HyperHammer: Breaking Free from KVM-Enforced Isolation

Wei Chen
harlwei@outlook.com
Peking University
Beijing, China

Zhi Zhang*
zhi.zhang@uwa.edu.au
University of Western Australia
Perth, Australia

Xin Zhang
zhangxin00@stu.pku.edu.cn
Peking University
Beijing, China

Qingni Shen*
qingnishen@ss.pku.edu.cn
Peking University
Beijing, China

Yuval Yarom
yuval.yarom@rub.de
Ruhr University Bochum
Bochum, Germany

Daniel Genkin
genkin@gatech.edu
Georgia Institute of Technology
Atlanta, United States

Chen Yan
yanchen1302@gmail.com
Peking University
Beijing, China

Zhe Wang
wangzhe12@ict.ac.cn
SKLP, Institute of Computing Technology
Chinese Academy of Sciences & Zhongguancun Laboratory
Beijing, China

Abstract

Hardware-assisted virtualization is a key enabler of the modern cloud. It decouples virtual machine execution from the hardware it runs on, allowing increased flexibility through services such as dynamic hardware provisioning and live migration. Underlying this flexibility is the security promise that guest virtual machines are isolated from each other. However, due to the level of sharing between VMs, hardware vulnerabilities present a serious threat to this usage. One such vulnerability is Rowhammer, which allows attackers to modify the contents of memory to which they have no access. While the attack has been known for over a decade, published applications against such environments are limited, compromising only co-resident VMs, but not the hypervisor. Moreover, due to security concerns, a key component enabling their attack has been disabled. Hence, this attack is no longer applicable in a contemporary virtualized environment.

In this paper, we examine how Rowhammer can affect virtualized systems. We present HyperHammer, a Rowhammer attack that breaks hypervisor-enforced memory isolation and further compromises the hypervisor. Due to the highly specific system requirements for leveraging Rowhammer bit flips, HyperHammer is demonstrated on a very particular system configuration. Therefore, as demonstrated, HyperHammer is more a proof-of-concept than an immediate threat to computer systems. Nonetheless, our work demonstrates that hardware-assisted virtualization does not fully protect the hypervisor, and that with sufficient engineering a determined attacker might achieve complete hypervisor compromise.

CCS Concepts: • **Security and privacy → Virtualization and security**.

Keywords: Hardware-assisted Virtualization, KVM, Memory Isolation, Rowhammer, Hypervisor Compromise

ACM Reference Format:
Wei Chen, Zhi Zhang, Xin Zhang, Qingni Shen, Yuval Yarom, Daniel Genkin, Chen Yan, and Zhe Wang. 2025. HyperHammer: Breaking Free from KVM-Enforced Isolation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716002

*Corresponding authors.

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716002

1 Introduction

Hardware-assisted virtualization allows cloud providers to efficiently share physical hardware among multiple users while keeping their workloads isolated. In a typical installation, the resources of a *host* machine are shared among multiple *guest* virtual machines (VMs) or hardware-assisted virtual machines (HVMs), which are unaware they are not running directly on physical hardware. A *hypervisor* software, running on the host, manages the guest VMs and provisions the host's resources among them. Thus, the hypervisor controls the number of CPU cores that are allocated to each VM, the memory space it can access, and the hardware devices available to it. The operating system in the VM is oblivious to the actual resources available on the machine it runs on. Instead, it only manages the resources that the hypervisor supplies.

Implementations of hypervisors, such as KVM [32], Xen [6], or Hyper-V [28] use hardware features such as VT-x and VT-d [21] to provide efficient virtualization. These features provide services such as nested page translations that allow seamless isolation of the memory provided to guest VMs, or device pass-through, which allows VMs direct access to some of the host's hardware devices.

Since all guest VMs share the host's physical memory, maintaining isolation between VM address spaces is a critical responsibility of the hypervisor.

Currently, modern computers primarily use Dynamic Random-Access Memory (DRAM) for their main memory. DRAM, however, is known to be vulnerable to read disturbance attacks, such as Rowhammer [31] and Rowpress [37], where reading memory at specific access patterns can cause bit flips in memory locations that are not accessed at all. Rowhammer potentially affects all modern DRAM chips [11, 14, 18, 24, 29, 33].

The ability to modify memory contents without accessing them can have disastrous security implications. A large number of attacks have been demonstrated compromising system security [10, 13, 16, 17, 44, 46, 50, 54, 59, 61, 63]. All of these attacks involve a crucial step, called *memory massaging*, in which the attacker manipulates the system into placing sensitive objects, such as page tables, in memory locations that the attacker can manipulate via Rowhammer. Until now, all existing memory massaging techniques rely on memory management features, which are *unavailable* for exploitation in contemporary virtualized environments. To the best of our knowledge, only two attacks specifically target virtualized environments. Xiao et al. [59] focus on Xen paravirtualized environments, which are not commonly used in contemporary cloud environments. Razavi et al. [44] target hardware virtualized environments, but their work relies on a memory deduplication feature that has been long turned off in commodity hypervisors [54, 56]. Furthermore, their work targets co-resident VMs, rather than compromising the hypervisor. Given the lack of evidence showing how Rowhammer affects modern hypervisors, in this work we ask the following question: *Are modern virtualization platforms vulnerable to Rowhammer attacks?*

1.1 Our contribution

Our research demonstrates that modern virtualization platforms are indeed vulnerable to Rowhammer attacks. We present HyperHammer, an attack on a modern virtualized environment that builds on Rowhammer together with a combination of recent hardware-assisted virtualization features. We show that HyperHammer can compromise the isolation provided by KVM, a popular virtualization solution that is part of the Linux kernel. HyperHammer allows a malicious VM to escape KVM-enforced isolation and gain arbitrary access to the host's memory.

Our implementation of HyperHammer is tailored to a specific system configuration, including the use of the KVM hypervisor that supports transparent huge pages [12], implements the software countermeasure to the iTLB Multihit bug [19] and uses the `virtio-mem` driver. Moreover, the attack is statistical in nature, with a success probability that depends on the amount of host memory that is allocated to the VM. Even when allowing the VM to use most of the available memory, the expected runtime of HyperHammer is measured in months. Consequently, we do not view HyperHammer as an immediate threat to computer security. Instead, we use it as a case study to assess the amount of engineering required to achieve hypervisor compromise in hardware-assisted virtualization and to demonstrate that determined attackers can use Rowhammer to breach such systems.

From the technical side, HyperHammer follows the general structure of previous Rowhammer attacks, consisting of three main steps: memory profiling, memory massaging, and exploitation. However, in each of these steps, we need to overcome the unique challenges that the virtualized environment introduces. We now overview these steps, the challenges they introduce, and how we overcome them. Admittedly, while the challenges may be relevant in a wider context, the way we address them is highly specialized to the specific system configuration we use. Nonetheless, we believe that insights from the technical details may carry over to wider contexts.

Memory Profiling. The aim of memory profiling is to identify vulnerable bits and the exact conditions (timing, access patterns, and memory layout) required to trigger them. The main challenge here is that the attacker, which controls a malicious VM, cannot recover the physical addresses of the memory that the hypervisor allocates for it. Consequently, the attacker cannot make use of known memory mappings to construct efficient Rowhammer attacks [42, 57, 59]. We overcome this challenge by relying on the default use of transparent hugepages (THP) [12] in the hypervisor. With THP, the attacker can recover the 21 least significant bits of physical addresses, which are often sufficient for carrying out the profiling.

Memory Massaging. The aim of memory massaging is to place some sensitive data, in our case a page-table entry, in memory locations containing bits vulnerable to Rowhammer, allowing the attacker to change page mappings through bit flips. We present *Page Steering*, an attack technique that allows a VM to perform memory massaging.

The main challenge for Page Steering is that the hypervisor pre-allocates the address space of the VM, including all of its page tables, at VM creation time. To place a page-table entry on a vulnerable bit, we need a way of allocating new page tables at runtime. Our solution is to exploit a countermeasure for the iTLB Multihit bug [19], which affects many Intel processors.[1] When the countermeasure is triggered, the hypervisor changes the memory mapping of the VM, allocating a new page for the page table in the process.

Another challenge is that the memory address space of the VM is largely static. However, the attacker needs a way of releasing memory identified during profiling as vulnerable. We observe that while `virtio-mem` enables the hypervisor to set memory allocation targets, it does not enforce that VMs adhere to these directives, allowing a VM to release pages even when not requested by the hypervisor.

A third issue stems from the way that the kernel at the host manages free memory. Due to the use of the `virtio-mem` driver, the VM has to release at least 2 MB of memory. However, when allocating from the free list, the kernel at the host tends to prefer using smaller blocks. Before we can reuse the memory that the attacker releases, we need to get rid of pages in these smaller free blocks. For that, we exploit vIOMMU [4], an abstraction of the input/output memory management unit (IOMMU). Specifically, we create a large number of mappings from the IO virtual address space to a single memory page in the VM. Each of these mappings consumes an additional 4 KB page, allowing us to remove all small blocks from the free list.

Exploitation. The main aim of the exploitation step of HyperHammer is to change the page mappings so that the address space of the VM contains a page-table page. We achieve this by using Rowhammer to flip a bit in the page table, which the memory massaging step has placed in a vulnerable location. This allows the VM to change the mapping of its own address space to arbitrary physical memory pages, which provides full access to the physical memory of the host. The main challenge here is how to identify that the attack succeeds, which we overcome by introducing techniques for detecting page-table entries in the address space.

Summary of Contributions. In summary, in this paper, we make the following contributions:

- We present HyperHammer, a modern Rowhammer attack targeting virtualized environments (Section 4). We describe how to overcome the specific challenges that virtualization presents to memory profiling (Section 4.1), memory massaging (Section 4.2), and Rowhammer exploitation (Section 4.3).
- We implement HyperHammer on two machines, featuring consumer- and server-class processors with current versions of KVM and OpenStack. To the best of our knowledge, HyperHammer is the first Rowhammer attack effective in modern hypervisors, and the first to attack the hypervisor and not only co-resident VMs (Section 5).
- We perform extensive experiments, evaluating success rates and time requirements for the attack. We will make the source code of the attack available to the public.

[1]The latest microarchitecture with this bug is Comet Lake, launched in 2020 and still supported [19].

1.2 Responsible Disclosure

We reported our findings to the Linux Kernel, AWS, Google Cloud, and OpenStack on June 22, 2024. The Linux Kernel team confirmed that they would review kernel changes for HyperHammer on June 25, 2024. AWS confirmed that they had informed the relevant team for investigation on July 29, 2024. We developed a QEMU patch to mitigate HyperHammer and submitted it on November 26, 2024. Following a detailed discussion with the QEMU maintainer, we concluded that the proposed patch would impact the functionality of `virtio-mem` on November 30, 2024 [9].

2 Background and Related Work

In this section, we first briefly describe the Rowhammer attack. We then introduce preliminaries for Page Steering.

2.1 Rowhammer Attack

Dynamic random-access memory (DRAM) is the standard technology used for main memory in modern computers. When a DRAM row's cells are activated within the refresh period, it affects the amount of charge in their adjacent rows' cells, producing a circuit-level interference. While each individual charge disturbance is small, repeated disturbances can accumulate over time, eventually causing bits to flip in adjacent memory cells, changing their stored values from 0 to 1 or vice versa.

Rowhammer is such a circuit interference that causes bit flips in DRAM cells. It was first published in 2014 [31], with the key observation that activating rows can cause charge leaks to their adjacent rows. When a row is activated frequently, it can cause bit flips in adjacent rows. However, not all DRAM rows are prone to Rowhammer. We refer to the rows that can experience bit flips as *victim rows*, and those that are activated to trigger bit flips as *aggressor rows*. Physical pages on victim rows or aggressor rows are called *victim pages* or *aggressor pages*, respectively.

Prior to our work, the most relevant attack that exploits Rowhammer against hardware-assisted virtualization was demonstrated by Razavi et al. [43] They abuse the page deduplication feature to corrupt targeted files (e.g., OpenSSH public keys) residing in the page cache of a victim VM, breaking inter-VM isolation, and compromising the victim VM. To prevent this attack, page deduplication has therefore been disabled [54, 56].

In a broader context, Xiao et al. [59] used Rowhammer to compromise the Xen hypervisor from a paravirtualized VM (PVM). In Xen's paravirtualization, a PVM is modified to be aware of its execution in a virtualized environment. In particular, to manage the PVM's address translation, Xen uses a single level of page table hierarchy. In this approach, known as direct paging, the PVM is aware of the location of the page tables in memory. As a result, the attack deterministically flips a Page Middle Directory entry to point to a forged page table. This attack is clearly not applicable in the hardware-assisted virtualization scenario, which employs two levels of page table hierarchy, as discussed below.

2.2 Hardware-assisted Virtualization Paging

In a native environment, the CPU uses page tables to translate virtual addresses (VAs) to physical addresses (PAs). Within a VM, the guest manages mappings from *guest virtual addresses* (GVAs) to *guest physical addresses* (GPAs) in *guest page tables* (gPTs), while the hypervisor maintains an additional level of page tables called *extended page tables* (EPTs) that store mappings between GPAs to *host physical addresses* (HPAs). When the CPU accesses a GPA that has not been mapped before, a page fault occurs. In response, the hypervisor uses multi-level EPTs to establish the required memory mapping. There are two modes for multi-level EPTs, i.e., 4-level and 5-level EPTs. When a GPA does not have a valid mapping, *EPT entries* (EPTEs) are created across the levels of the EPTs. An EPTE is 64 bits in size and therefore each table contains 512 entries. In this paper, we focus on the leaf EPT pages under the mode of 4-level EPTs.

2.3 Linux Page Allocator

One of the major components of the Linux memory management subsystem is the buddy system, where pages are organized as *page blocks.* A page block contains 2^n consecutive pages where the block order n is a number between zero and an architecturally-specific maximum `MAX_ORDER`. On x86_64, `MAX_ORDER` is 11, so the largest page block on x86_64 contains 2^{10} pages. The kernel uses a data structure called *page list* to keep track of page blocks of the same order.

2.4 Page Migration Types

Page migration is a Linux feature that moves a page to another physical location while retaining its mapped virtual address. It is typically used to move pages to a NUMA node closer to the current process, thus reducing memory access latency. Not all pages in Linux can be migrated. Linux assigns each page a migration type that determines whether and how it can be moved in physical memory. In this paper, we focus on two types: `MIGRATE_MOVABLE` and `MIGRATE_UNMOVABLE`.

In the Linux buddy system, there are `MAX_ORDER` free lists for each migration type, with each list corresponding to a unique order. When pages in a free list are exhausted, Linux attempts to split page blocks of higher orders. If all pages of the desired migration type are exhausted, the system falls back to *stealing* pages of other migration types.

2.5 DMA and IOMMU

Direct memory access (DMA) allows peripheral devices to access memory without involving the CPU. This is useful for I/O-intensive devices, such as network interface cards (NICs) and GPUs. If a device is compromised or its driver is vulnerable, an attacker can exploit its DMA capability to access unauthorized system memory via physical addresses (PAs) directly, so-called malicious DMA access. To mitigate such attacks, the OS uses the *I/O memory management unit* (IOMMU), which works analogously to that of the memory management unit (MMU). That is, the OS maintains IOMMU page tables (IOPTs) that store mappings from *I/O virtual addresses* (IOVAs) to PAs. Devices can perform DMA only via IOVAs, which the IOMMU then translates to PAs.

2.6 PCI Device Assignment and VFIO

Hypervisors often emulate devices for the guest VMs. However, for better performance, the hypervisor can use *PCI device assignment* (a.k.a. PCI passthrough) to allow the VM to bypass the hypervisor and access the device directly. Because of the performance benefits, PCI device assignment is widely supported by modern cloud providers [3, 15]. To ensure logical isolation between the hypervisor and the VM, PCI device assignment relies on the IOMMU.

One of the consequences of PCI device assignment is that the hypervisor is not part of the communication between the VM and the device and is therefore not aware of outstanding DMA requests. To ensure the correctness of DMA requests, the hypervisor pins the pages of the VM and marks them as `MIGRATE_UNMOVABLE`, preventing them from being moved or swapped out.

IOMMU implementations can differ significantly across different architectures. As such, Linux introduces *Virtual Function I/O* (VFIO), which is an IOMMU/device-agnostic framework that abstracts away the differences. Since it exposes direct device access to user applications, it benefits both high-performance computing applications and VMs. For example, *Data Plane Development Kit* (DPDK) is a set of libraries and drivers designed to optimize packet processing on high-speed network interfaces.

3 Threat Model and Assumptions

Aligned with Flip Feng Shui [44], we assume that the attacker is a regular tenant of a public cloud and controls one VM. The attacker aims to flip exploitable bits in sensitive data structures, such as leaf EPTs. We further assume that the attacker does not have access to these sensitive data structures and that, due to the hidden mapping from guest to host addresses, the attacker is not even aware of the physical address in which these structures are stored. Finally, like all existing Rowhammer attacks, we assume that the physical memory is vulnerable to Rowhammer and that the software, including the OS and hypervisor, works correctly without any vulnerabilities.

In addition, we assume the attacker has been assigned at least one PCI device, e.g., a NIC, with vIOMMU enabled. We note that this is often supported in public cloud settings, such as OpenStack [40]. We also assume that the cloud enables memory overcommit to support on-demand memory allocation for VMs at runtime. The KVM hypervisor supports two overcommitment techniques: `virtio-mem` and `virtio-balloon`. In this work we mainly focus on `virtio-mem`, which is compatible with PCI device assignment to guest VMs, and assume that the hypervisor utilizes it. For completeness, we also discuss `virtio-balloon` in Section 6. Our attacker aims to compromise the hypervisors. Consequently, unlike Flip Feng Shui, we do not assume the existence of a co-resident victim VM.

Our proof-of-concept attack further relies on multiple features of the underlying hypervisor, including the use of transparent huge pages and the deployment of software countermeasures for the iTLB Multihit bug. While these features are fairly common, the overall combination of such features may not be very common. Thus, the threat of HyperHammer is mainly theoretical. The attack demonstrates that systems can be compromised, but is less likely to pose an immediate threat to real-world systems.

4 HyperHammer

We now describe HyperHammer, a proof-of-concept attack that shows how to use Rowhammer to achieve hypervisor compromise from an HVM. At a high level, HyperHammer consists of three main steps, similar to past works. First, the attacker profiles the memory to identify vulnerable bits, i.e., bits that can be flipped with Rowhammer. Then, the attacker manipulates the memory layout of the system to bring a sensitive page to the vulnerable location. Finally, the attacker uses Rowhammer to flip the bits and compromise the system.

While these steps are common to most Rowhammer attacks, their implementations differ. In HyperHammer the aim of the attack is to use Rowhammer to flip a bit in an EPTE, altering the mapping of a page in the attacker's VM to point to another EPT page. When achieved, this allows full access to the physical memory of the host system. To accomplish this, the attacker must navigate several challenges unique to the virtualization setting. Unlike processes in bare-metal operating systems, which can dynamically allocate and release memory within their virtual address space at runtime, the address space of VMs is typically pre-allocated at VM creation and can only change under extreme conditions. Consequently, VMs tend to have more limited control over their own memory allocation than is typically available to OS processes.

Additionally, memory management in hypervisors is more constrained, with allocations often occurring in larger chunks. For example, the `virtio-mem` driver, which we use, manages memory in chunks of 2 MB, whereas the `mmap` interface in operating systems has a granularity of 4 KB. The reduced granularity provides much lower flexibility to the attacker and complicates memory manipulation.

In the rest of this section, we describe how HyperHammer implements these steps and overcomes the challenges.

4.1 Memory Profiling

The first step of the HyperHammer attack is to profile the memory. Unlike prior works that just search for pages with vulnerable bits [44, 46, 52], HyperHammer has strict requirements on the bits that it aims to flip, mainly due to the use of the `virtio-mem` memory driver. When using `virtio-mem`, the hypervisor allocates and releases guest memory in chunks of 2 MB (termed sub-blocks in the `virtio-mem` nomenclature), which align with both the CPU's 2 MB hugepages and with order-9 page blocks in the buddy system of the hypervisor's Linux kernel.

To exploit a vulnerable bit, the attacker releases the page containing the bit to the hypervisor, while keeping the pages that contain the aggressor rows in its address space. However, the 2 MB chunks that `virtio-mem` manages typically contain multiple consecutive DRAM rows. Consequently, when releasing a 2 MB chunk that contains a vulnerable bit, the attacker relinquishes access to at least one of the adjacent memory rows, precluding the use of a double-sided Rowhammer attack [46]. Instead, we resort to using single-sided Rowhammer, where the attacker uses the two rows above or below the victim row.

A second challenge for profiling is that the hypervisor hides physical addresses from the attacker. To perform the attack, the attacker needs to access two adjacent aggressor rows within the same DRAM bank. Physical address bits determine the DRAM bank and row. Without the knowledge of physical addresses, the attacker needs a method for finding suitable aggressor rows. A naive solution for this issue is to brute force all possible rows. However, this can take a long time. To speed up this process, we rely on transparent hugepages (THP), a feature of the memory subsystem of Linux. Since THP can improve performance, particularly for memory-intensive programs, the KVM hypervisor enables it by default [12]. With THP enabled, the memory that the hypervisor allocates to the VM is likely to be backed by 2 MB hugepages. If the attacker configures the VM to use THP, there is a high likelihood that 2 MB hugepages in the VM are backed by 2 MB physical pages, which implies that the 21 least significant bits of a virtual address are preserved when converted to a physical address.

Prior works [39, 42, 57, 59] reveal that many x86 processors use the least significant 21 bits of the physical address to determine the corresponding DRAM bank. With THP, address translation preserves these bits, allowing the attacker to determine the DRAM bank that stores the contents of a memory address. We observed this property in both Intel processors we used in our experiments. (See Section 5.1.)

Knowing the DRAM bank mapping function, the attacker can pick two addresses that map to the same bank in two consecutive rows at the start or the end of a 2 MB hugepage as aggressor rows. The attacker then scans the memory to determine whether the attempt is successful, i.e., if a bit in a different 2 MB hugepage has flipped.

Even when the DRAM bank mapping function cannot be determined [42], the attacker can still gain some benefits from using THP. In such cases, the attacker can brute force all pages at the border of a 2 MB hugepage, without having to consider all rows. This will be relatively slower than the previous scenario by a factor that depends on the row size, but is still viable within a reasonable amount of time.

The final step in profiling is to filter exploitable bits. Recall that the attacker attempts to change the mapping of one of the VM's pages to point to an EPT page, which would allow the attacker full access to the physical memory of the host. For that, the attacker aims to change one of the PFN bits in an EPTE, i.e., bits 12–47 within a 64-bit entry in the page [22]. However, we note that flipping any of bits 12–20 still points to the same 2 MB hugepage, which is unlikely to contain an EPT. Moreover, flipping the high bits of the PFN will cause it to point outside the physical memory. Thus, the attacker needs only to focus on vulnerable bits that fall within the range 21–$\lceil \log_2(mem\_size) \rceil$.

4.2 Page Steering

The attacker now needs to release the vulnerable pages back to the hypervisor and coerce the hypervisor into reusing these pages to host EPTEs. To achieve this, we propose *Page Steering*, a novel method that takes advantage of memory reuse mechanisms in hypervisors. Since we target KVM, we exploit unique features of KVM to trigger page reuses within the VM. The process, as depicted in Figure 1, consists of three main steps.

First, the attacker needs to set up the system so an allocation of an EPT page uses the page the attacker relinquishes. One of the challenges for achieving this is that `virtio-mem`, which we use, manages memory in chunks of 2 MB, or order-9 blocks. Because the page allocation algorithm favors allocating smaller blocks, the attacker needs to first exhaust these blocks. To achieve this, the attacker exploits vIOMMU [4], a KVM feature that allows assigning devices to VMs.

Secondly, the attacker needs to relinquish memory pages containing vulnerable bits. The main challenge is that unlike processes, which can allocate and release memory at will, VMs have much less control of their address space. To allow the VM to release pages, we abuse the Virtio [53] interface that allows KVM hypervisors and VMs to negotiate the amount of memory allocated to the VM.

Last, the attacker needs to cause the hypervisor to place an EPT page on the vulnerable location. However, because

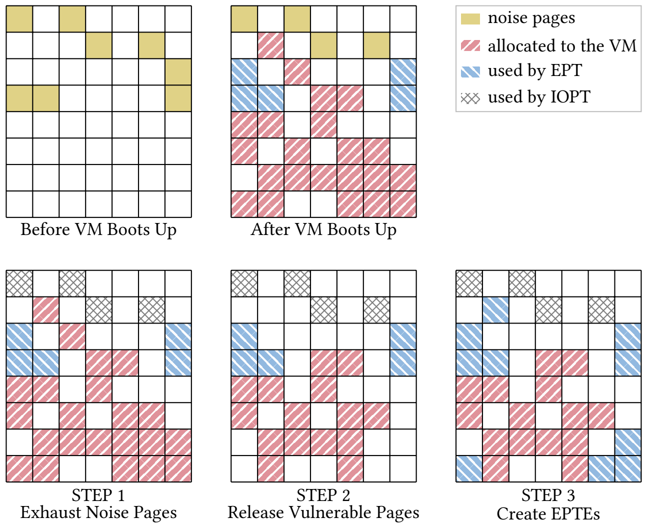


Figure 1. The process of Page Steering. Each block represents a page in the physical memory. After the VM boots up, a number of pages are allocated to the VM (▨), and a few more are used in the EPT (▨). We exploit the IOPT (▩) to exhaust noise pages (■; Section 4.2.1) that negatively impact the attacker's chance of success in STEP 1, and release vulnerable pages from the VM in STEP 2. Finally, in STEP 3, we trigger page reuses by creating many EPTEs.

the hypervisor allocates the page tables for the VM when the VM is initialized, there appears no need to create a new EPT page. We overcome this challenge by exploiting the countermeasure to the iTLB Multihit bug [19], which demotes 2 MB hugepages into 4 KB pages, creating EPT pages in the process.

4.2.1 Exhausting Free Lists.

In `virtio-mem`, memory is managed in units of sub-blocks, which are 2 MB chunks of memory. These typically consist of consecutive physical memory that is stored as order-9 blocks in the MM free list when released. EPT pages are of size 4 KB. Hence, when allocating them, the kernel prioritizes using small blocks from the free list. To increase the likelihood that EPT page allocations use the pages released by the VM, the attacker needs to first exhaust such pages. We refer to such pages as *noise pages* henceforth. An important observation that we make is the EPT pages are allocated as `MIGRATE_UNMOVABLE`. Recall that the kernel maintains a separate free list for each migration type, and allocates pages from the matching free list before changing the migration types of pages. Thus, we aim to exhaust small-order blocks in the free list of `MIGRATE_UNMOVABLE` pages. To achieve that, we exploit properties of vIOMMU [4], a hypervisor feature that virtualizes the IOMMU of the processor. This feature is primarily used for nested virtualization [4, 43] (VMs that provide virtualization

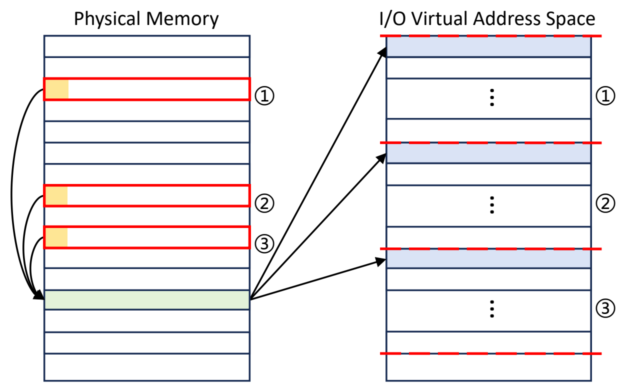


Figure 2. Using I/O virtual addresses (🟦) to exhaust small blocks from the `MIGRATE_UNMOVABLE` free list. The attacker allocates a single page (🟩) in its own virtual address space and creates multiple I/O mappings (🟨) to the page, forcing each of the mappings to consume an `MIGRATE_UNMOVABLE` page (red boxes). Each of these pages covers a 2 MB chunk (between two red dotted lines) in the IOVA space.

services) and DPDK [20] (applications that access devices directly for high-performance).

Exploiting the vIOMMU. To exhaust the free list of `MIGRATE_UNMOVABLE` pages, the attacker exploits the behavior of the vIOMMU, which manages IOMMU mappings on behalf of the VM when the VM requests memory mappings. These mappings are stored in order-0 `MIGRATE_UNMOVABLE` pages. Specifically, the attacker manipulates vIOMMU to populate its I/O address space. The approach is depicted in Figure 2.

The attacker allocates a single page in its own address space. It then uses vIOMMU to create multiple mappings from the I/O virtual address space to the allocated page. The attacker aims to force vIOMMU to allocate an IOPT page for each of these mappings. Similar to CPU page tables, each entry in IOPT translates the address of a 4 KB page. With 512 entries in an IOPT page, using virtual addresses that are 2 MB ($512 \cdot 4$ KB) apart ensures that each mapping uses a different IOPT page. Thus, by creating enough mappings, the attacker consumes all of the small-order `MIGRATE_UNMOVABLE` free blocks.

Two limitations of this approach are that a large I/O virtual address space is required for consuming a large number of `MIGRATE_UNMOVABLE` pages, and that, by default, vIOMMU only allows 65,535 mappings per IOMMU group. We note, however, that `MIGRATE_UNMOVABLE` pages are mostly used for data buffers, which are not too common. In our experiments, we find that under normal operations the number of `MIGRATE_UNMOVABLE` pages rarely exceeds 50,000. Moreover, we note that under some settings, the hypervisor assigns more than one device in different IOMMU groups to the VM (e.g., one GPU and several NICs, presumably provisioned through *SR-IOV*, While this is unlikely to occur in most real-world systems, it allows the attacker to consume more `MIGRATE_UNMOVABLE` pages under these settings.

When to stop consuming `MIGRATE_UNMOVABLE` pages. Our approach for exhausting the small-order free blocks does not provide the attacker with feedback on the status of the free list. Consequently, the attacker has no indication when all small-order blocks are consumed. If the attacker continues allocating IOPT after all small blocks are consumed, the host kernel splits larger, order-9 or order-10, blocks. This again adds 512 or 1,024 pages to the small-order lists. In Section 4.2.3 we describe how we handle these remaining small-order `MIGRATE_UNMOVABLE` pages.

4.2.2 Releasing Vulnerable Pages.

Because VMs often do not use all of the allocated memory space, cloud providers tend to over-commit memory, and dynamically adjust the amount of physical memory allocated to the VMs. While the hypervisor can unilaterally use swap space to reduce the amount of memory allocated to a VM, coordinating memory availability between the VM and the hypervisor yields better performance. This coordination is usually implemented through paravirtualized devices called *guest memory devices* (gMD), which facilitate memory negotiation between the host and the VMs. On QEMU/KVM, gMDs follow the Virtio specification [53]. For each VM, the hypervisor maintains configuration information that specifies the currently allocated memory size, targeted memory size, and maximum memory size allowed for allocation. When the hypervisor changes a configuration, the VM is notified and updates its memory settings accordingly. For example, to decrease the memory allocated for a VM, the hypervisor decreases the target memory size and notifies the VM. The VM then typically chooses a set of pages that it can relinquish and passes references of these pages to the hypervisor via the gMD. Upon receiving this list, the hypervisor looks up the physical pages and releases them back to the host. As mentioned in Section 3, the KVM hypervisor supports two implementations of gMD devices, `virtio-mem` and `virtio-balloon`. In this work, we exploit the `virtio-mem` implementation.

Voluntary Page Releases. In normal operation, when using a Virtio device, the hypervisor initiates changes in memory allocation. However, we observe that while the hypervisor can use `virtio-mem` to request memory changes, it does not enforce that the VM adheres to these requests. This means a VM can release memory pages even when not requested by the hypervisor or ignore hypervisor requests to release memory.

Page Steering, exploits this lack of enforcement to release pages that contain vulnerable bits. We make two modifications to the `virtio-mem` driver in the VM. First, we add the functionality of releasing desired pages to the host. Specifically, having identified a vulnerable page, the attacker converts its virtual address to a guest physical address. It then finds the `virtio-mem`'s identity of the page as block and sub-block numbers and uses the function `virtio_mem_sbm_unplug_sb_online` of the driver to relinquish the page. The driver at the VM communicates with the hypervisor, which identifies the physical address matching the sub-block, and uses the `madvise` system call to release the page to the kernel's free list. Due to the use of THP, the typical end result of this process is that the released memory is added to the KVM/Linux buddy system as an order-9 block of free memory, which is available for future allocation.

The second change we make in the `virtio-mem` driver is to avoid automatic memory allocations at the VM. Virtio is a collaborative protocol. The hypervisor sets the desired target allocation, and the VM issues memory requests to reach the target allocation. When the VM voluntarily relinquishes memory, it creates a discrepancy between the actual memory allocation and the target allocation. Under normal driver operation, the driver then notices that the current allocation falls short of the target, and immediately issues a request to allocate memory. As this request is served from the free list, executing it will return the recently freed sub-block to the VM. Our change prevents this undesired behavior.

4.2.3 Creating EPTEs.

So far, the attacker reduced the number of small-order `MIGRATE_UNMOVABLE` free blocks to below 1,024, and released a `virtio-mem` sub-block as an order-9 `MIGRATE_UNMOVABLE` free block. The remaining step is to allocate an EPTE on a vulnerable bit. The challenge is twofold. First, when using VFIO to assign a PCIe device to the VM, the hypervisor pre-allocates all of the address space of the VM. Thus, it is not clear how to create new EPT pages for the VM. Secondly, the new EPT page should be created on the page that contains a vulnerable bit. However, without knowing how many small-order blocks are in the free list, it is not clear how the attacker can manipulate the free list to achieve this. To overcome this challenge, we exploit the mitigation for the iTLB Multihit bug [19]. We first describe the iTLB Multihit bug. We then follow with a description of the countermeasure as implemented in KVM. Finally, we describe how Page Steering exploits the countermeasure.

iTLB Multihit Bug. To speed up address translation, the CPU caches recent translation results in the translation lookaside buffer (TLB). On recent Intel Core and Xeon processors, each of these TLBs is further divided based on the page size it translates. In particular, the processor supports separate instruction TLBs (iTLB) for regular pages of size 4 KB and for hugepages of size 2 MB. The processor queries both iTLBs concurrently during translation.

When the hypervisor or the OS needs to change a mapping and the size of a hugepage, they first change the page-table entries and then invalidate the TLB entry. On some Intel processors, a machine check error can occur when code requires address translation of the affected page while its stale iTLB entry hasn't been validated. This error typically results in system hangs or shutdowns. This opens up the possibility of a malicious VM launching a denial-of-service attack against the hypervisor.

Countermeasure. As a fix, KVM marks all mappings to hugepages that back VM addresses as non-executable (`NX`) by clearing bit 2 in the EPTE. Consequently, code in such pages cannot execute, and therefore the processor never creates an iTLB entry for the mapping. Because hugepages are marked as non-executable, attempting to execute code in a hugepage results in a page fault, which the hypervisor intercepts and then splits the hugepage into 512 4 KB pages, marks them as executable by setting bit 2 in their EPTEs, and resumes code execution. When execution proceeds from these 4 KB pages, the address translation results are stored in the 4 KB iTLB, which is not vulnerable to the Multihit bug. Despite the potential performance impact, KVM enables this countermeasure by default [51].

Exploit. In Page Steering, we exploit the observation that when splitting a hugepage into 4 KB pages, the hypervisor uses a new EPT page to store the mappings of the 4 KB pages. Thus, for the attack, we force code execution on a hugepage. This triggers the countermeasure for the Multihit bug, which allocates an EPT page when splitting a hugepage. Before allocating pages from the buddy system for EPT page creation, one source of noise pages is free page caches, such as the header page cache and the per-CPU pageset (PCP), which are prioritized when allocating pages. After that, the other source is the imprecision of our method for consuming small-order blocks (Section 4.2.1), which can leave up to 1,024 pages in the small-order lists. Finally, the attacker releases vulnerable hugepages of size 2 MB. These contain 512 4 KB pages, only one of which is vulnerable. Thus, even if we could precisely exhaust the small-order blocks, the likelihood that we would get the vulnerable location on the allocation of an EPT is very small.

To ensure that an EPTE is placed in a vulnerable location, we use a limited spraying approach. Specifically, instead of allocating a single EPT page, we allocate several hundred. That is, if we release N hugepages, we allocate at least $512 \cdot (N + 2)$ EPT pages. These first consume pages in the header page cache and the PCP, then pages from the remaining small-order free blocks in the buddy system, and finally all of the pages of the N hugepages we released. Thus, with a high likelihood, the vulnerable bits in the released hugepages will be allocated for EPTEs.

In practice, to allocate such a number of EPT pages, we allocate a large memory buffer. Due to the use of THP, the buffer is likely to be backed by 2 MB hugepages. Alternatively, because we control the VM, we can set up a large number of hugepages and use `mmap` to use them. In this buffer, we write the machine code of the function in Listing 1 before executing the code. The function basically does nothing. As splitting a hugepage creates one EPT page, to create 512 EPT pages we need 512 hugepages, or 1 GB of memory. Hence, to place EPTEs on vulnerable locations in N hugepages, the size of the allocated buffer needs to be $N + 1$ GB.

```
push %rbp
mov %rsp,%rbp
nop
... ; a number of `nop`s
nop
pop %rbp
ret
```

Listing 1. An Idling Function

4.3 Exploitation

After performing Page Steering, the system is in a state where some leaf EPTEs are placed in memory locations that contain bits vulnerable to Rowhammer. Our aim in this third and final step of HyperHammer is to exploit these vulnerable EPTEs for privilege escalation. For that, the attacker uses Rowhammer to flip the vulnerable bits and changes the physical page that a leaf EPTE points to. Privilege escalation is achieved when the changed mapping points to another leaf EPT page, allowing the attacker to modify it and access arbitrary physical memory locations.

Performing the Rowhammer attack is straightforward. The aggressor rows have not been released and are still within the address space of the VM. Thus, the attacker can repeat the patterns used during profiling to cause bit flips. For a successful attack, the attacker needs to achieve two aims. First, they should identify that the mapping has changed. Then, if the mapping changes, the attacker should identify whether the new mapping gives access to an EPT page.

Identifying Mapping Change. To identify that the EPT has changed, the attacker first marks memory pages with a magic value. By checking the magic value, the attacker can validate the mapping of the page address. If the Rowhammer attack is successful, the page that the new mapping points to no longer contains the correct magic value.

Identifying EPT Pages. With a mapping change, the attacker needs to check whether the new target of the modified mapping is an EPT page. The attacker first performs a sanity test, checking whether the contents of the page look like an EPT page. For this, the attacker scans every group of 8 bytes (matching an EPTE) on the page. The format matches if all of the bits in the eight bytes are zero or if the 8 bytes contain a large value, where at least one of the least significant 12 bits is not zero. If the format matches for all 512 groups of 8 bytes, the attacker proceeds to validate that the page is indeed an EPT page.

To validate that the page is an EPT page, the attacker modifies the EPTE entries one by one. After modifying each entry, the attacker again scans the address space checking for the presence of the magic values in pages. If after the change new mapping changes are detected, the attacker knows that the page is an EPT page. By changing the EPTE entries on the page, the attacker can now escape the VM and access arbitrary locations in the physical host memory.

Improving Success Rates. The HyperHammer attack can fail for several reasons. First, due to system noise, Page Steering may fail to place an EPT page on a vulnerable bit.

Second, even when an EPT page is placed on a vulnerable bit, it is not clear that the bit will actually flip. This can happen both because Rowhammer can sometimes be non-deterministic, but also because Rowhammer flips tend to be unidirectional, i.e., some bits flip from 0 to 1, whereas others from 1 to 0. So, depending on the EPTE contents, the vulnerable bit may or may not flip.

Third, even if the flip is successful, there is only a small probability that the physical page the entry points to after the flip is an EPT page. Each EPT page describes 512 memory addresses. In a hypervisor setting, the attacker cannot create multiple EPT mappings to the same physical page. Hence, in a typical system, there will be at least 512 pages for each EPT page. (At least because not all physical memory pages have EPTEs pointing to them.) Assuming that bit flips change the mapping to a random page, the probability of getting an EPT page is therefore less than $1/512$.

Finally, for a simple VM escape, the attacker requires that the EPT page it accesses describes the address space of its own VM. Otherwise, the attacker can change other VMs, but not access the modified mappings.

Due to the probabilistic nature of the attack, increasing the number of vulnerable bits that the attacker targets increases the likelihood of success. However, there is a limit to the number of vulnerable bits that the attacker can exploit. Recall that for each hugepage containing a vulnerable bit that the attacker releases to the hypervisor, the attacker needs to create 512 EPT pages. With each EPT page consuming 2 MB of address space, the attacker needs 1 GB of guest physical memory for each vulnerable bit released. Thus the maximum number of vulnerable bits the attacker can exploit is limited by the size of the guest physical memory.

A consequence of the limitation and the low success rate is that most attack attempts are likely not to be successful. Because the Page Steering step of conversion from hugepages to 4 KB pages is not reversible, when an attack attempt fails, the attacker needs to take the VM down and respawn it for another try.

5 Evaluation

We now evaluate HyperHammer on two different microarchitectures. The first is Intel Core i3-10100 (Ice Lake), intended for consumer-grade systems. The second is Intel Xeon E3-2124 (Coffee Lake). Both machines use two non-ECC 8 GB Apacer DDR4-2666 DIMMs (part number: `D12.2324WC.001`) and identical software configurations. We denote these machines as S1 and S2, respectively. The hypervisor runs Ubuntu 20.04 with kernel version 6.1.66 with the default memory configuration of THP enabled. The attacker HVM is configured with 4 vCPUs, 13 GB of memory and one NIC, running Ubuntu 22.04 with a modified kernel version 6.1.66. We allocate 12 GB of the VM's memory for memory profiling.

We further evaluate Page Steering within the popular platform OpenStack. Specifically, we use DevStack to deploy a single-node OpenStack instance (nova version 29.1.0) on the same physical machine as S1. It runs Libvirt 10.4 and QEMU 9.0 with the default configuration provided by DevStack. Both the host and HVM are running Ubuntu 22.04 with kernel version 6.1.66. We denote this setup by S3. In the following, we evaluate the main steps of HyperHammer.

Table 1. Results of Memory Profiling.

System	Time	Total	1⟶0	0⟶1	Stable	Expl.
S1	72 h	395	213	182	246	96
S2	48 h	650	329	321	40	90

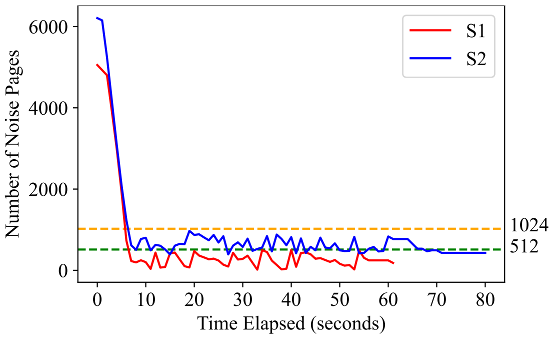
5.1 DRAM Memory Profiling

We profile the 12 GB of memory allocated to the attacker HVM to find vulnerable pages that contain bits susceptible to Rowhammer and their paired aggressor pages. We first verify that the bank functions of our processors only use bits preserved by hugepage mappings. For that, we use DRAMDig [57] to reverse engineer the DRAM address function of each processor. We find that the bank function of the Core i3-10100 processor uses address bits $(17, 21)$, $(16, 20)$, $(15, 19)$, $(14, 18)$, $(6, 13)$ to determine the bits of the bank number. Similarly, the Xeon E3-2124 uses $(17, 20)$, $(16, 19)$, $(15, 18)$, $(7, 14)$, $(8, 9, 12, 13, 18, 19)$. Because all of the bits used for the bank functions are preserved when translating virtual addresses of hugepages to physical addresses, we can use the 21 least significant bits of the guest physical address (GPA) to determine the bank used for storing the address.

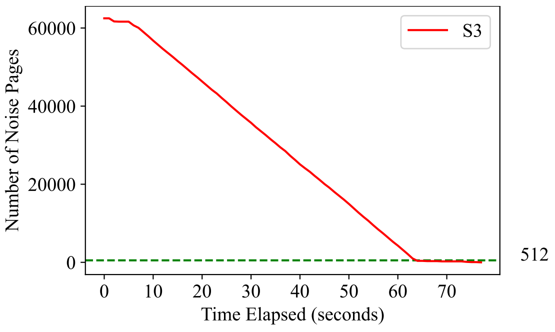
We further find that both processors use bits 18–33 to determine the row number. Thus each row spans over 256 KB and each 2 MB hugepage contains eight rows. While we cannot determine the row number from the GPA, we can determine the three least significant bits of the row number, specifying eight rows. This information allows us to identify the rows that are near the borders of the hugepage, which we can use as aggressor rows. We note that with 32 banks, two 4 KB pages of each row map to each bank.

Next, we use TRRespass[2] to identify an effective hammer pattern for the DIMMs. The results show that single-sided Rowhammer can trigger reproducible bit flips. To profile the memory, we try all combinations of aggressor row and bank. For each combination we first use the identified hammer pattern for 250,000 rounds. We then follow with a scan of all other 2 MB regions to detect bit flips.

[2]https://github.com/vusec/trrespass



(a) The number of noise pages at VM's runtime on S1 and S2



(b) The number of noise pages at VM's runtime on S3

Figure 3. The number of noise pages at VM's runtime. The green and orange dotted lines denote 512 and 1,024 pages respectively.

Table 1 summarizes the results of the profiling stage. Profiling on either machine takes multiple days and identifies hundreds of vulnerable bits. We note that only a fraction of these vulnerable bits are stable, i.e., they can be flipped reliably. Finally, we also check which of these bits is exploitable for the HyperHammer attack. Recall that for the attack we search for bit flips in bits $20–\lceil \log_2(mem\_size) \rceil$ of a PTE. (With 16 GB of memory, we have $\lceil \log_2(mem\_size) \rceil = 34$.) In total, we find 96 exploitable bits on S1, 46 that flip from 1 to 0, and 50 that flip from 0 to 1. On S2 we find a total of 90 exploitable bits, with 49 flipping from 1 to 0 and 41 flipping from 0 to 1.

5.2 Page Steering

We now evaluate the steps described in Page Steering on each of our settings.

Exhausting Noise Pages. In the VM, we allocate a page and map it to 60,000 IOVAs, starting from virtual address 0x1 0000 0000 at intervals of 2 MB. To demonstrate the decrease in the number of noise pages, we insert an artificial delay of one second after every 1,000 mappings. Concurrently with the creation of the mappings, we periodically sample the `/proc/pagetypeinfo` interface of the hypervisor to count the number of noise pages.

As Figure 3(a) shows, the number of noise pages on both S1 and S2 drops rapidly below the threshold of 1,024 noise pages. After the drop, the number of noise pages fluctuates between zero and the threshold. The third configuration, S3, starts with many more noise pages. (See Figure 3(b)). Consequently, the decrease in the number of noise pages takes much longer.

Releasing Pages from the VM and Triggering Page Reuses. The hypervisor is expected to reuse as many pages released by the VM as possible when allocating EPTs. As the VM's memory size caps the number of EPT pages that can be allocated, an increase in the number of sub-blocks released by the VM leads to a rise in the number of pages reused by the EPTs. This continues until all of the VM's memory is mapped. Beyond this point, the number of pages reused by the EPTs remains stable, as no additional EPT pages can be allocated.

To showcase the page reuses, we add two functions in the hypervisor. One function is to log PFNs of the pages that are released from the VM. The other function is to dump EPT pages in the system after EPTE creation is done. After releasing B page blocks from the VM and using a memory size of S for EPT creation to trigger vulnerable page reuses, we can use the two functions to calculate the number of released pages that end up in the EPT. The results are shown in Table 2. R_N is defined as the ratio between the number of pages reused by the EPTs (R) and the number of pages released by the VM (N). R_E is defined as the ratio between R and the number of EPT pages in the system (E). For all three settings, when S grows from 5 GB to 10 GB with N unchanged, both R_N and R_E grow significantly as we can create more EPT pages with a larger S. When N increases from 20 to 100 with S unchanged, R_E statistically grows as well.

5.3 Exploitation

We now turn our attention to the exploitation, assuming initially that the attacker successfully profiled the memory. Recalling that HyperHammer is probabilistic, we start with a theoretical analysis to determine a bound on the expected probability of attack success. We then experimentally validate the theoretical result. Finally, we conclude this section with an estimated time of a successful attack.

5.3.1 Analysis. Recall that for a successful attack, the attacker must first use Page Steering to massage an EPT page into a vulnerable position. The attacker then attempts to flip a vulnerable bit. The attack is successful if after flipping the bit, the new address in the EPT entry points to an EPT page. To bound the probability of success, we focus only on the last step, and assume that both Page Steering and the Rowhammer attack are successful.

Table 2. The number of pages that are released from the VM and the number of released pages reused by EPTs.

Setting	S	B	N	E	R	R_N	R_E
S1	5 GB	100	51200	3090	709	1.4%	22.9%
	10 GB	100	51200	5689	5194	10.1%	91.3%
	10 GB	70	35840	5690	4885	13.6%	85.9%
	10 GB	30	15360	5699	3339	21.7%	58.6%
	10 GB	20	10240	5633	2290	22.4%	40.7%
S2	5 GB	100	51200	2532	1943	3.8%	76.7%
	10 GB	100	51200	4868	4190	8.2%	86.0%
	10 GB	70	35840	4863	4363	12.2%	89.7%
	10 GB	30	15360	4857	3879	25.3%	79.9%
	10 GB	20	10240	4802	2449	23.9%	51.0%
S3	5 GB	100	51200	2817	1102	2.2%	39.1%
	10 GB	100	51200	5009	3900	7.6%	77.9%
	10 GB	70	35840	5113	3705	10.3%	72.5%
	10 GB	30	15360	5098	2679	17.4%	52.6%
	10 GB	20	10240	5106	1982	19.4%	38.8%

During Page Steering, the attacker releases a 2 MB hugepage that contains a single vulnerable bit. To place an EPT page at this vulnerable bit, the attacker needs to allocate 512 EPT pages. Each such allocation consists of converting one 2 MB page to 512 4 KB pages. Thus, Page Steering consumes 512×2 MB = 1 GB of memory, while it creates 512 EPT pages. The total number of EPT pages created is therefore limited by the size of the address space of the VM. At the same time, the total number of pages in the system is determined by the size of the memory. Consequently, the probability of success is roughly

$$\frac{\text{VM Memory Size}}{512 \times \text{Host Memory Size}}$$

Hence, statistically, at the limit, the attacker could succeed once every 512 attack attempts. Yet, more commonly, when the VM is allocated only a small part of the physical memory, the expected success rate can be much lower.

5.3.2 Experiment. To test our analysis, we profile the memory of the systems, recording the locations of vulnerable bits. We then repeatedly perform Page Steering, targeting 12 vulnerable bits at a time. (Our HVM uses 12 GB of memory for the attack and each vulnerable bit consumes 1 GB of that.) After each attempt, we scan the address space of the VM to detect any changes due to bit flips. If we identify a change, we check the page format to see if it matches an EPT page. If it matches, we change one of the entries to point to a hypervisor page that contains a magic value. We then scan the memory again, looking for the magic value. The attack attempt is considered a success if we find the magic value in the VM address space. If the attack attempt fails, we restart the VM and repeat the attack.

Table 3 shows the cost of the tests. Each attack attempt takes less than 5 minutes. On S1 we achieve success after 250 attempts and on S2 after 432. The experiment requires less

Table 3. The cost of HyperHammer tests.

Setting	Avg. Time for Single Attempt	Time for 1st Success	Attempts for 1st Success
S1	4.0 mins	16.7 hrs	250
S2	4.7 mins	33.8 hrs	432

than 34 hours on either machine. We note, however, that this timing does not include memory profiling. Instead, for this experiment, we implemented a hypercall to translate guest physical addresses to host physical addresses. This allows us to reuse profiling results without performing the profiling step for each attempt.

5.3.3 Expected Time for An End-to-End Attack. For an end-to-end attack, the attacker needs to perform the profiling for each attack attempt. As profiling time dominates the attack time, we now analyze the expected time required for a successful attack. Table 1 shows the time required to fully profile the VM memory. However, because the attacker can only release a small number of vulnerable bits, there is no need to generate a complete profile. Instead, the attacker can stop when enough bits, 12 in our case, are found. Thus, for S1, we can expect each profiling attempt to take $12/96 \times 72 = 9$ hours, whereas for S2, the expected time is $12/90 \times 48 = 6.4$ hours. Assuming 512 attempts are required for a successful attack, we can expect an end-to-end attack to succeed within $9 \times 512/24 = 192$ days on S1, and $6.4 \times 512/24 = 137$ days on S2. As mentioned in Section 5.3.1, the expected number of attempts depends on the portion of the host memory that is allocated to the attacker's VM. Hence, in the case that the VM is relatively small, the attack is likely to be much longer.

6 Countermeasure and Discussion

Quarantining VM Communications. Since the hypervisor does not check the semantics of the communications, an attacker can exploit gMDs to achieve Page Steering. Consequently, to defend a system against HyperHammer without disabling certain functions of the VM, one of the methods is for the hypervisor to quarantine communications initiated by the VM. It refers to the hypervisor's approach of detecting abnormal requests from the VM that display suspicious memory size change patterns and subsequently ignoring those requests. Let the target memory size configured by the hypervisor be defined as T, the current size of the VM's memory as V and the requested size of change as Δ. An attacker will initiate communications where $|\Delta| > |T - V|$ or $\Delta \cdot (T - V) < 0$. When the hypervisor notices such patterns, it defends itself by ignoring the request.

As such, we submitted a QEMU patch that quarantines the requests from VMs on November 26, 2024, where QEMU would respond with a NACK upon detecting a malicious memory-unplug request. However, the developer of `virtio-mem` pointed out that since the Linux driver does not expect a NACK during unplugging, this patch could lead to inconsistencies between the guest and the host. Additionally, when the Linux driver fails to plug in a memory block, it first unplugs the block and then retries—a behavior that, from QEMU's perspective, resembles a malicious unplug request. Therefore, fully implementing this countermeasure requires introducing a new feature flag in QEMU and updating both the Linux and Windows drivers.

Mitigating Rowhammer Attacks. Existing Rowhammer mitigations can generally be divided into two categories: software-only and hardware-based approaches. As the names suggest, software-only solutions [5, 7, 8, 34, 36, 50, 55, 58, 62] do not require hardware changes, thus being compatible with existing hardware. However, *none* of these works are effective in a KVM setting against HyperHammer. While [36] proposes the only solution that aims to protect EPTs from Rowhammer, the solution disables memory overcommitting, a common practice enabled by gMDs. As such, it is incompatible with our settings. Hardware-based solutions [25–27, 30, 31, 35, 38, 41, 45, 47–49, 60] aim to mitigate Rowhammer bit flips by tweaking hardware. Among them, ECC (Error Checking and Correcting) and TRR (Targeted Row Refresh) have been adopted by DRAM manufacturers in production. However, both production ECC and TRR have been reverse-engineered [11, 14, 23, 24].

HyperHammer's Limitations. As discussed in Section 5.3, the current implementation of HyperHammer is estimated to take 133–188 days to compromise the hypervisor, which is a considerable amount of time. The effectiveness of HyperHammer drops with the size of the VM. Cloud environments tend to allocate a much smaller portion of the host memory to each VM, which would increase attack time significantly. Additionally, HyperHammer is evaluated on small-scale Intel processors with non-ECC DIMMs, which differs from typical commodity servers that use Xeon scalable processors with ECC DDR4. Furthermore, the Intel processors need to be affected by the iTLB Multihit bug for the attack to be viable.

Prerequisites for HyperHammer. HyperHammer requires THP, the countermeasure for the iTLB Multihit bug, `virtio-mem`, VFIO, and vIOMMU. While the open-source KVM/OpenStack supports all these features, it remains unclear whether these are fully supported in commodity cloud environments. Specifically, THP is typically enabled by default in commodity clouds [1]. The `virtio-mem` driver is commonly used for memory overcommitment in virtualized environments. The countermeasure is available on certain Intel processors [19]. Both VFIO and vIOMMU are supported on specific EC2 instances, such as i3.metal [2], although vIOMMU is less frequently utilized in such environments.

Broader Implications of HyperHammer. HyperHammer reveals a vulnerability in the interaction between VMs and the hypervisor, specifically in how physical memory is allocated, managed, and reused within virtualized environments. One key insight from HyperHammer is that VMs can indirectly influence memory management in ways that may undermine security. In our attack, the attacker manipulates the memory allocation system within KVM by using specific hypervisor features, such as vIOMMU and the Virtio interface, to control the memory layout and force the reuse of pages. This process, while leveraging legitimate VM features, demonstrates how indirect control over memory—like relinquishing pages or exploiting device assignment—can have significant security implications in how hypervisors handle page mappings for sensitive data, such as EPT entries.

Our attack also highlights the risks associated with memory management decisions made at the hypervisor level, particularly when these decisions are not sufficiently isolated from the VM's control. For example, the use of 2 MB hugepages and the demotion to 4 KB pages as a countermeasure for the iTLB multihit bug illustrates how even seemingly harmless hypervisor optimizations or countermeasures can inadvertently create opportunities for exploitation. This shows that memory management mechanisms, when not carefully protected and segregated, can be abused to gain privileged access to the whole system.

Thus, the broader lesson of HyperHammer is that hypervisors should better decouple VMs from their management and, in particular, avoid relying on VM requests or actions for memory allocation, as this can provide attackers with a vector for manipulating critical system memory. In response to this lesson, we proposed the aforementioned countermeasure where the hypervisor actively quarantines or validates memory management communications initiated by the VM. The core principle behind this countermeasure is to introduce validation and monitoring of any VM requests that involve memory changes, reflecting the lesson above that hypervisors should maintain control and validation of memory allocation. Besides, we apply this lesson to our following analysis of `virtio-balloon`, another memory overcommit technique in KVM, and Xen, a widely-used hypervisor. Based on the analysis, we leave it to future work that focuses on the engineering efforts required to adapt HyperHammer to the `virtio-balloon` and Xen settings.

As introduced in Section 4.2.2, `virtio-balloon` allows the hypervisor to dynamically manage the VM's memory. Therefore, it also introduces potential risks, similar to those associated with `virtio-mem` in the HyperHammer attack. Unlike `virtio-mem`, when malicious a VM uses `virtio-balloon`, the attacker no longer needs to exhaust pages in low-order blocks, as `virtio-balloon` manages memory on a per-page basis rather than by memory blocks. However, without VFIO, the attacker must find alternative methods to convert the pages released by the VM to `MIGRATE_UNMOVABLE`. One potential exploitation involves using the `virtio-net-pci` driver to exhaust all existing `MIGRATE_UNMOVABLE` 4 KB pages in the system. Once these pages are exhausted, the system will be forced to *steal* `MIGRATE_MOVABLE` pages to satisfy new allocation requests, creating an opportunity for the attacker to manipulate the memory management system.

For the Xen hypervisor, it also supports memory ballooning and vIOMMU. Moreover, an attacker can proactively relinquish pages using the `XENMEM_decrease_reservation` hypercall, which releases pages to Xen via the `free_domheap_pages` function. Later, when Xen allocates pages for EPT, it calls `alloc_domheap_pages` and may reuse the pages previously released by the VM, as it does not differentiate between `MIGRATE_UNMOVABLE` and `MIGRATE_MOVABLE` pages. Consequently, launching Page Steering may be even easier on Xen than on KVM.

The broader lesson remains consistent: hypervisors must carefully scrutinize any memory management requests initiated by VMs. Whether through memory ballooning, device assignment, or other dynamic memory allocation techniques, hypervisors must implement strict memory validation to ensure that these requests do not introduce vulnerabilities.

7 Conclusion

In this paper, we present HyperHammer, an attack that breaks HVM isolation by tricking the hypervisor into placing EPTEs on victim pages and using Rowhammer to modify the EPT to access arbitrary memory. The key technique in HyperHammer is *Page Steering*, which provides powerful ways to force the hypervisor to reuse pages released by the VM. While HVMs are generally thought to be secure, our work underscores how hardware vulnerabilities threaten their security.

Acknowledgments

We would like to thank the anonymous reviewers for their advice and comments, which helped shape this paper. In particular, we are grateful to our shepherd, Nadav Amit, for the careful reading, the guidance, and the support.

This research was supported by the National Key R&D Program of China under Grant No. 2022YFB2703301; PKU-OCTA Laboratory for Blockchain and Privacy Computing; the Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0079; the Alfred P Sloan Research Fellowship; an ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022; gifts from Qualcomm and Zama; and the CAS Project for Young Scientists in Basic Research under Grant No. YSBR-118.

References

[1] Alibaba Cloud. 2024. THP-related performance optimization in Alibaba Cloud Linux. https://www.alibabacloud.com/help/en/ecs/transparent-huge-page-thp-related-performance-optimization-in-alibaba-cloud-linux-2

[2] Amazon. 2019. How to use DPDK on AWS EC2 instances and AWS-based container platforms. https://aws.amazon.com/cn/blogs/china/

how-to-use-dpdk-in-aws-ec2-instances-and-aws-based-container-platforms-i/

[3] Amazon. 2024. AWS Whitepaper: The Security Design of the AWS Nitro System. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/the-components-of-the-nitro-system.html

[4] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, Assaf Schuster, et al. 2011. vIOMMU: Efficient IOMMU Emulation. In *USENIX Annual Technical Conference*.

[5] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. 2016. ANVIL: Software-based Protection against Next-Generation Rowhammer Attacks. In *Architectural Support for Programming Languages and Operating Systems*. 743–755.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM symposium on Operating systems principles*. 164–177.

[7] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. 2019. RIP-RH: Preventing Rowhammer-based Inter-Process Attacks. In *Asia Conference on Computer and Communications Security*. 561–572.

[8] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security Symposium*.

[9] Wei Chen. [n. d.]. [PATCH] hw/virtio/virtio-mem: Prohibit unplugging when size <= requested. https://mail.gnu.org/archive/html/qemu-devel/2024-11/msg04881.html

[10] Yueqiang Cheng, Zhi Zhang, Surya Nepal, and Zhi Wang. 2019. Cattmew: Defeating software-only physical kernel isolation. *IEEE Transactions on Dependable and Secure Computing* (2019).

[11] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. 2019. Exploiting correcting codes: on the effectiveness of ECC memory against rowhammer attacks. In *IEEE Symposium on Security and Privacy*. 55–71.

[12] The Kernel Development Community. 2024. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html

[13] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: accelerating microarchitectural attacks with the GPU. In *IEEE Symposium on Security and Privacy*.

[14] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *IEEE Symposium on Security and Privacy*. 747–762.

[15] Google. 2024. Compute Engine Documentation: GPU platforms. https://cloud.google.com/compute/docs/gpus

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *IEEE Symposium on Security and Privacy*. 245–261.

[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Program for testing for the DRAM rowhammer problem using eviction. https://github.com/IAIK/rowhammerjs

[18] Hasan Hassan, Yahya Can Tugrul, Jeremie S Kim, Victor Van der Veen, Kaveh Razavi, and Onur Mutlu. 2021. Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications. In *IEEE/ACM International Symposium on Microarchitecture*. 1198–1213.

[19] Intel. 2019. Machine Check Error Avoidance on Page Size Change. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/advisory-guidance/machine-check-error-avoidance-page-size-change.html

[20] Intel. 2019. Memory in DPDK Part 2: Deep Dive into IOVA. https://www.intel.com/content/www/us/en/developer/articles/technical/memory-in-dpdk-part-2-deep-dive-into-iova.html

[21] Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[22] Intel. 2023. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[23] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. 2022. BLACKSMITH: Scalable Rowhammering in the Frequency Domain. In *IEEE Symposium on Security and Privacy*.

[24] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *USENIX Security Symposium*.

[25] JEDEC. 2012. DDR4 SDRAM Specification. https://xdevs.com/doc/Standards/DDR4/JESD79-4%20DDR4%20SDRAM.pdf

[26] JEDEC Solid State Technology Association. 2015. LOW POWER DOUBLE DATA RATE 4 (LPDDR4). https://www.jedec.org/standards-documents/docs/jesd209-4b

[27] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. 2023. CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer. In *IEEE Symposium on Security and Privacy*. 236–252.

[28] Jason A. Kappel, Anthony T. Velte, and Toby J. Velte. 2009. *Microsoft Virtualization with Hyper-V: Manage Your Datacenter with Hyper-V, Virtual PC, Virtual Server, and Application Virtualization.* McGrow Hill.

[29] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. 2020. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *International Symposium on Computer Architecture*.

[30] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W Lee, and Jung Ho Ahn. 2022. Mithril: Cooperative Row Hammer Protection on Commodity DRAM Leveraging Managed Refresh. In *IEEE High Performance Computer Architecture*.

[31] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture*. 361–372.

[32] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux Virtual Machine Monitor. In *Ottawa Linux Symposium*, Vol. 1. 225–230.

[33] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. 2022. Half-Double: Hammering From the Next Row Over. In *USENIX Security Symposium*.

[34] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. 2018. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *Operating Systems Design and Implementation*. 697–710.

[35] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. 2019. TWiCe: preventing row-hammering by exploiting time window counters. In *International Symposium on Computer Architecture*. 385–396.

[36] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. 2023. Siloz: Leveraging DRAM Isolation Domains to Prevent Inter-VM Rowhammer. In *Symposium on Operating Systems Principles*. 417–433.

[37] Haocong Luo, Ataberk Olgun, Abdullah Giray Yaglikçi, Yahya Can Tugrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. 2023. RowPress: Amplifying Read

Disturbance in Modern DRAM Chips. In *International Symposium on Computer Architecture*. 28:1–28:18.

[38] Michele Marazzi, Patrick Jattke, Solt Flavien, and Kaveh Razavi. 2022. PROTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *IEEE Symposium on Security and Privacy*.

[39] Seaborn Mark. [n. d.]. How physical addresses map to rows and banks in DRAM. http://lackingrhoticity.blogspot.com.au/2015/05/how-physical-addresses-map-to-rows-and-banks.html

[40] OpenStack. 2024. Attaching physical PCI devices to guests. https://docs.openstack.org/nova/latest/admin/pci-passthrough.html

[41] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. 2020. Graphene: Strong yet Lightweight Row Hammer Protection. In *IEEE/ACM International Symposium on Microarchitecture*. 1–13.

[42] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*. 565–581.

[43] QEMU. 2023. Features/VT-d. https://wiki.qemu.org/Features/VT-d

[44] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*. 1–18.

[45] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J Nair. 2022. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In *Architectural Support for Programming Languages and Operating Systems*. 1056–1069.

[46] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA'15*.

[47] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2016. Counter-based tree structure for row hammering mitigation in DRAM. *IEEE Computer Architecture Letters* 16, 1 (2016), 18–21.

[48] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. 2018. Mitigating wordline crosstalk using adaptive trees of counters. In *International Symposium on Computer Architecture*. 612–623.

[49] Mungyu Son, Hyunsun Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Making DRAM stronger against row hammering. In *Design Automation Conference*. 1–6.

[50] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *USENIX Annual Technical Conference*.

[51] The Kernel Development Community. 2018. iTLB multihit — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/admin-guide/hw-vuln/multihit.html

[52] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. 2022. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *IEEE Symposium on Security and Privacy*.

[53] Michael Tsirkin and Cornelia Huck. 2022. Virtual I/O Device (VIRTIO) Version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html

[54] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security*. 1675–1689.

[55] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 92–113.

[56] VMware. 2014. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735). https://kb.vmware.com/s/article/2080735

[57] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. 2020. DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping. In *Design Automation Conference*.

[58] Xin-Chuan Wu, Timothy Sherwood, Frederic T. Chong, and Yanjing Li. 2019. Protecting Page Tables from RowHammer Attacks Using Monotonic Pointers in DRAM True-Cells. In *Architectural Support for Programming Languages and Operating Systems*. 645–657.

[59] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium*. 19–35.

[60] A Giray Yağlıkçi, Minesh Patel, Jeremie S Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. 2021. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *IEEE High Performance Computer Architecture*. 345–358.

[61] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. PThammer: Cross-user-kernel-boundary rowhammer through implicit accesses. In *IEEE/ACM International Symposium on Microarchitecture*. 28–41.

[62] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Nepal Surya, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. 2022. SoftTRR: Protect Page Tables Against RowHammer Attacks using Software-only Target Row Refresh. In *USENIX Annual Technical Conference*.

[63] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Dongxi Liu, Kang Li, Surya Nepal, Anmin Fu, and Yi Zou. 2022. Implicit Hammer: Cross-Privilege-Boundary Rowhammer through Implicit Accesses. *IEEE Transactions on Dependable and Secure Computing* (2022).