapped headers contain overwritable function pointers, as shown in figure 3. Linux always has these headers in different memory regions, but it sometimes allocates these with the general purpose kernel allocators, leaking all sorts of other data that may be present from adjacient allocations [14]. The amount of driver code that needs to be hardened against this is very large. To deal with this issue, Markuze et al. [15] have build automated tools to detect these issues in existing Linux driver code. Through their *SPADE (Sub-Page Analysis for DMA Exposure)* tool it is, for example, possible to detect calls inside the kernel with a potential for generating multiple mappings for the same page [15, p. 400].

4 MALICIOUS VM

An attacker can use direct device access from within a virtual machine to compromise the system. Such settings are common for cloud providers, where virtual machines of different tenants are collocated. For safety reasons, untrusted guest systems may only use virtual functions, but not physical ones. Unfortunately, even virtual functions can be used for guest-initiated attacks.

EFC and OAM. Network cards are a common use case for SR-IOV, which also makes them an attractive target for exploits. Because Ethernet was originally not designed for sharing a single network card between multiple distrusting partners, there are functions within the protocol and its extensions that simply are not compatible with this security model. A prominent issue that affected many Ethernet network cards allowed virtual functions, to emit *Ethernet flow control* (EFC) packets [21]. These packets are meant to be sent by a device whose buffers are full, to request the sender to stop transmitting packets for a requested period of time. Ethernet is not a connection-oriented protocol and the link is shared between all virtual functions. Hence, virtual functions can construct a DOS attack on the whole link by repeatedly sending these packets to all connected MAC addresses, causing all senders to stop their transmissions. A similar attack is possible with the *Ethernet operations, administration and maintenance (Ethernet OAM)* protocol, which is supported by most network devices [28]. This protocol supports a link fault message, causing a receiving network switch to disconnect the respective host. Once again, a single virtual machine can use this functionality for a DOS attack on all other connected VMs. While the EFC problem is fixable with firmware updates [21], securing OAM currently requires to stop using SR-IOV and use a virtualized wrapper driver in the hypervisor again [28], switching from figure 2 case ② back to ①.

PCIe Error Propagation. In addition to vulnerabilities through network protocols, device implementers must also avoid DOS attacks through the PCIe protocol itself. For example, the

"Advanced Error Reporting Capability" feature of PCIe describes a field in the TLP header that may be set to indicate an error. One subtype of this error is an "Uncorrectable Fatal Error". If such an error is propagated to the operating system, it may decide to reset the link, the whole system or perform some other implementation-defined action [16]. If a virtual function can be manipulated to send this kind of error to the PCIe root complex, the owning virtual machine gains DOS capabilities. An example where this problem occurred were Intel's 700 Series Ethernet Controllers, as reported in *CVE-2019-0144* [5]. It was later fixed by a firmware update [4]. Overall, it becomes apparent that PCIe devices must be carefully designed to prevent untrusted drivers, virtual machines or containers from escalating privileges using these devices. Although IOMMUs theoretically allow secure sharing of devices by memory isolation, the presented attacks have shown that the protocols and implementations also need to enforce the isolation paradigm.

5 MALICIOUS REMOTE SYSTEM

Remote attacks are especially dangerous, and there are several know attacks without full mitigations, especially for older hardware.

Rowhammer. The basic idea of Rowhammer is to introduce bitflips in RAM by forcing the hardware to repeatedly access the same memory locations. This is typically not a remote exploit, but Tatar et al. demonstrated in 2018 [24], that high-speed RDMA networks make remote Rowhammer possible. It helps here that DMA bypasses CPU caches, increasing the possible frequency of RAM accesses. Newer DRAM standards like DDR5 and software-based approaches can partially mitigate, but not fully prevent this. The authors demonstrated the technique by crafting an end-to-end exploit of an RDMA-*memcached* instance and achieved arbitrary remote code execution.

Memcached Rowhammer Exploit. The memory on a DRAM Module is internally subdivided into *rows*. When any data should be read, the row which contains the requested data is activated and fully loaded into the *row buffer*. This partially discharges the capacitors that form the memory cells so they need to be regularly refreshed. As modern DRAM charge cells have shrunk to very few atoms, it can, very rarely, happen that different cells exchange charge due to physical effects, causing bits in memory to cross a charge threshold and "flip". The idea behind the Rowhammer attack is to read the same rows with a high frequency in order to explicitly trigger this effect. Because the row buffer caches the last accessed row, accessing a single row repeatedly is not optimal for getting a high frequency of discharges. At least two rows should be accessed alternatingly to bypass this cache effect.

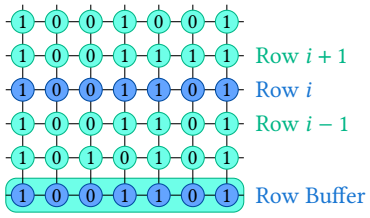

Figure 4: Double-sided Rowhammer [17, Figure 3]. The goal is to flip *Row* i by repeatedly reading from $i-1$ and $i+1$.

Repeatedly accessing both neighbors of a row i (i.e., $i-1$ and $i+1$ as seen in figure 4) significantly raises the likelihood and reliability of introducing a certain flip in row i. This is called *double-sided* Rowhammer. [7, p.17]

In the given example [24], the final memcached exploit does the following:

(1) Send RDMA packets that read from two specific "aggressor addresses" [24, p. 4] with a high frequency for a fixed amount of time.
(2) Read out the memcached data (using regular `GET`) and see where bit flips have occurred in the data.
(3) Strategically delete and create new memcached entries in order to create headers at a location where an exploitable bit flip was observed ("memory massaging" [17]). Then add a malicious entry at a location where pointers from the header will end up pointing to after a bit flip.
(4) Trigger (hopefully the same) bit flip again by emitting the same flood of RDMA read requests.
(5) Send memcached requests to trigger the usage of the forged data in a way that leads to arbitrary writes through the forged pointers. This can then be used to escalate to arbitrary code execution.

There are some mitigations against Rowhammer attacks. On a hardware level, ECC RAM, as it is often used in data centers, makes flipping bits much harder, though not impossible [3]. Furthermore, newer generations of DRAM like DDR5 improve the protections against Rowhammer [11, p. 14]. There are also software-based mitigations like the *ALIS* allocator [24], which allocates guard rows around DMA-accessible DRAM rows.

Cache Side-channel Attacks. Another way of indirectly bypassing the IOMMU is by using cache side-channel attacks. They allow an adversary to read otherwise not accessible data by observing changes in cache behavior. In 2012, Intel introduced *Data Direct I/O (DDIO)* [12], which is present on the Xeon E5 and E7 v2 lines of processors [9]. It applies DMA operations in the *last level cache (LLC)* instead of main memory. This makes DMA vulnerable to cache side-channel attacks.

Kurth et al. managed to exploit this by reading out sensitive data over the network using a *PRIME+PROBE* attack [12].

Through reverse engineering, the authors have been able to compute *eviction sets*, which are a set of addresses that, for a target address x, evict x from the cache when all written to. The target program, RDMA-memcached, only allows for RDMA accesses relative to an unknown (though page-aligned) base address. It is therefore necessary to dynamically probe the target machine in order to compute eviction sets through observing the results of educated guesses. Once eviction sets for the desired addresses are found, it is possible to evict a targeted cache line by performing writes to the eviction set addresses. It also becomes possible to observe whether the currently loaded cache lines from any eviction set have changed through timed reads. Cached reads will be consistently faster, even over RDMA [12, fig. 3].

One attack based on this records the timings of network packets, specifically the inter-packet times [12]. The exploit is achieved by sieving the DDIO cache lines for certain eviction patterns that are typical for networking ring buffers. Once the cache location of the correct ring buffer is found, it is possible to deduce that a packet has arrived by observing cache evictions. This enables an attacker to, for example, approximate the pressed keys in an interactive SSH session, where arriving packets map to the times between keystrokes [12].

The aforementioned exploits show that the high throughput and low latency of RDMA networking make it possible to exploit vulnerabilities which were formerly restricted to local adversaries. Therefore, DMA-based devices need to be better isolated when designing hardware, as the protection from the IOMMU is insufficient in these cases.

6 CONCLUSION

As direct device pass-through spreads out to production multi-tenant environments, the importance device security grows. In this survey, we outlined several cases where IOMMUs were not able to guarantee their apparent role. Fortunately, many mitigations already exist and are deployed as operating system, device firmware, or BIOS updates. Although, in some cases, like the Rowhammer attack, new hardware is the only efficient remedy. Nevertheless, experience demonstrates that IOMMUs are not invincible.

Our intention was to rise awareness of potential security issues connected to IOMMU-based isolation for device and protocol implementors. Implementation bugs can be addressed by design and test automation tools. Vulnerabilities in specifications can be reduced through verification processes. But even with all these precautions, physical resource sharing must be considered as a potential threat and PCIe-based connectivity should be handled with similar caution as other untrusted networks.

REFERENCES

[1] AMD 2021. *AMD I/O Virtualization Technology (IOMMU) Specification, 48882*. AMD. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf

[2] Apple 2021. *Apple Platform Security: Direct memory access protections for Mac computers*. Apple. https://support.apple.com/guide/security/direct-memory-access-protections-seca4960c2b5/1/web/1

[3] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. 55–71. https://doi.org/10.1109/SP.2019.00089

[4] Hareesh Khattri; Nagaraju N Kodalapura (Raju); Nam N Nguyen;Intel Corporation. 2021. *PCIe Device Attacks: Beyond DMA*. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-PCIe-Device-Attacks-Beyond-DMA-Exploiting-PCIe-Switches-Messages-And-Errors.pdf

[5] CVE-2014-0144 2019. INTEL-SA-00255. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00255.html

[6] DPDK 2013. *Data Plane Development Kit*. DPDK. https://www.dpdk.org/

[7] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. 300–321. https://doi.org/10.1007/978-3-319-40667-1_15

[8] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MasQ: RDMA for Virtual Private Cloud. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3387514.3405849

[9] Intel. 2021. *Intel Data Direct I/O Technology*. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html

[10] Intel Corporation. 2019. *Intel Virtualization Technology for Directed I/O*. https://www.intel.com/content/dam/develop/public/us/en/documents/vt-directed-io-spec.pdf

[11] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. Blacksmith: Scalable Rowhammering in the Frequency Domain. In *S&P*. https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf

[12] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCAT: Practical Cache Attacks from the Network. In *2020 IEEE Symposium on Security and Privacy (SP)*. 20–38. https://doi.org/10.1109/SP40000.2020.00082

[13] TU Dresden 2018. *L4Re – The L4 Runtime Environment*. TU Dresden. https://l4re.org/

[14] A. Markettos, Colin Rothwell, Brett Gutstein, Allison Pearce, Peter Neumann, Simon Moore, and Robert Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. https://doi.org/10.14722/ndss.2019.23194

[15] Alex Markuze, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3447786.3456249

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

[17] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack.

[18] Björn Ruytenberg. 2020. Breaking Thunderbolt Protocol Security: Vulnerability Report. https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf

[19] Björn Ruytenberg. 2021. *Thunderspy 2: Kernel DMA Protection for Unpatched Thunderbolt Systems.* https://thunderspy.io/ts2.html

[20] Björn Ruytenberg. 2021. *Thunderspy. When Lightning Strikes Thrice: Breaking Thunderbolt 3 Security.* https://thunderspy.io/

[21] Igor Smolyar, Muli Ben-Yehuda, and Dan Tsafrir. 2015. Securing Self-Virtualizing Ethernet Devices. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 335–350.

[22] SPDK 2013. *Storage Performance Development Kit.* SPDK. https://spdk.io/

[23] SR/IOV 2010. Single Root I/O Virtualization and Sharing Specification. , 100 pages.

[24] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 213–225.

[25] The kernel development community. 2021. The kernel's command-line parameters. https://www.kernel.org/doc/html/v5.15/admin-guide/kernel-parameters.html

[26] Mika Westerberg. 2018. Thunderbolt changes for v4.21. http://lkml.iu.edu/hypermail/linux/kernel/1812.1/01167.html

[27] Xen Project 2018. *Xen Networking*. Xen Project. https://wiki.xenproject.org/wiki/Xen_Networking

[28] Zhe Zhou, Zhou Li, and Kehuan Zhang. 2017. All Your VMs are Disconnected: Attacking Hardware Virtualized Network. 249–260. https://doi.org/10.1145/3029806.3029810

[29] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI)*. USENIX Association.