IOMMU Deferred Invalidation Vulnerability: Exploit and Defense

Chathura Rajapaksha, Leila Delshadtehrani, Richard Muri, Manuel Egele, Ajay Joshi
Department of ECE, Boston University, {chath, delshad, rmuri, megele, joshi}@bu.edu

***Abstract*—Direct Memory Access (DMA) introduces a security vulnerability as peripherals are given direct access to system memory, exposing privileged data to potentially malicious Input/Output (IO) devices. Modern systems are equipped with an IO Memory Management Unit (IOMMU) to mitigate such DMA attacks. The OS uses the IOMMU and IO page tables to map and unmap a designated memory region before and after the DMA operation, constraining each DMA request to the approved region. IOMMU protection comes at the cost of reduced throughput in IO-intensive workloads, mainly due to the high IOTLB invalidation latency. The Linux OS eliminates this bottleneck by deferring the IOTLB invalidation requests to a later time. This opens a vulnerability window during which a memory region is unmapped but the relevant IOTLB entry remains. In this paper, we present a proof-of-concept exploit, empirically demonstrating that a malicious DMA-capable device can use this vulnerability window to leak data used by other devices. Furthermore, we propose hardware-assisted mitigation for the deferred invalidation vulnerability by making minor changes to the existing IOMMU hardware and OS software. We implemented the proposed mitigation in the Intel IOMMU implementation in QEMU and the Linux kernel. Our security evaluation showed that our proposed mitigation successfully mitigated the deferred invalidation vulnerability and provided 12.7% higher throughput compared to the strict invalidation mode.**

***Index Terms*—IOMMU, DMA attacks**

I. Introduction

Modern computing systems are complex with many Input/Output (IO) devices such as Network Interface Cards (NICs) and accelerators connected to them. Direct Memory Access (DMA) allows IO devices to access system memory directly without involving the Central Processing Unit (CPU). However, DMA poses a threat to data confidentiality, integrity and availability, as IO devices are given direct access to system memory.

Attacks carried out by malicious DMA devices have been a concern for the security of computing systems for more than a decade [1], [2]. Trivial DMA attacks [1]–[4] were mitigated with the introduction of the IO Memory Management Unit (IOMMU) [5], [6]. The IOMMU allows IO devices to use IO Virtual Addresses (IOVAs) to access the system memory. The IOMMU uses IO page tables maintained by the OS to verify the read/write permission of each memory access and translate IOVAs to Physical Addresses (PAs).

The device driver of a DMA-capable IO device is responsible for mapping and unmapping memory for DMA operations. The IOMMU contains an IO Translation Lookaside Buffer (IOTLB) to cache recently translated IOVA-to-PA mappings. When the memory mapped for a DMA transaction is unmapped, the OS first sends an invalidation request to the IOMMU to invalidate the relevant IOTLB entries and then updates the IO page tables to prevent the devices from accessing the particular memory region. The IOTLB invalidation is known to take several thousand cycles [7] and becomes a bottleneck for high-throughput IO operations.

To alleviate the effect of IOTLB invalidation on the IO throughput, Linux has adopted deferred invalidation. In the deferred invalidation mode, IOTLB invalidation requests are batched, and a global IOTLB invalidation is performed every 10 ms or 256 invalidation requests. This deferred invalidation opens a vulnerability time window during which a DMA buffer is unmapped but the IOTLB entry for the buffer access is still valid, allowing the IO device to still access the physical address of the DMA buffer. As we demonstrate in Section IV, a malicious IO device can use this vulnerability window to read or write to the data of another device.

Over the last decade, two noteworthy software optimizations have been proposed [8], [9] to mitigate deferred invalidation vulnerability. However, these solutions either struggle to scale with increasing IO throughput demands [8] or suggest optimizations tailored exclusively to network workloads, which are incompatible with other IO workloads like storage [9].

In this paper, we present a proof-of-concept exploit and a hardware-assisted mitigation for the IOMMU deferred invalidation vulnerability. We show, for the first time (to the best of our knowledge), how a malicious DMA-capable IO device can use the deferred invalidation vulnerability to access the data of another IO device. Additionally, the exploit we present is also applicable to the IOMMU sub-page vulnerability [10], [11], another IOMMU-related known vulnerability.

To address the need for secure, low-overhead DMA operations for high-throughput IO workloads (e.g., GPU-intensive computer games, multi-threaded writes to NVMe storage, 200 Gbps network), we propose a hardware-assisted mitigation for the deferred invalidation vulnerability. Our solution reduces the large IO latency associated with strict invalidation while providing the same level of security guarantee, by preventing the reuse of stale IOTLB entries until the IOTLB is invalidated. The proposed mitigation for deferred invalidation vulnerability works at the IOMMU hardware level, making it compatible with any DMA operation.

We demonstrate the proof-of-concept exploit on an x86 machine emulated by QEMU [12]. The emulated machine contains two IO devices — a storage device connected through AHCI and an Ethernet connection through an Intel 82574L NIC — and an Intel IOMMU. We show that the malicious NIC can take advantage of the deferred invalidation of its unmapped memory to access data that was read from the storage device. We also implement and evaluate the proposed mitigation within the QEMU Intel IOMMU implementation using the same QEMU setup we used for the exploit. The proposed mitigation achieves 12.7% higher throughput compared to the strict invalidation mode while providing the same security guarantees. We chose QEMU for the implementation and evaluation because support for an IOMMU is not available in open-source hardware. Additionally, we demonstrate that the variations in the average network throughput of QEMU in different IOMMU

modes follow the same trend as that of a real hardware system (Section VI). In summary, we make the following contributions:

1) We demonstrate that IOMMU deferred invalidation can be used by a malicious DMA-capable device to leak the DMA data of other devices that utilize the same IOMMU. To the best of our knowledge, this is the first published proof-of-concept exploit for the IOMMU deferred invalidation vulnerability.
2) We propose a low-overhead, hardware-assisted mitigation for the deferred invalidation vulnerability, which is compatible with any DMA operation. The proposed mitigation involves minor modifications to the existing IOMMU hardware and Linux OS software.
3) We implement the proposed mitigation in QEMU and open-source it[1] along with the proof-of-concept deferred invalidation exploit.

II. Related work

The relevant IOMMU work can be divided into two main categories: (1) DMA attacks or exploits in the presence of an IOMMU [10], [11], [13], [14], and (2) performance or security [7]–[9], [15] improvements related to the IOMMU. Morgan et al. [13], [14] demonstrated an IOMMU-bypass exploit by updating the IO page tables at boot time before the IOMMU is enabled using a malicious peripheral. Thunderclap [10] explores how IOMMU protection is used in different OSes and demonstrates several DMA attacks that exploit the sub-page vulnerability in the IOMMU using a malicious DMA-capable device. In the aftermath of Thunderclap, Linux added support for software bounce buffers [16] as a mitigation to sub-page vulnerabilities in IOMMU. Markuze et al. [11] extend the sub-page vulnerability exploits demonstrated in Thunderclap by characterizing different variants of sub-page vulnerabilities, providing a complete picture of the attack surface. All the aforementioned works exploit and analyze sub-page vulnerabilities related to the IOMMU. In contrast, we focus on the deferred invalidation vulnerability and demonstrate that it can be exploited using a malicious DMA-capable device.

Markuze et al. [8] propose a software-based method to provide complete IOMMU protection using a set of permanently mapped pages (a.k.a. shadow buffers) in the IOMMU, while achieving better IO throughput compared to the IOMMU strict invalidation mode. Device access is restricted to shadow buffers and DMAed data is copied from/to these buffers, achieving byte-granular protection. Shadow buffers mitigate the deferred invalidation vulnerability because IOTLB invalidation is never required due to permanently mapped DMA buffers used by the devices. However, shadow buffers do not scale well with growing IO throughput demands due to their high CPU and memory overhead caused by copying DMAed data between buffers and memory duplication. Markuze et al. proposed DMA Aware Malloc for Networking (DAMN) [9] to alleviate the overhead of shadow buffers specifically for network workloads. DAMN selectively copies parts of the DMA buffer that are processed by the OS kernel (e.g., Ethernet header). DAMN is capable of achieving IO throughput comparable to the throughput of IOMMU in deferred mode. Unfortunately, DAMN is only compatible with network IO workloads, and does not improve workloads such as storage and zero-copy IO.

[1]https://github.com/bu-icsg/IOMMU-DIV

TABLE I
#CPU cycles IOMMU needs to process an invalidation descriptor.

	Avg	Min	Max
Intel Xeon Silver 4314 @ 2.40GHz	2548	1652	24394
AMD EPYC 7402P @ 2.80GHz	6968	420	38010

In contrast to the aforementioned works, we propose a hardware-assisted alternative to strict invalidation. Our solution can mitigate the deferred invalidation vulnerability with a comparatively lower performance cost and it has a broader applicability compared to DAMN [9] due to its compatibility with any DMA workload, including storage workloads. Furthermore, our solution is scalable compared to shadow buffers [8] as it does not require additional CPU time and memory to operate.

III. Background

The goal of this section is to familiarize the reader with the IOMMU operation, security, and IO throughput performance of different IOMMU modes. All experiments in this section were conducted on CloudLab [17] machines. For network throughput experiments, two machines with an Intel Xeon Silver 4314 CPU, an Intel IOMMU and 244GB RAM were connected through dual-port Mellanox ConnectX-6 NICs with a 200 Gbps connection between them. Linux uses the IOMMU in two modes: strict invalidation and deferred invalidation. We first provide an overview of how the IOMMU operates; then, we explain in detail about each IOMMU mode.

A. IOMMU Operation

OSes use the IOMMU to constrain the DMA of IO devices to designated DMA buffers. When DMA is used for receiving data from a device, the device driver first allocates the memory for the DMA buffer. Then the driver requests the kernel to map the physical address of the DMA buffer to an IO virtual address. The kernel does this by creating an IO page table entry in the IO page table for the device. Each page table entry contains the IOVA-to-PA translation information and access permission (read/write/both) for the PA. Furthermore, address translation and access control through IO page tables happen in 4 KB memory chunks called pages[2].

Next, the device driver signals the device to initiate the DMA transaction. The device accesses the DMA buffer using the IOVA of the buffer, and the IOMMU performs the IOVA to PA translation by walking the IO page table for the device and verifying the access permission to the PA before completing the translation. The IOMMU caches the recent IOVA-to-PA translations along with access permission in an IOTLB.

Once the DMA operation is completed, the device should no longer have access to the DMA buffer, so that the memory can be reallocated without breaking data confidentiality and integrity. For this purpose, the device driver unmaps the DMA buffer and the OS kernel sends an invalidation request to the IOTLB to invalidate the entry containing the IOVA-to-PA translation of the DMA buffer.

B. Strict Invalidation

In this mode, IOTLB invalidation is performed strictly during the IOVA unmapping process itself. IOTLB invalidation is known

[2]IO page size can vary depending on the OS and IOMMU hardware.

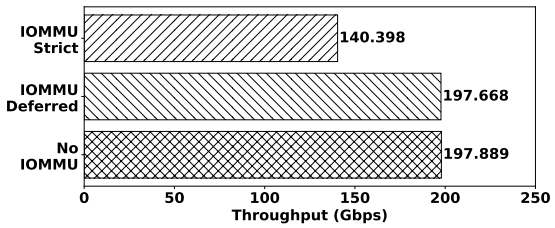

Fig. 1. Linux TCP throughput over 200 Gbps Ethernet, measured with 16 netperf [18] instances running in parallel for three different IOMMU modes.

to take more than 1000 CPU clock cycles [7]. We conducted an experiment to confirm whether this is still true with the modern IOMMU implementations. We determined the IOTLB invalidation latency by instrumenting the Linux kernel (6.4.0) source with RDTSCP instructions and measuring the average cycle count taken by the IOMMU to process a single invalidation descriptor. As shown in Table I, IOTLB invalidation still has a high latency in both Intel (Xeon Silver 4314) and AMD (EPYC 7402P) systems. We show in subsection III-D that this high latency causes the IO throughput to drop, especially in high-throughput IO workloads where IOVA mapping/unmapping happens millions of times per second.

C. Deferred Invalidation

To alleviate the aforementioned bottleneck, OSes have adopted deferred invalidation. In this mode, IOTLB entries are not invalidated strictly during the unmap process. Instead, IOTLB invalidation descriptors are queued in a per-CPU queue and all IOTLB entries belonging to an IOMMU domain are invalidated when the number of the descriptors in the queue exceeds 256 or every 10ms, whichever happens first[3]. Note here that an IOMMU domain is a set of address mappings and access rights that can be shared by multiple devices [6]. In most cases, each DMA-capable device is allocated its own domain. Therefore, invalidation of an IOTLB entry can be delayed by at most 10ms, leaving a time window during which a device has access to a physical memory address that may not belong to it. We discuss how a malicious device can exploit this deferred invalidation vulnerability in Section IV.

D. IOMMU Performance

Figure 1 shows the effect of different IOMMU modes on the maximum throughput of a multi-gigabit (200 Gbps) network connection in a modern computing system. We observe that the strict invalidation mode reduces the network throughput by 28.97% compared to the deferred invalidation mode.

IV. Exploiting IOMMU Deferred Invalidation

In this section, we demonstrate how the deferred invalidation vulnerability can be used by a malicious IO device to leak data of other devices. When the OS defers the invalidation of an IOTLB entry, it can lead to two vulnerable scenarios:
(1) After receiving DMA data from a device, the OS kernel unmaps the DMA buffer and then starts processing the data. However, the device can still access the physical address of the DMA buffer through a stale IOTLB entry. Such access can enable "time of check to time of use" (TOCTTOU) attacks for DMA-writable buffers, e.g., modifying a packet after it passes firewall checks [9].
(2) After a device (device A) reads from a DMA buffer, the OS frees the memory and may reallocate it immediately [19] to another device (device B) that is performing DMA write. As we show later in this section, this scenario can happen commonly during the deferred invalidation time window. This is a vulnerability as device A now has access to device B's data through the stale IOTLB entry.

At a high level, scenario (1) poses a threat to data integrity while scenario (2) poses a threat to the confidentiality of data. Below, we present a proof-of-concept exploit based on the second scenario.

[3]This is the behavior of the Linux kernel, other OSes may perform invalidation using a different criteria.

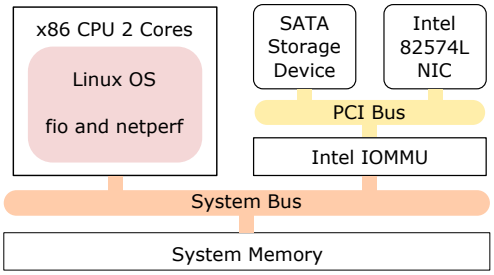

Fig. 2. Overview of the QEMU-emulated system that was used for IOMMU deferred invalidation proof-of-concept exploit.

A. Threat Model

We assume there is a malicious DMA-capable device connected internally to the system (e.g., a malicious SmartNIC, third-party integrated accelerator). The rest of the system including the IOMMU, other IO devices, and OS, works as expected and is trusted at all times. Furthermore, we consider a system that is protected against boot time DMA attacks [14]. The goal of the malicious device is to snoop on the data accessed by other devices.

B. QEMU-based Proof-of-concept Exploit

System setup: We use a QEMU-emulated x86 machine with the KVM hypervisor to demonstrate our proof-of-concept (QEMU version 6.2.0). Figure 2 shows the high-level system we used in QEMU. We have two devices connected to the system through an Intel IOMMU: a malicious Intel 82574L NIC and a victim SATA storage device. For this demonstration, the emulated system runs Linux 6.4.0 in deferred IOMMU mode with two CPU cores and 1 GB of memory. We used a system with an Intel Xeon Silver 4114 CPU as the host.
Software setup: The first step for exploiting deferred invalidation (with the second scenario) is to generate DMA traffic for both NIC and the storage device simultaneously so that a PA allocated for NIC gets reallocated to the storage device within a deferred invalidation time window. For this purpose, we used netperf [18] TCP_STREAM and fio [20] random read benchmarks to generate DMA traffic for the NIC and the storage device, respectively. We used the QEMU event trace feature to confirm that PAs of the NIC indeed get reallocated to the storage device frequently within a deferred invalidation time window. We also required a way to identify if the malicious DMA reads were able to read the DMA data of the storage device. For this purpose, we filled the storage device with a magic word (`0xc0fecafe`).
Malicious behavior: We integrated the malicious behavior into the 82574L NIC by editing the relevant source code in QEMU.

We added the capability of maintaining a pool of IOVAs of the DMA buffers used by the NIC for legitimate DMA reads. Then, when the NIC performs a legitimate DMA access, we follow it with a malicious DMA read request on a randomly selected IOVA from the pool (which may or may not be successful). If the malicious read request fails, we remove the IOVA from the pool. In other words, we are trying to read the data at IOVAs that were recently (and legitimately) used by the NIC, hoping that the PA of the IOVA is mapped to another device while the IOTLB entry of the legitimate access still remains.

Exploit: Figure 3 provides a detailed timeline of how the malicious NIC was able to read the victim's storage device DMA data during a deferred invalidation window in our exploit. ① The NIC device driver maps a memory region to the device for DMA, starting at IOVA `0xff528000` which maps to PA `0xe670000`. ② The NIC reads the data in the buffer through DMA and the device driver unmaps and frees the memory. However, IOTLB invalidation has not been performed yet (due to deferred invalidation). (③) At this point, the OS kernel considers PA `0xe670000` as unused and reallocates it to the storage device to write data. ④ Malicious NIC device reads the PA `0xe670000` through IOVA `0xff528000`, leaking the data of the storage device. The IOTLB entries created at ① and ③ are invalidated at ⑤ and ⑥, respectively. To check the exploit's functionality, we searched for the magic word, i.e., `0xc0fecafe` in the results of successful malicious DMA reads from the NIC.

Observations: We consistently observed the magic word in the malicious DMA read results when the exploit is running, indicating a successful exploit. The root-cause analysis for each reported leakage revealed two causes: 1) deferred invalidation vulnerability, and 2) sub-page vulnerability [8], [10], [11]. In other words, the malicious behavior of the NIC is capable of exploiting both these vulnerabilities. We discuss how sub-page vulnerability (which is not the main focus of this paper) causes data leakage in the following sub-section.

C. Data Leakage Through Sub-page Vulnerability

IOMMU enforces access control at the page granularity [5] while DMA buffers used by IO devices can be smaller than a page. For example, the default Maximum Transmission Unit (MTU) is 1500 Bytes for an Ethernet packet, which is mapped to a DMA buffer when transferring the packet through the Ethernet controller. This mismatch of granularity between the IOMMU protection and DMA buffer allocations can cause the DMA buffers of two devices to get mapped to the same page. This opens up the so-called sub-page vulnerability as each device is given access to the data of the other device.

The leakage through sub-page vulnerability can happen in the following scenario. The storage device uses the first half of a page as a DMA buffer to write device data. In the meantime, a DMA buffer of the NIC gets mapped to the second half of the same page. However, due to sub-page vulnerability, a malicious read from the NIC to the first half of the page will be successful, causing a storage data leak.

V. Hardware-assisted Mitigation for IOMMU Deferred Invalidation Vulnerability

A. Overview

We propose a low-overhead, hardware-assisted mitigation for the deferred invalidation vulnerability. This method offers the same security guarantees as strict invalidation but with a lower performance overhead. Implementing this mitigation involves minor adjustments to existing OS software and IOMMU hardware.

IOTLB invalidation latency: Prior work has shown IOTLB invalidation latency as the cause for the high-performance cost of strict invalidation [7], [9]. We take a step further and analyze the IOTLB invalidation process and the factors that may contribute to its high latency. To reduce the overall page table walk latency, IOMMU implementations may cache intermediate paging structure entries that are read during the page table walk [5]. An IOMMU compatible with a 4-level IO page table may have up to three paging-structure caches. When invalidating an IOTLB entry, relevant entries in the paging-structure caches must be invalidated to maintain the accuracy of the translations [5]. This process involves a cache lookup and invalidation in each paging-structure cache. Furthermore, IOTLB invalidation descriptors are provided to hardware through a queue that resides in the system memory. Therefore, the IOMMU performs a memory access to fetch the invalidation descriptor, which adds up to the invalidation latency.

B. Mitigation

The proposed mitigation stems from the observation that exploiting the deferred invalidation vulnerability requires a malicious device to reuse outdated IOTLB entries (see Section IV). So rather than strictly performing IOTLB invalidation, which significantly impacts performance (see Section III-B), we suggest preventing the reuse of stale IOTLB entries during the deferred invalidation window. We propose to do this by adding an indicator bit to each IOTLB entry. For an IOTLB entry, this indicator bit can be reset when we don't want any DMA request to use the IOTLB entry (after unmapping it). We argue that this prevention of the reuse of a stale IOTLB entry incurs a lower performance cost than an IOTLB invalidation during strict invalidation.

Functionality The indicator bit will be set to 1 when the IOTLB entry is initialized. When the OS kernel unmaps an IOVA, the IOMMU sets the indicator bit to 0 in the relevant IOTLB entry. If during a memory access, we get an IOTLB hit that has the indicator bit set to 0, the IOMMU returns a translation error, as this can happen only due to a malicious activity or bug in the device, thereby mitigating the exploitation of the deferred invalidation vulnerability. It is important to note that the proposed mitigation still requires the OS to perform deferred IOTLB invalidation, as the mitigation itself does not perform IOTLB invalidation. In other words, our mitigation relies on the IOMMU deferred invalidation mode, which is the default mode of operation in Linux for Intel and AMD systems. In summary, the proposed mitigation requires two actions: 1) the OS software communicates the IOVA of a DMA buffer to the IOMMU during its unmapping, and 2) the IOMMU hardware marks the relevant IOTLB entry (if it exists) for that IOVA as unusable by setting the indicator bit to 0.

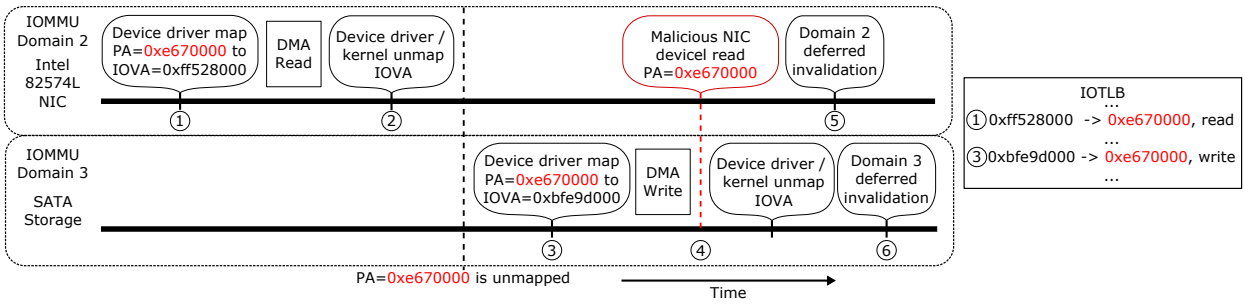

Fig. 3. Timeline for the deferred invalidation proof-of-concept exploit.

C. Design and Implementation

When using our proposed mitigation approach, when we have to reset the indicator bit during an IOVA unmap, we must perform an IOTLB lookup, which requires the relevant IOVA and the source ID of the device to which it is mapped. The source ID serves as a unique identifier for each device connected to the IOMMU, typically using the Bus:Device:Function (BDF) number for PCI/PCIe devices. Therefore, the main design question that we need to answer for implementing the proposed mitigation is: How do we communicate the IOVA and the source ID of the device to the IOMMU hardware when unmapping it? We decided to include a dedicated memory-mapped register (which we call `HW_INV_REG`) in the IOMMU hardware to receive IOVA and source ID during their unmap process. This requires minimal changes in both hardware and software as IOMMU already has memory-mapped configuration registers and the OS has the infrastructure in place for accessing these registers. We have successfully implemented this proposed mitigation in the QEMU Intel IOMMU implementation and the Linux kernel version 6.4.0. Further details regarding the specific hardware and software changes necessary to implement this mitigation are discussed below.

IOMMU Hardware Modifications The Intel IOMMU specification [5] limits the possible IOVA size to 56 bits for a five-level IO page table and 47 bits for a four-level IO page table. The QEMU Intel IOMMU implementation supports a four-level IO page table, allowing us to use 47 bits for the IOVA. Hence, we are able to use a 64-bit control register (`HW_INV_REG`) for capturing both IOVA and the 16-bit source ID. Furthermore, a single bit was added to each IOTLB entry as the indicator bit. Additionally, we changed the QEMU Intel IOMMU behavior according to the functionality of the mitigation as follows: When an IOTLB entry is first initialized, the indicator bit is set to 1. When unmapping an IOVA, OS writes the IOVA and relevant source ID to `HW_INV_REG`. Upon this write, the IOMMU hardware triggers an IOTLB lookup with the provided IOVA and source ID. If the lookup was a hit with the indicator bit equal to 1, the indicator bit in the IOTLB is reset to 0 (indicating that the IOTLB entry is now unusable). If a translation request to the IOMMU results in an IOTLB hit and the indicator bit in the IOTLB entry is already 0, IOMMU returns a translation error.

OS Software Modifications We first define the address of `HW_INV_REG` in the Linux kernel Intel IOMMU driver. Then we change the existing IOVA unmapping function in Linux to concatenate the IOVA that is unmapped and its corresponding source ID to a 64-bit word and write it to `HW_INV_REG`.

In Linux, it is possible for multiple IOVAs to unmap at the same time (by different threads), creating a race condition as unmap operations contend for the `HW_INV_REG` register. To avoid this, we acquire the existing mutex lock for writing to the IOMMU CSRs and release it after the write.

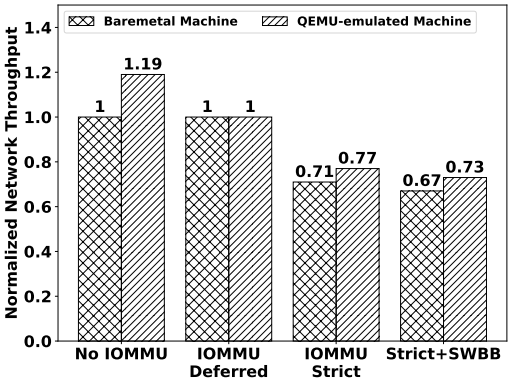

Fig. 4. Normalized network throughput of the real hardware machine and the QEMU-emulated machine in different IOMMU modes and software bounce buffers (SWBB) with strict mode (Strict+SWBB). Normalization is done with respect to the throughput in deferred invalidation mode.

VI. Evaluation

In this section, we evaluate the proposed mitigation for the IOMMU deferred invalidation vulnerability, in terms of security and network throughput. We used the same QEMU-emulated system that we used for the proof-of-concept exploit for the evaluation. We justify our use of QEMU for performance evaluation by providing an IO throughput trend comparison between a QEMU-emulated system and a real hardware system in Figure 4, where we show the normalized network throughput of the QEMU-emulated system and a real hardware system in different IOMMU modes. The network throughput of both systems roughly follows the same trend. Therefore, we argue that any network throughput performance changes that we observe in QEMU due to our modifications will follow the same trend in a real hardware system. We used the same system setup and throughput numbers reported in section III-D for the real hardware system. For the QEMU-emulated system, we used the hardware setup in section III-D. The QEMU-based system has 8 CPU cores, 32GB RAM, an Intel IOMMU, and an Intel 82574L NIC with KVM enabled. All evaluations were conducted with Hyperthreading and dynamic frequency scaling turned off in the host system.

A. Performance Evaluation

We used the QEMU setup mentioned above to evaluate the performance impact of the proposed mitigation for the deferred

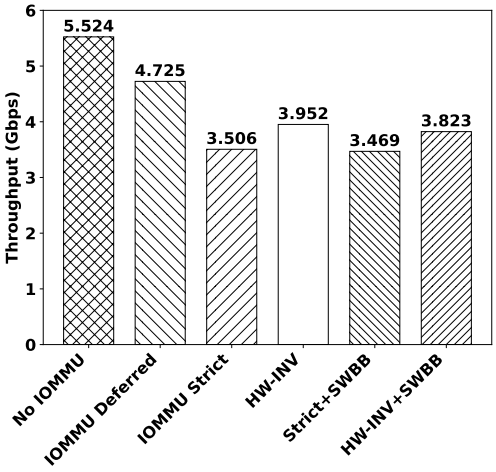

Fig. 5. Network throughput performance of different IOMMU modes and our proposed mitigation (HW-INV). We combined HW-INV with SW bounce buffers (SWBB) to provide complete IOMMU protection.

invalidation vulnerability. For each configuration, we ran the 4 instances of netperf TCP_STREAM benchmark for 10 seconds and accumulated the achieved throughput. This experiment was conducted 10 times for each configuration to get the average accumulated throughput. Four netperf instances were used as they provided the maximum network throughput in the deferred invalidation mode (baseline). As shown in Figure 5, our mitigation (HW-INV) was able to achieve 12.7% higher network throughput compared to the IOMMU strict invalidation mode.

B. Security Evaluation

After implementing the proposed mitigation, we performed a security evaluation to ensure that our exploit (described in Section IV) is no longer effective, i.e., data leakage across devices is not observed when our mitigation is present. For this evaluation, we used the same setup used for the proof-of-concept exploit. To account for randomness in the proof-of-concept exploit, we performed the security evaluation 5 times for each configuration. Our evaluation showed that our mitigation was able to block any data leakage due to deferred IOMMU invalidations. However, we observed slight data leakage (from the storage device). Our root-cause analysis showed that this leakage was purely due to sub-page vulnerability, which we observed for strict invalidation mode as well. To address sub-page vulnerability, we relied on the existing software-based mitigation in the Linux kernel, i.e., software bounce buffers [16], due to its low-performance overhead. We then combined our proposed mitigation with SW bounce buffers to provide complete IOMMU protection. With this configuration, we no longer observed the leakage.

As shown in Figure 5, the proposed mitigation combined with software bounce buffers mitigation (HW-INV+SWBB) achieved a 10.2% throughput gain compared to the combination of IOMMU strict mode and SW bounce buffers (Strict+SWBB).

VII. Discussion

We demonstrated a proof-of-concept exploit for the IOMMU deferred invalidation vulnerability and its mitigation using a QEMU-emulated system. In this section, we briefly discuss considerations for adapting the exploit and mitigation to a real hardware system. Exploiting the deferred invalidation in a real hardware system can be demonstrated using an FPGA device (similar to Thunderclap [10]), which we leave for future work. The IOTLB size should be considered when reproducing our exploit in a real hardware system as the IOTLB size of a real system can be considerably smaller (32 [15]) than that of QEMU (1024). However, the malicious device can still keep a translation in the IOTLB by regularly using it so that it does not get evicted.

To implement the proposed mitigation in a real system, we need access to the IOMMU hardware design. However, this was not possible as there was no open-source IOMMU implementation available. Instead, we used QEMU for our implementation. Section VI details our experiments that show how the IO throughput performance of a QEMU-emulated system and a real system correlate.

VIII. Conclusion

We have demonstrated that the deferred invalidation vulnerability can be exploited with a DMA-capable malicious device. Operating systems use IOMMU strict invalidation mode to mitigate the deferred invalidation vulnerability at the cost of reduced IO throughput for IO-intensive workloads. We proposed an alternative, low-overhead mitigation for deferred invalidation vulnerability with minimal changes to the existing IOMMU hardware and OS software. Our proposed mitigation achieved 12.7% higher network throughput compared to strict invalidation mode in a QEMU-emulated system.

Acknowledgment

This work was funded in part by NSF Award 1916393.

References

[1] M. Becher *et al.*, "Firewire: all your memory are belong to us," in *CanSecWest Applied Security Conference*, 01 2005.
[2] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digit. Investig.*, 2004.
[3] G. Beniamini, "Over the air: Exploiting broadcom's wi-fi stack (part 2)," https://lwn.net/Articles/786558/, 2017.
[4] J. FitzPatrick and M. Crabill, "Stupid pcie tricks, featuring the nsa playset," in *DEFCON 22*, 2014.
[5] "Intel virtualization technology for directed i/o, revision 4.0," 2022.
[6] "Amd i/o virtualization technology (iommu) specification, rev 3.07," 2022.
[7] M. Malka *et al.*, "Riommu: Efficient iommu for i/o devices that employ ring buffers," in *ASPLOS '15*, 2015, p. 355–368.
[8] A. Markuze *et al.*, "True iommu protection from dma attacks: When copy is faster than zero copy," in *ASPLOS '16*, 2016, p. 249–262.
[9] A. Markuze *et al.*, "Damn: Overhead-free iommu protection for networking," in *ASPLOS '18*, 2018.
[10] A. T. Markettos *et al.*, "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," in *NDSS'19*, 2019.
[11] A. Markuze *et al.*, "Characterizing, exploiting, and detecting dma code injection vulnerabilities in the presence of an iommu," in *EuroSys '21*, 2021.
[12] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.
[13] B. Morgan *et al.*, "Bypassing iommu protection against i/o attacks," in *"LADC '16*, 10 2016, pp. 145–150.
[14] B. Morgan *et al.*, "Iommu protection against i/o attacks: A vulnerability and a proof-of-concept," *Journal of the Brazilian Computer Society*, vol. 24, 12 2018.
[15] N. Amit *et al.*, "Iommu: Strategies for mitigating the iotlb bottleneck," in *ISCA 2010*, 2010, pp. 256–274.
[16] M. Rybczyńska, "Bounce buffers for untrusted devices," https://lwn.net/Articles/786558/.
[17] D. Duplyakin *et al.*, "The design and operation of CloudLab," in *USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14.
[18] "Netperf – a network performance benchmark," https://github.com/HewlettPackard/netperf.
[19] J. Corbet, "Hot and cold pages," https://lwn.net/Articles/14768/, 2002.
[20] J. Axboe, "Flexible i/o tester," https://github.com/axboe/fio.