GPUBreach: Privilege Escalation Attacks on GPUs using Rowhammer

Chris S. Lin, Yuqin Yan, Guozhen Ding, Joyce Qu, Joseph Zhu, David Lie, Gururaj Saileshwar
University of Toronto

***Abstract*—NVIDIA GPUs with GDDR memories have been shown susceptible to Rowhammer-based bit-flips, similar to CPUs. However, Rowhammer exploits on GPUs have been limited to injecting untargeted bit-flips in victim data like weights of machine learning models, to degrade model accuracy, unlike CPU exploits shown capable of privilege escalation.**

In this paper, we demonstrate that GPU Rowhammer exploits can be as potent as CPU Rowhammer attacks. By exploiting the GPU page table management to identify when and where new page tables are allocated, we enable an unprivileged user CUDA kernel of one process to use RowHammer bit-flips to gain access to the GPU memory of other processes or co-tenants via targeted tampering of such page-tables resident on the GPU memory. Using this newly found primitive, we demonstrate the first GPU-side privilege escalation attacks, leaking secret data such as cryptographic keys from cuPQC libraries, and even tampering with the model's GPU assembly code to degrade models more stealthily than previous attacks. We further demonstrate that GPU-side privilege escalation can lead to CPU-side privilege escalation, defeating the protections provided by the IOMMU, enabling a malicious user-level program with GPU access to gain root shell and system-wide control, even in a non-multi-tenant setting.

1. Introduction

Modern DRAM is vulnerable to Rowhammer attacks [1], where an attacker can induce bit-flips in memory cells by repeatedly activating neighboring rows. Rowhammer has been extensively studied on CPU DRAM (DDR3–5 and LPDDR4–5) across Intel, AMD, and ARM CPUs [2]–[10], and more recently on GPUs with GPUHammer [11] demonstrating bit-flips on GDDR6 memories. While CPU based exploits have been extensively studied, and shown capable of weakening cryptographic keys [12]–[14], corrupting `sudo` to gain supervisor privileges [6], [15], or tampering with page tables to obtain kernel-level privileges [2], [4]–[6], [16], [17], there has been a limited study of Rowhammer exploits on GPUs. So far, GPU Rowhammer exploits have been only shown to have limited capabilities like tampering with co-resident ML model weights, to degrade model accuracy [11] or jailbreak large language models [18], making it unclear if they can be as potent as CPU-based exploits.

This paper explores whether privilege escalation exploits are feasible on NVIDIA GPUs using Rowhammer. On NVIDIA GPUs, user code runs as CUDA kernels that allocate memory resident in the GPU. Kernels from different processes (CUDA contexts) can run in a time-sliced manner in a single-tenant GPU (e.g., workstation), while kernels from different users can run concurrently, spatially sharing the GPU, via NVIDIA's Multi-Process Service (MPS) in containerized deployments [19], [20]. A CUDA kernel with elevated privileges can arbitrarily read or modify another process's GPU memory (e.g., leak or tamper with cryptographic keys, ML model weights, etc.). Worse, it may potentially corrupt the CPU-side data of the GPU driver, that executes on the CPU at an elevated privilege, enabling system-wide privilege escalation even on a single-tenant machine. Thus, GPU privilege escalation can have serious implications in both single and multi-tenant settings.

Privilege escalation exploits on CPUs [7], [16] use Rowhammer to corrupt privileged binaries (e.g., *sudo*), or to tamper with page tables. While GPUs do not execute privileged binaries like *sudo*, GPU page tables are stored entirely in GPU memory, making them appealing targets for GPU-based Rowhammer. Here, the attacker's goal is to gain arbitrary read/write access to GPU physical memory by corrupting their own GPU page tables (PT). Achieving this requires: (1) a page table entry (PTE) to be tampered with a bit-flip, and (2) the corrupted page-frame number (PFN) to point to another page table page. This enables the attacker's CUDA kernel to read and write its own page table, controlling its GPU virtual-to-physical mappings and enabling arbitrary GPU memory access. However, obtaining this level of control over GPU page tables is nontrivial and faces several challenges (**C1–C3**).

C1. Where are GPU Page Tables Allocated? Limited documentation exists on where NVIDIA GPUs place page tables in device memory. For Rowhammer based PTE tampering to work, a page table page needs to be sandwiched between two user data pages in neighboring rows. If the GPU allocates page tables in a separate memory region, then the attack cannot succeed. Thus, we must first determine how NVIDIA GPUs allocate page tables in memory to ensure interspersed allocation of data and page table pages.

C2. Populating GPU Memory with PTEs. CPU-based attacks often use `mmap` to map a single physical page to many virtual pages, forcing the memory to fill with page tables. However, the equivalent in GPUs (`cuMemMap`) fails to achieve the same effect, making this strategy ineffective. We thus require new GPU mechanisms to efficiently fill PTEs in memory, while minimizing the data memory requirements.

C3. Positioning GPU PTEs at Target Locations. A successful exploit requires not only flipping a bit in PTEs, but

also ensuring the corrupted PFN points to another page-table page with valid PTEs. This requires placing a page-table page precisely at the location that a tampered PFN will reference. How do we reliably steer GPU page-table allocations to these target physical addresses?

In this paper, we develop techniques to overcome these challenges and successfully launch Rowhammer-based privilege escalation exploits on NVIDIA GPUs. Similar to prior work [11], we demonstrate these techniques on an NVIDIA RTX A6000 GPU with GDDR6 memory, that is widely available in workstations and cloud servers, and shown to be vulnerable to Rowhammer bit-flips.

Uncovering Memory Locations of GPU Page Tables. Building on prior reverse engineering of GPU TLBs [21], we patch the open-source NVIDIA GPU driver to dump the memory locations of GPU page tables and analyze its allocation strategy. We find that the driver places all page tables in contiguous 2MB regions, which we call *page table regions* (PT regions). However, the initial PT region is far from user data. Only after this initial region fills, will a new region be allocated from the same physical memory pool as user pages, enabling interspersed allocations required for a Rowhammer exploit. Thus, we need primitives to efficiently fill PT regions with PTEs, with minimum memory usage.

Efficient Allocation of GPU Page Table Pages. To reliably induce bit-flips in PTEs, we need the ability to (1) allocate new PT regions at chosen locations, by filling PTEs in existing PT regions, and (2) densely pack PT regions with valid PTEs for effective Rowhammer targeting. While GPUs support multiple data page sizes (2 MB, 64 KB, and 4 KB), the NVIDIA driver automatically promotes allocations to 2 MB pages [21], without providing the user the capability to request specific page frame sizes. Consequently, allocating even a single new 2 MB PT region naively consumes roughly 256 GB of memory, far beyond GPU capacity. To address this, we develop new allocation primitives using Unified Virtual Memory (UVM) that coerce the allocator into producing desirable 64KB and 4KB page frames. Using these primitives, we can allocate new PT regions while consuming only 1 GB of memory and densify existing PT regions for effective Rowhammer targeting.

Side-Channel for Detecting Page Table Allocations. To position PT pages at desired locations, the attacker must detect *when* a new PT region is allocated. To that end, we uncover a *new* timing side-channel from UVM allocations that trigger evictions to CPU memory, when the GPU is near capacity. By keeping GPU memory just below full and issuing UVM allocations, the attacker can observe which allocations cause an eviction due to the creation of an extra PT region, thus detecting new PT region allocations. By controlling which memory region is freed, the attacker can guide page-table placement to the desired physical address.

Exploitation. With these techniques, we not only massage a page table page into a location with a known bit-flip, but also a second page table page at the address referenced by the corrupted PFN. Thus, the attacker gains control of its own page table, and gets arbitrary read/write privilege on the GPU memory. We use this capability to demonstrate three real-world exploits on a GPU that is temporally shared by kernels of different user processes.

First, we demonstrate that NVIDIA's cuPQC, a post-quantum cryptography (PQC) library using GPU acceleration, that stores its keys in GPU DRAM during operations such as key-exchange, can have its secret keys leaked by an attacker that runs in a time-sliced manner with a victim.

Second, we demonstrate code tampering in PyTorch ML model inference. By tampering with a single branch in the SASS (GPU assembly) of NVIDIA's cuBLAS library in GPU memory, an attacker can universally degrade model accuracy from up to 80% down to 0%, more stealthily than prior works that tamper with model weights [11], [22]–[24]. We also show leakage of LLM weights from GPU memory.

Third, we leverage the compromised GPU to issue malicious DMA accesses to the CPU and tamper with the privileged GPU driver memory on the CPU. By corrupting implicitly trusted, GPU-writable data structures in the privileged driver, we induce out-of-bounds (OOB) accesses and arbitrary writes within the Linux kernel, even with standard defenses such as IOMMU enabled, [25]–[27] leading to system-wide privilege escalation and a root shell for the attacker. These results show that GPU Rowhammer attacks threaten not just multi-tenant settings, but all GPU users.

Overall, this paper makes the following contributions:

1) We demonstrate the first privilege escalation exploit on NVIDIA GPUs via page table tampering using Rowhammer, granting arbitrary GPU R/W privileges.
2) We uncover unknown GPU page table allocation behaviors and new side-channels that make page table massaging and tampering practical on NVIDIA GPUs.
3) Leveraging arbitrary R/W privileges on GPU memory, we demonstrate a wide range of exploits from stealing or tampering sensitive data (ML models, cryptographic keys) to system-wide privilege escalation.

Responsible Disclosure. We responsibly disclosed our exploit to NVIDIA on 11th November, 2025, and subsequently to Google, AWS, and Microsoft. Google awarded us a bug bounty award of USD 600. NVIDIA responded that they may update their Rowhammer security notice [28]. Our code is open-sourced at https://github.com/sith-lab/gpubreach.

2. Background

2.1. GPU Privilege Levels

Modern GPUs do not follow the traditional privilege-ring model of CPUs. Instead, their management is divided between proprietary user-level code, the kernel-mode driver on the CPU, and privileged on-device microcontrollers. Consequently, we define a party as *privileged* on the GPU if it possesses capabilities equivalent to the kernel-mode driver, such as arbitrary access to GPU memory or the ability to issue control commands. Next, we describe the role of the user-level runtime, the kernel-mode driver, and the on-device privileged hardware in NVIDIA GPUs.

User-Level Components. The application code, written in CUDA or other GPU programming frameworks, executes with the lowest privilege level on the GPU. Such code runs on the GPU's CUDA cores under isolation enforced by the driver and hardware. On the host side, user applications interact with the GPU exclusively through runtime and driver APIs, which mediate all resource allocation, kernel launches, and memory operations. On the device side, CUDA kernels are confined to GPU virtual address spaces set up by the driver, preventing them from accessing data of other processes or privileged data governing GPU functionality.

Kernel Driver. The GPU kernel-mode driver operates in the host's privileged (kernel) mode and acts as the primary mediator between the user and the GPU hardware. It manages GPU memory allocation, context switching, command submission, and I/O data transfers. Importantly, the driver controls the virtual memory management for the GPU for a given process: it allocates GPU page tables, establishes mappings between GPU virtual addresses and physical memory, and enforces process isolation on the GPU-side.

Privileged Hardware Components. Modern GPUs include privileged hardware components responsible for GPU management. The GPU System Processor (GSP) is an embedded co-processor on the device running trusted firmware for GPU initialization, resource management, and configuration. The GPU Memory Management Unit (GMMU) enforces virtual address translation, memory protection policies, and address space isolation between clients. These components communicate with the privileged driver via DMA to manage the execution environment of the GPU.

In this paper, we investigate whether a malicious CUDA code executing at the user-level on the GPU, can achieve privilege-escalation on the device-side, and access arbitrary VRAM of other processes (equivalent to the kernel mode driver), and worse, if it can escalate privileges to the OS-kernel level on the host side, to get system-wide control.

2.2. GPU Virtual Memory & Page Tables

Virtual Memory. On NVIDIA GPUs, each process running CUDA kernels has a distinct GPU context with a unique GPU virtual address (VA) space. At runtime, on accessing a GPU VA, the GMMU performs a virtual-to-physical translation using the GPU page table. Each GPU context has distinct page tables stored in the GPU memory. As shown in Figure 1, the NVIDIA GPU page table has a multi-level design, with page directories (PD3-PD0), page tables (PT), and page table entries (PTEs) like the CPU page table.

Page Tables. Modern GPUs, starting from the Turing generation (including the Ampere GPU we study), support multiple page sizes, including 2MB, 64KB, and 4KB page frames [21], [29]. For 2MB page frames, the translation is stored directly in 16B entries at the PD0 level. For 4KB and 64KB page frames, the 16B PD0 entry is divided into two 8B halves, each pointing to the last-level page table (PT). The PT sizes are 4KB and 256B for 4KB and 64KB page

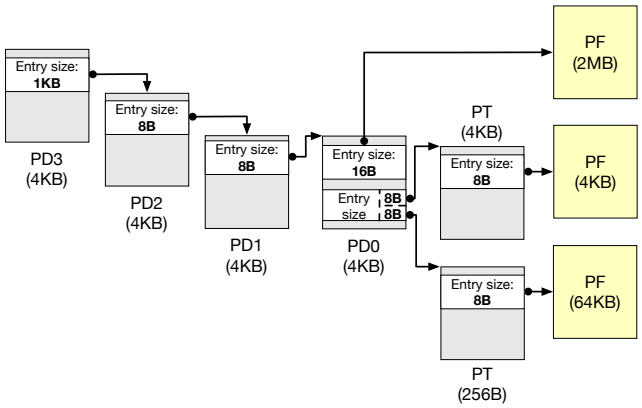


Figure 1: NVIDIA GPU Page Table. It has 4 or 5 levels, including PD3 to PD0 for all data pages, and a PT level only for 4KB and 64KB data pages.

frames, respectively. CUDA memory allocation APIs do not provide a way to request specific page sizes. The driver uses 2MB pages for `cudaMalloc`'ed pages and while it initially allocates smaller pages for unified virtual memory (UVM), these are also aggressively coalesced to 2MB pages [21].

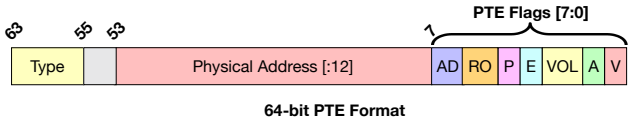


Figure 2: Format of GPU Page Table Entries. The bottom 8 bits are metadata flags, and the next 46 bits contain the page-frame-number.

Page Table Entries (PTEs). Figure 2 shows the format of a PTE in GPU Page Tables (PT) [30]. The 8th to 53rd bits store the page frame number, i.e., the physical address without the bottom 12 bits, as the minimum page size is 4KB. This gives us 58 bits of addressable physical memory. The PTEs also contain metadata flags; some notable ones are the RO (read-only bit), A (2-bit aperture, which indicates the memory domain for the page, 00 and 01 correspond to VRAM, and 10 and 11 correspond to coherent and non-coherent system memory), and P (privilege bit).

2.3. Unified Virtual Memory

In modern NVIDIA GPUs, unified virtual memory (UVM) enables a shared virtual address space between CPU and GPU. To enable access to this memory from both sides, the GPU driver transparently migrates physical pages between the GPU and CPU when accessed from the other. This avoids the application having to explicitly copying memory from CPU to GPU and vice versa, via `cudaMemcpy`.

UVM adopts a lazy loading strategy, where data is physically placed in GPU memory only when it is first accessed. When the total UVM allocated memory (via `cudaMallocManaged`) exceeds the GPU's physical capacity, the driver avoids out-of-memory errors by evicting

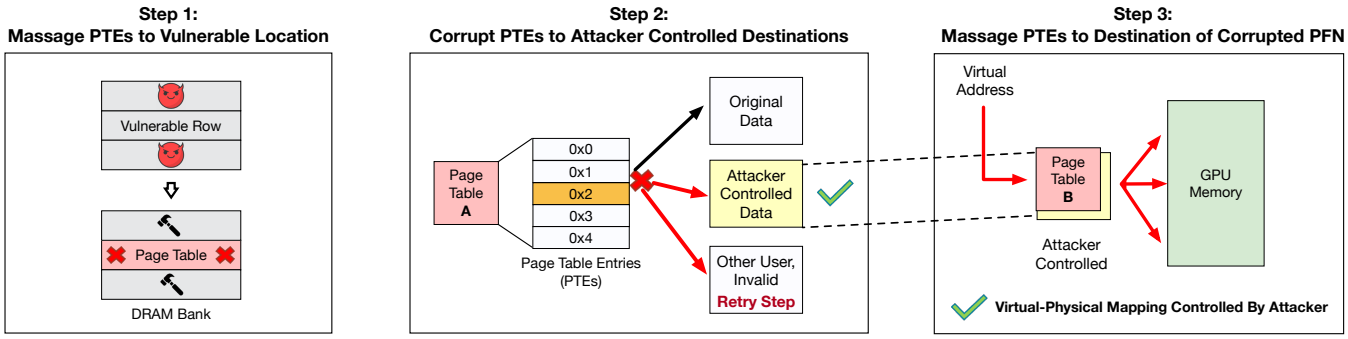


Figure 3: **Overview of the GPUBreach attack**. Steps to tamper GPU page tables with Rowhammer bit-flips to achieve GPU privilege escalation.

inactive GPU-resident pages back to host memory to make room for newly accessed data. The evictions of GPU resident pages occur in Least-Recently-Used (LRU) order.

2.4. Rowhammer

Modern DRAM stores data in cells, with one cell per bit, stored as electrical charge in capacitors arranged in rows and columns. The charge determines the cell's bit value, either 0 or 1. A data access to the DRAM involves activating its corresponding data row, and bringing it into a row buffer. Rowhammer is a read-disturbance phenomenon in which rapid activations of an aggressor row in DRAM causes charge leakage in an adjacent victim row, ultimately inducing bit-flips [1]. This vulnerability gets worse as DRAM scales to smaller technology nodes [31], and new variants of such data-disturbance vulnerabilities such as Rowpress [32], ColumDisturb [33], and Half-Double [5] have continued to surface in recent years. Prior works [2]–[8], [10], [31], [34], have shown that DDR3-5 and LPDDR4-6 memories, on Intel, AMD, and ARM CPUs are vulnerable to Rowhammer bit-flips, including those with on-die and system-level error correction codes (ECC).

GPU Rowhammer. While Rowhammer attacks have been extensively studied on CPU-based systems, recent research, GPUHammer [11], has shown its feasibility on GPUs. GPUHammer triggered bit-flips on NVIDIA GPUs such as RTX A6000 with GDDR6 memory, by reverse engineering the virtual-physical DRAM row mapping and developing specialized hammering patterns for GPUs that maximize the hammer intensity, while still being synchronized to refreshes [3] and many-sided [2] to bypass in-DRAM mitigations like TRR [35]. Using these bit flips, GPUHammer demonstrates untargeted flips in machine learning model weights [22], which significantly degrade model accuracy.

Compared to the large number of CPU-based Rowhammer exploits, that demonstrate privilege escalation, by tampering with page tables or instructions in *sudo* binaries [2], [4]–[6], [15]–[17] or compromising web browsers [3], [9], the capability of GPU Rowhammer attacks [11] has been limited to tampering with victim data such as ML model weights to degrade model accuracy. To that end, this paper studies the potential for GPU privilege escalation attacks, by utilizing Rowhammer bit-flips on GDDR memories.

3. Overview of GPUBreach & Challenges

3.1. Threat Model.

Setting. We target GPUs with GDDR6 memories, which have been shown vulnerable to Rowhammer in prior works [11]. Our primary focus is on GPUs natively connected to the host, like in workstations (used by video editors, gamers, etc.), or used in multi-tenant containerized settings (e.g., GPU shared between containers managed by Kubernetes [19], [20]). We demonstrate our attacks on the RTX A6000 GPU, for which open-sourced attack code is available [36], although our exploitation methods are generally applicable to any workstation GPU susceptible to bit-flips.[1] We do not focus on vGPUs or Multi-Instance GPUs (MIG), since they are primarily used in server-class GPUs with HBM2-3 memories that do not have demonstrated Rowhammer attacks as yet.

We assume Error-Correction-Code (ECC) is disabled on the A6000 GPU. This is reflective of real world usage, since we observe two cloud service providers (names redacted to protect customers) providing A6000 and A4000 GPU instances with ECC disabled by default. This is because enabling ECC induces up to 10% slowdown and 6% memory capacity loss on GPUs with GDDR memories, as shown in prior work [11], [37]. Users in single-tenant settings or those not running ML applications, may keep ECC disabled to avoid these overheads, as these settings have so far not been shown vulnerable to GPU Rowhammer exploits [11].

Attacker Capabilities. The attacker is assumed to have the capability to run user-level CUDA code. Knowing the target GPU model, the attacker can reverse engineer the virtual address to DRAM row mappings, craft attack patterns that hammer the GPU memory with the highest intensity, and defeat in-DRAM mitigations, as shown in prior

1. NVIDIA's security notice on Rowhammer [28] suggests that GPUs from multiple generations (Ampere, Ada, Hopper, Blackwell) may be affected. While we focus on the Ampere A6000 GPU, future works may develop attack patterns to defeat in-DRAM defenses on other newer GPUs.

work [11]. The attacker can then identify vulnerable DRAM rows with bit-flips on the victim GPU. The attacker's goal is to use these bit-flips to tamper with GPU page tables and escalate privileges on GPU-side to get arbitrary read/write access to GPU memory, to be able to steal or tamper with sensitive data on the GPU, or gain system-wide control.

3.2. Overview of the GPUBreach Attack

GPUBreach enables privilege escalation attacks on GPUs by tampering with GPU page tables using Rowhammer bit-flips. The crux of the attack is to inject bit-flips in the attacker's GPU PTE, specifically in the Page Frame Number (PFN) field, to redirect the attacker's virtual address to a page table page. As a result, the attacker gets the ability to fully modify their virtual to physical mappings, allowing it arbitrary read/write access to the entire GPU memory.

The attack consists of 3 key steps, as shown in Figure 3:

1) **Massage PTEs to a Vulnerable Location.** GPUBreach requires *massaging PT regions* to place them precisely at vulnerable memory locations. This is because naively using `cuMemMap` to duplicate virtual pages and fill the memory with PTEs, like prior CPU exploits that use `mmap` [16], only allocates up to 4 PT regions (<8 MB of PTEs) on the GPU, after which it crashes the program. Thus, *massaging* PT regions, i.e., precisely placing them in vulnerable memory locations, is needed, which requires uncovering the page table allocation algorithm, gaining control of a vulnerable memory page frame, and releasing it exactly when the page-table page is to be allocated, so that PTEs get allocated at the vulnerable memory location.
2) **Corrupt PTEs to Attacker Controlled Destinations.** Subsequently, the attacker hammers neighboring rows, to induce bit-flips in the PTE's PFN field. On the first attempt, the corrupted PFN's destination is unlikely to be a PT region, and is rather likely to be an invalid page, or even another user's page. Steps 1 and 2 are repeated until the corrupted PFN destination happens to be another data page controlled by the attacker.
3) **Massage PTEs to Destination of Corrupted PFN.** Once we identify that the corrupted PFN's destination is an attacker-controlled data page, this data page frame is freed and a PT region is *massaged* to this page frame, enabling the attacker to control PTEs, resulting in GPU privilege escalation.

A key requirement for this attack is being able to precisely massage page table entries to desired locations, in both Step 1 and 3. This introduces significant challenges.

3.3. Challenges in Page Table Massaging

Challenge 1: Unknown PT Allocation Algorithm. The ability to predict *where* page tables are allocated on GPUs is critical for any attempt to manipulate or massage their placement. Unlike CPU operating systems, with well-documented or reverse engineered MMUs, GPU drivers use proprietary and undocumented algorithms to allocate and organize page table pages in device memory. While prior work [21] demonstrated the capability to dump GPU page tables by instrumenting the GPU driver, key aspects such as their location in memory, allocation granularity, and memory reuse strategies remain unknown. To address this, we empirically study locations of page tables in GPU memory and develop mechanisms to influence their placement, in Section 4.3.

Challenge 2: Impractical Memory Requirements. On GPUs, memory is primarily managed in 2 MB page frames for data [21], [29]. We find that page table pages (page directory and page table) are also allocated within contiguous 2 MB regions, which we call *PT regions*. Hence, forcing the GPU to allocate a page table page at a specific location requires first exhausting the entire 2 MB PT region with page table pages. Given 16 B PTEs per 2 MB data page, this would require allocating roughly 256 GB of data, well beyond the capacity of even the latest NVIDIA Blackwell B200 GPUs (180 GB DRAM per GPU) [38].

While smaller page frame sizes exist for UVM (e.g., 64 KB or 4 KB), that would reduce this memory requirement, the driver provides no control to the application to select which page type should be allocated, and adopts proprietary algorithms to aggressively coalesce pages to larger 2 MB pages. To address this, we develop techniques to efficiently fill existing PT regions using smaller data page frames, and show *how* to allocate new PT pages at arbitrary locations under practical memory constraints, in Section 4.2.

Challenge 3: Allocations are Silent. To massage PT regions to the desired locations, the attacker needs to know *when* a new PT region allocation occurs, without any driver modifications. On CPUs, many OSes expose CPU page table memory consumption to users that can serve as an explicit signal for CPU PT allocations: for example, Linux reports leaf-level page table usage via the `PageTables` field in `/proc/meminfo` [39]. On GPUs however, the CUDA runtime or driver do not expose any such information about GPU page table memory consumption, preventing processes from knowing when page table allocations occur. To address this, we demonstrate a new timing side channel in UVM memory allocations, which can reveal when PT region allocations occur, in Section 4.4.

4. Techniques for Page Table Massaging

To address the above challenges in page table massaging and understand *where, how,* and *when* page tables can be allocated, we develop the following techniques.

4.1. Inspecting Page Tables

To understand GPU page table layouts, we instrument the NVIDIA GPU driver, like prior work [21], and dump the entire physical memory, and inspect page table locations for a GPU process. All our analysis is executed on the NVIDIA driver version 580.95.05 (released a few months ago), but

we confirm the same behavior on a wide range of major driver versions (545.23.08-580.95.05).

By default, GPU page table pages are allocated in **2 MB PT regions** away from data. To allocate new PT regions at vulnerable locations, we need to fill existing PT regions, without impractical data memory usage.

4.2. Memory-Efficient PT Region Allocation

A naive approach to populate the 2 MB PT region with PTEs by allocating 2 MB data page frames, the default for `cudaMalloc`, is impractical. As shown in Table 1, with 2 MB page frames, each PD0 entry occupies 16 B, yielding a data-to-PTE memory ratio of 128K : 1, requiring around **256 GB** of data memory to fully populate a PT region.

While this overhead can be substantially reduced by using smaller page frames supported by UVM, the driver provides no control to applications over which page sizes are used on memory allocations. For instance, with 64 KB and 4 KB data page frames in UVM, each PT entry is 8 B, decreasing the data-to-PTE ratio to 8K : 1 and 512 : 1, respectively. This reduces the required data memory to **16 GB** and **1 GB**. Therefore, to populate PT regions efficiently, we develop techniques to coerce the driver into using 64 KB and 4 KB data pages for UVM allocations, without coalescing them into 2 MB pages.

TABLE 1: User memory required to fill a 2MB large page with PTEs, for different data page frame sizes.

Data Page Frame Size →	2MB	64KB	4KB
PTE Size	16B	8B	8B
Ratio of Data to PTE Bytes	128K : 1	8K : 1	512:1
User Memory for 2MB PTEs	256GB	16GB	1GB

Constraints. While populating PT regions, we must satisfy two opposing goals: (1) *memory-efficient PTE filling* for initial PT regions, minimizing the data memory needed, and (2) *high-density PTE filling* for the PT region mapped to the vulnerable Rowhammer location, maximizing the chance that a bit-flip targets a valid PTE. However, no prior work has demonstrated how to allocate 4KB GPU data pages with UVM, the most memory-efficient option. Moreover, prior work [21] shows that UVM allocated 64KB pages are automatically *merged* into a 2MB page once more than 16 out of 32 of their PD0 entries are used, resulting in PT regions being half-empty. Thus, naive filling of PT regions either waste memory (2MB frames) or yield sparsely populated page tables (64KB frames), as shown in Figure 4 (left). We address both these limitations next.

(1) Memory-Efficient PTE Filling. First, we develop techniques to fill the PT region with PTEs using 4KB page frames. We start by systematically analyzing the page-frame sizes used by the driver for UVM allocations. Figure 5 shows the page-frame sizes as UVM allocation size increases. We observe that for allocation sizes till 1MB, 64KB page frames are used, and then for 1MB to 2MB allocations, 2MB page frames are used. However, allocating 2MB+4KB memory, results in both a 2MB and a separate 4KB page, as illustrated in Figure 4 (bottom right). Moreover, successive allocations of 2MB + 4KB, result in new 4KB PT pages with a single valid PTE, corresponding to the 4KB page frame. Thus, we achieve a ratio of data (2MB + 4KB) to PTE (4KB) bytes of 513 : 1, allowing us to efficiently fill the 2MB PT region using 1GB of UVM allocated memory.[2]

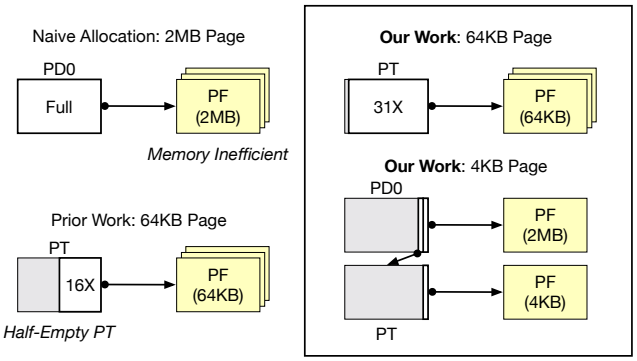


Figure 4: Page Table Filling Techniques. Prior techniques are memory-inefficient or leave PTs half-empty. We uncover two techniques that address both deficiencies.

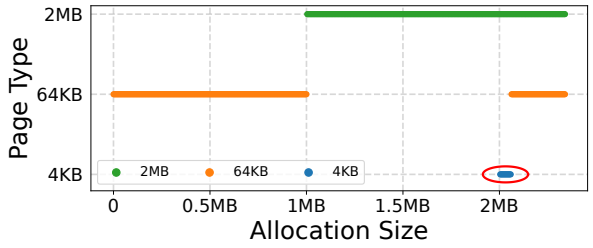


Figure 5: Page types used as UVM allocation size vary. From 2MB + 4KB to 2MB + 64KB, 4KB pages are used.

(2) Ensuring High Page Table Density. 4KB page frames are useful for filling initial PT regions, but not for filling PT regions at vulnerable memory locations. This is because we see that one can only sparsely fill 16 out of 512 PTEs per PT page using 4KB page frames, before they start to get merged by the driver to 64KB pages. Although 64KB frames provide a denser PT population, prior work [21] shows that these can also fill only 16 of 32 entries in a PT page (Figure 4, bottom left), leaving it *half-empty* and reducing the chance that a bit-flip affects a valid PTE.

To address this, we introduce a technique that nearly fills the entire 256B PT page. We observed that GPU evicts

2. For the memory-efficient PTE filling procedure, for each 2MB + 4KB allocation, the 2MB portion can also be moved to the CPU side (where memory may be more abundant). This can reduce the GPU memory needed for this procedure to 2MB, suitable even for memory-constrained GPUs.

UVM memory to CPU in 64KB chunks. Hence, instead of directly allocating 64KB pages, we allocate a 2MB UVM page and then access a 64KB slice from the CPU, forcing that slice to be evicted to system memory. This causes the 2MB UVM page to be splintered into 32×64KB pages, with 31 remaining on the GPU and 1 on the CPU, resulting in a densely filled PT page with 31 valid PTEs out of 32.

With this technique, we densely fill PTEs in approximately 97% of the 2MB PT region containing the vulnerable location with the bit-flip, ensuring successful PTE tampering, while consuming 16GB memory. Other 2MB PT regions (e.g., the initial PT region) can remain sparsely filled, requiring 1GB memory per 2MB PT region.

Thus, the attacker can fill the GPU PT region, either **memory-efficiently** (for the initial 2MB PT region) or **densely, with PTEs** (near vulnerable memory).

4.3. Influencing PT Region Placement

To tamper page tables with Rowhammer-based bit-flips, the attacker requires its data pages to be co-located with PT regions in neighboring rows. To enable this, we first understand the allocation heuristics of PT regions.

Allocation Heuristics. By observing the memory dumps of GPU processes, we empirically observe the PT regions are allocated between 64-128MB away from data pages. However, once we deplete the initial 2MB PT region, using the memory-efficient PT allocation techniques from Section 4.2, we observe subsequent PT regions are allocated from the same memory pool as data pages, enabling co-location of data pages and PT pages in neighboring DRAM rows.

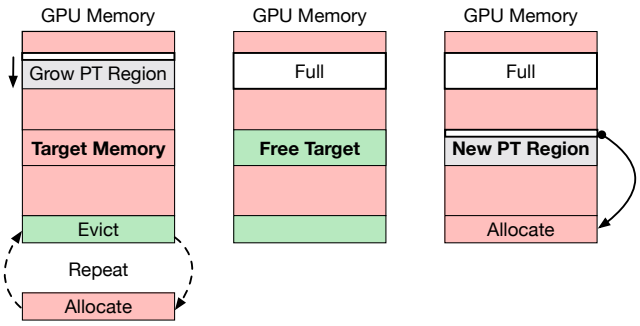


Figure 6: Workflow for Page Table Massaging.

Massaging PTEs to Vulnerable Locations. Since regions are allocated from the same pool as user data pages, we use this to steer a PT region into a chosen physical 2 MB frame vulnerable to bit-flips. Our goal is to reach an allocator state that is one allocation away from creating a PT region, and only two page frames are free: one, the 2MB frame vulnerable to bit-flips, and another page frame for the data memory allocation, as shown in Figure 6 (center). This accommodates both the new PT region to be allocated at the vulnerable location and the data memory allocation that triggers the PT creation.

Figure 6 illustrates our massaging technique. First, we exhaust device memory via UVM allocations. We then repeat an evict-reallocate cycle, iteratively evicting a selected attacker data page from the GPU to the CPU and reallocating a new data page in its place, advancing the PT allocator to the point when it will create a new 2MB PT region. Before the next allocation, we evict the 2MB target data page (the page frame known to be vulnerable), by accessing it from the CPU, and evict another data page (2MB, 64KB or 4KB) elsewhere. The subsequent memory allocation forces the allocator to place the new 2MB PT region in the target slot, following the eviction order.

By **creating holes in the memory**, one at the vulnerable page frame and another elsewhere for a data frame, and **triggering a memory allocation creating a PT region**, the PT gets placed at the vulnerable location in memory.

This PT region can now be filled densely with PTEs, as described in Section 4.2 for PTE tampering.

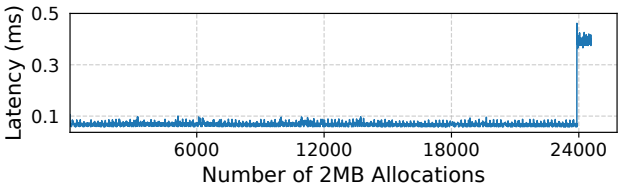


Figure 7: Spike in access latency when UVM memory allocation exceeds GPU DRAM capacity (48GB), due to page evictions from GPU to CPU.

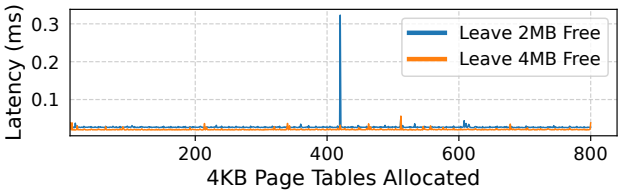


Figure 8: Spike in access latency when memory allocation causes 2MB PT region creation, if we leave only one page-frame free (e.g., 2MB free). No spike if free memory can fit both data and PT region allocations (e.g., 4MB free).

4.4. Page Table Page Allocation Side-Channel

For the attack, a critical requirement is that an unprivileged process knows exactly *when* a new 2MB PT region is allocated. While a new PT region will be allocated every 1GB of memory allocation (as shown in Section 4.2), the initial state of the PT region can be hard to predict, and vary based on the GPU process context and code size, making it hard to exactly know when new PT region allocations occur without instrumenting the driver. For this, we leverage a new timing side-channel emanating from UVM memory eviction.

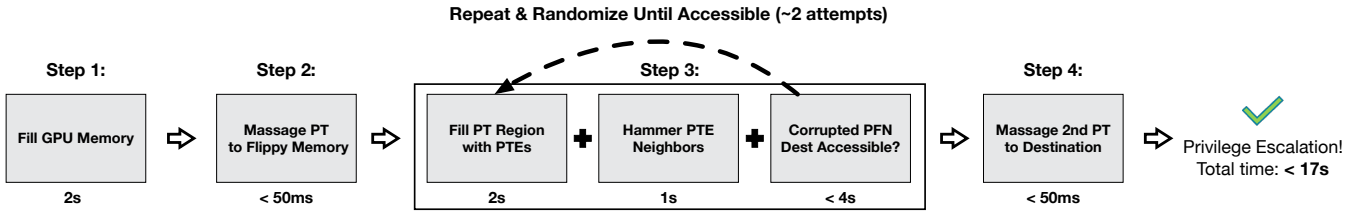


Figure 9: Workflow for the end-to-end attack on page tables to achieve GPU privilege escalation. Step 3 has a success rate of 44% ($n = 50$ attempts). The total exploit time takes 17s on average, including execution of Step 3 on average 2.3 times.

When the GPU memory is full and a new UVM memory page is to be allocated, a GPU page is evicted to the CPU. This eviction causes a high access latency for the first access, as shown in Figure 7, allowing us to precisely identify *when* a UVM page eviction occurs from the GPU. We leverage this page-eviction timing side-channel to infer when new 2MB PT regions are allocated. By keeping just one page-frame free on the GPU DRAM before a memory allocation, which also requires a PT region allocation, it results in an eviction-caused spike in access latency, as shown in Figure 8. This allows the attacker to identify when the *first* 2MB PT region allocation occurs. Subsequent PT region allocations occur periodically after each 1GB memory allocation.

UVM page evictions introduce measurable timing side-channels. By **triggering page evictions on PT-region allocations**, an attacker can observe the timing variations to learn *when* PT regions are created.

5. Attacking GPU Page Tables

In advance, the attacker profiles the memory for vulnerable locations susceptible to bit-flips by executing Rowhammer campaigns similar to prior work [11]. Subsequently, using page table manipulation techniques from Section 4, we launch an end-to-end attack on GPU page tables using Rowhammer, to achieve privilege escalation. Figure 9 shows the steps (1-4) for the end-to-end attack, as described next.

Step 1: Fill GPU Memory. The first step of the attack is to fill the entire GPU memory using UVM allocations, since the page-table massaging relies on a page-eviction side channel that requires all GPU memory to be occupied. Unlike `cudaMalloc`, that fails when out of memory, UVM swaps memory to the CPU once the GPU DRAM is exhausted. As the initial occupancy of the GPU DRAM is unknown, we detect when the memory is full, by measuring access latency after each 2 MB UVM allocation and checking for a latency spike that indicates a page eviction to CPU (Figure 7). Once the attacker observes consecutive spikes, they know the GPU DRAM is fully allocated.

Step 2: Massage PT to Vulnerable Memory. Next, to enable PTE tampering, we force PTEs to be allocated in the physical page frames vulnerable to Rowhammer bit-flips. This consists of two stages: (1) Filling the first PT region and (2) Massaging the next PT region allocation to vulnerable frames. First, we fill the initial 2MB PT region with our memory-efficient PTE filling techniques, using 2MB + 4KB pages, as discussed in Section 4.2. To monitor when the PT region is full, we iteratively free up a 2MB region of the GPU DRAM (by touching it from the CPU and swapping to the CPU), and then allocate 512 × 4KB page frames on the GPU (allocating 2MB + 4KB UVM each time and accessing only the 4KB on the GPU side) as shown in Figure 4, and monitoring the access latency. On a timing spike, as shown in Figure 8, we know a new PT region has been allocated by evicting an existing page.[3] In this PT region, each 2MB + 4KB allocation allocates 32B in a PD0 entry and a 4KB PT page. Thus, the number of 2MB + 4KB allocations (A) required to fill up the current 2MB PT region and create the next PT region is given by,

$$2\,\text{MB} = 4\,\text{KB} \cdot (A + \lceil A/128 \rceil) \tag{1}$$

Based on Equation (1), we require 508 allocations in total for the next PT region to be allocated. We verify this empirically as shown in Figure 10, where we measure the access latency as the number of 4KB pages allocated on the GPU grows. The first spike corresponding to the first PT region allocation is after 420 allocations, and subsequent spikes appear every 508 allocations of 4KB frames. Before the 508$^{\text{th}}$ allocation, we free up an additional 2MB frame by evicting the vulnerable page frame to CPU, and thus the next allocation is forced to map a new PT region to the target vulnerable 2MB frame.

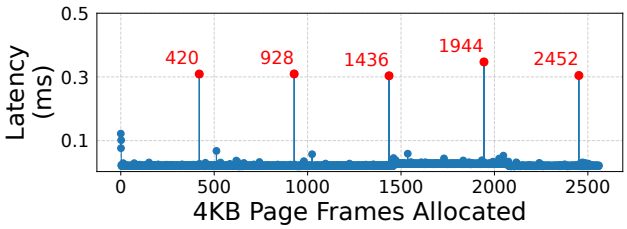


Figure 10: Access times after memory allocations of 4KB pages on the GPU. Spikes (red dots) occur when PT regions are allocated, every 508 data allocations, as per Equation (1).

3. We observe the UVM evictions are in LRU order; so we preserve the vulnerable page frame in the GPU memory by re-accessing it periodically.

Step 3: Tamper PTEs and Check Corrupted PFN. Next, we fill the 2MB PT region allocated to the vulnerable memory, densely with PTEs. As discussed in Section 4.2, using 64KB data frames, we densely fill the PT region (31 out of 32 PTEs valid in each 256B PT page), ensuring the best chance of a bit-flip hitting a valid PTE. Using techniques from GPUHammer [11], we hammer attacker-controlled data rows, neighboring the region where the PTEs are mapped, to flip the vulnerable bits and tamper PTEs.

Check Corrupted PFN Destination. After hammering, we check if this corrupted a PTE PFN, by sequentially reading through the attacker controlled data. Initially, each data page has a sequential identifier written to it. On streaming through the data, if some data is out of order, we know the PFN of the VA was corrupted, and the data value tells us the correct VA for the accessed frame. If this occurs, we know the tampered PFN destination is attacker controlled (x). Otherwise, if the data of the corrupted PFN destination is not recognizable, it may be an unassigned page, or a page of another user. In this case, we repeat Step-3; we randomly change the PFN values by evicting the corresponding data pages and re-allocating them in a different order, and repeating till the corrupted PFN destination is attacker controlled (x). In practice, Step-3 achieves this in 2-3 attempts.

Step 4: Allocate PTEs at Corrupted PFN Destination. Lastly, we evict the attacker-controlled page, x, that is at the corrupted PFN destination, and ensure another PT region is allocated into this frame. For this, we repeat the massaging process from Step 2. Once the PTEs are allocated to this physical frame, they can be accessed and modified using the VA of our previously corrupted PTE PFN. This completes our attack, as now we can arbitrarily control the PFN value for a given VA,[4] through which we can access the entire GPU memory, thus achieving GPU-side privilege escalation.

> Using the VA translated by the corrupted PTE, we access the second PTE and PFN2. By controlling PFN2, and using VA2 translated by PFN2, we can access the entire GPU memory, achieving GPU privilege escalation.

6. Exploitation Results

To assess the potential for exploitation with GPUBreach attacks, our evaluations answer the following questions:

1) Can we leverage Rowhammer bit-flips to tamper PTEs and escalate privilege on a GPU? (Section 6.1)
2) With our arbitrary GPU read primitives, can we leak sensitive data from GPU? (Section 6.2, Appendix D)
3) With our arbitrary GPU write primitives, can we tamper with GPU code, to enable stealthy accuracy degradation on ML models? (Section 6.3)
4) Can we use this to cross the device-host boundary and escalate privileges on the CPU? (Section 6.4)

4. Each time we change the PTE PFN in DRAM, we need to evict its TLB entry, to ensure a page-table walk. We evict TLB entries by streaming through 2 GB of memory with 64KB pages which thrashes the TLB.

We answer these questions by studying GPUBreach exploits on an NVIDIA RTX A6000 with 48 GB GDDR6 DRAM, known to be vulnerable to Rowhammer [11].

TABLE 2: System Configuration

Component	Specification
OS	Ubuntu 22.04.5 LTS
CPU	AMD Ryzen 5945WX
Compiler	g++ 10.5.0
GPU	NVIDIA RTX A6000
GPU Memory	Samsung, 48 GB GDDR6
NVIDIA Driver	580.95.05
CUDA Toolkit	12.8

6.1. GPU Privilege Escalation via PTE Tampering

To achieve successful PTE tampering, the Rowhammer bit-flips in a PFN must change its destination to another 2MB page frame containing a PTE. This requires the PFN to change by at least 2MB, which requires the bit-flip to be in a range of 27 bits (PFN[34:8]) within the 48-bit PFN (PTE[42:16]), as per the PTE format shown in Figure 2.

Profiling Phase. We use the code from GPUHammer [11], to induce Rowhammer bit-flips on an RTX A6000 GPU. We generate the mappings of virtual addresses to banks and rows using row-buffer conflict side-channels [40] (30 min/bank) and use synchronized multi-warp hammering patterns [2], [3], [11] to perform 24-sided hammering campaigns on 6 banks of the GDDR6 DRAM. We use the checkered data patterns, 0x55 and 0xaa, for the aggressor rows and their inverse values for the victim rows. Our hammering campaigns take 3 hours per DRAM bank.

In the six DRAM banks we hammered (labeled $A-F$) on the RTX A6000, we observed 34 bit-flips in total, out of which 9 bit-flips are suitable for the privilege escalation with PTE corruption. Table 3 shows the bit-flips that fit the criteria for privilege escalation with PTE tampering. The entire offline profiling phase takes 6 banks $\times$ 3.5 hours / 9 flips $\approx$ 3 hours per exploitable bit-flip.

TABLE 3: Locations of Bit-Flips within 64-bit PTEs, suitable for GPU Privilege Escalation with PTE tampering

Bit-flip No.	Byte Location	Bit Location	PTE Bit $byte*8+bit$	Jump Distance
A_1	2	4	20	16 MB
B_1	2	6	22	64 MB
C_1	2	4	20	16 MB
C_2	3	2	26	1 GB
D_1	2	4	20	16 MB
E_1	3	4	28	4 GB
F_1	3	0	24	256 MB
F_2	3	1	25	512 MB
F_3	3	7	31	32 GB

Online Phase. Without loss of generality, we choose bit-flip, A_1, for our exploits, that produces the desired page-table corruption and analyze its exploitation end-to-end. We

execute Steps 1-4 in our attack workflow from Section 5, observing that we are required to repeat Step-3 two times on average, before we get a tampered PTE pointing to an attacker-controlled page, where we allocate a PT region. Thus, by controlling its own PT, the attacker achieves arbitrary GPU R/W capabilities and device-side privilege escalation in less than 20 seconds. This attack can run concurrently with other processes, in a temporally sliced manner, or spatially sliced with MPS.

6.2. Leaking Crypto Keys from GPU Memory

Using the arbitrary GPU read primitive from GPUBreach, we show how an attacker process on the same system can recover secrets of another process resident on the GPU. As a case study, we leak private keys from NVIDIA's post-quantum cryptography (PQC) library, cuPQC [41], which accelerates PQC algorithms, and is used by libOQS [42].

Setting. We target a desktop or workstation GPU, where the victim runs short-lived, GPU-accelerated key-exchange routines (MLKEM512) from cuPQC. After each key-exchange routine, the victim copies its result back to CPU memory and the device memory is immediately freed. The attacker has code-execution capabilities on this GPU and has gained arbitrary R/W privilege over GPU memory.

Approach. Dumping the entire device memory (takes more than 10 seconds) is too slow relative to the secret key residency on the GPU (less than a millisecond). Instead, the attacker identifies a small candidate set of pages where the secret may be resident and performs rapid, targeted reads using the approach in Figure 11, as we describe below.
1. Locating candidate pages. We observe that allocator

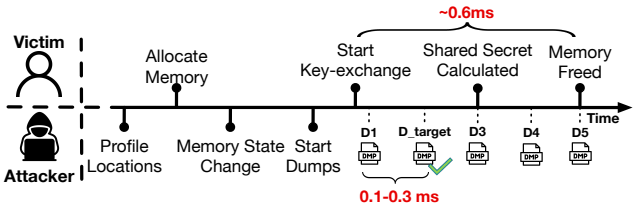


Figure 11: Timeline of attack on cuPQC

placement of variables is largely reproducible across runs of the workload, and freed regions are zeroed by the runtime. Thus, to locate secret-resident pages, the attacker pre-fills the user pool with a non-zero pattern (e.g., `0xFF`). When the runtime later allocates, then zeroes freed regions, the zeroed pages stand out and form a short candidate list to search for the secrets (MLKEM512 keys occupy a few KB).
2. Detect Victim Allocations. To detect when secret data becomes resident, the attacker monitors allocator state (e.g., via `cudaMemGetInfo`) to observe changes in free memory that correlate with victim allocations. Knowing when victim allocations occur, the attacker triggers targeted reads of our small set of candidate pages.
3. Rapid targeted dumping. Targeted reads of a few megabytes are fast: on our RTX A6000, dumping 2MB takes about 0.1–0.3 ms, while an MLKEM512 key typically resides for around 0.6 ms. If the dump of the page containing the secret key overlaps this window, the secret is captured.

Results. We run our exploit on the MLKEM512 example code in cuPQC. To locate candidate victim pages, we prefill and scan the entire GPU user memory (∼47 GB), requiring less than 5 minutes. After the victim ran, we observed ∼100 zeroed candidate pages. We applied our attack workflow, shown in Figure 11, against these pages, across 1000 independent victim key-exchanges. We observe full recovery of the private key in 4.4% of the key-exchange runs (44 out of 1000). This shows that an attacker with GPUBreach can reliably leak short-lived cryptographic secrets from GPUs.

In Appendix D, we similarly show how GPUBreach can leak sensitive model weights from a long-running large language model (LLM) inference server.

6.3. Stealthy ML Degradation via Code Tampering

Using the arbitrary-write primitive from GPUBreach, we demonstrate GPU code tampering. Prior work showed that although bit-flips in ML model weights can degrade accuracy [11], [22]–[24], integrity checks can easily detect such corruption [43]. Meanwhile, on CPUs, corrupting a single branch in BLAS code has been shown to stealthily degrade accuracy universally across all models [44]. In this exploit, we show that stealthy code tampering is also possible on GPUs: tampering a single branch in proprietary cuBLAS on GPUs can produce hard-to-detect accuracy degradation universally across ML models. As GPU-resident code is not observable by users, such GPU code corruption is stealthier than weight corruption or even CPU code corruption.

Setting. We assume the victim runs a long-lived inference process on the GPU. The attacker knows the victim's PyTorch/cuBLAS versions and can identify targets in the code offline. The attacker seeks to induce *silent* degradation universally across models by corrupting a *single* branch in cuBLAS code under the following constraints: (1) the model must not crash, (2) outputs remain syntactically valid (e.g., a valid class), and (3) minimal latency impact (within 10%).
Identifying Critical GPU Code. We find that the GPU kernel code is resident in global memory in dedicated 2MB pages, and often ends with recognizable patterns like `NOP` sleds, shown in Listing 1. We locate the cuBLAS code in the offline phase by taking the difference between code dumps before and after the cuBLAS library runs.

```
EXIT ; //Exit current GPU thread
BRA 0x230; //Trap to prevent further execution.
NOP;NOP;NOP; ..........
```

Listing 1: `NOP` sled at the end of a GPU kernel code

Corruption Strategy. Prior corruption used bit-flips to alter branch instruction directions. In GPUs, branch instructions (BRA) as shown in Listing 2 are all unconditional, with the direction (taken or not) controlled by the predicate register (P0). With our arbitrary R/W primitive, we can prevent a branch being taken by simply *replacing* `BRA` with a `NOP`.

```
ISETP.GE.AND P0, PT, R0, R2, PT; //R0 >= R2
PLOP3.LUT P0, PT, P0, PT, PT, 0x8, 0x0 ; //Mask
@P0 BRA 0x1d0 ; //Branch if P0 is TRUE (R0 < R2)
BRA 0xd0 ; //Branch Otherwise (R0 >= R2)
```

Listing 2: BRA instruction and predication in GPU

Vulnerable Branch Discovery. We analyzed PyTorch with five image-classification models and found that the cuBLAS code contains over 5000 branch instructions. Furthermore, we observe that tampering with randomly selected BRA instructions is ineffective: even after 1000 attempts (24 hours) of tampering a randomly selected branch, we cannot find a vulnerable branch that universally reduces inference accuracy. To find such vulnerable branches efficiently, we develop a 3-stage filtering pipeline, as shown in Figure 12:

- **Page filter.** We overwrite instructions in each 2 MB code page with the EXIT opcode from Listing 1. Because EXIT immediately terminates a kernel, this allows us to quickly filter out code pages without significant impact on model accuracy when removed.
- **Kernel filter.** In pages that survive the page filter, we isolate individual kernels by detecting NOP sleds placed at kernel boundaries. We then iteratively replace half of the remaining kernels with EXIT, at each step, to identify a subset of kernels that are critical to accuracy.
- **Branch Identification.** Within each candidate kernel, we iteratively replace half of the remaining branch instructions with NOPs in each step and observe the inference, converging quickly on the critical branch.

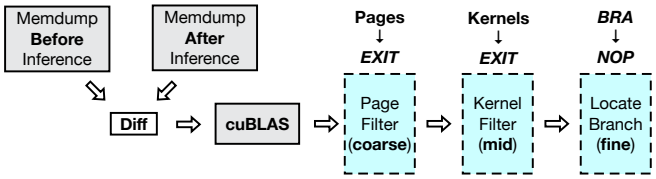


Figure 12: Filtering pipeline to locate vulnerable branches with coarse (page), mid (kernel), and fine (branch) reduction.

Evaluation Results. We ran the offline phase with PyTorch's pretrained AlexNet and benchmarked its accuracy on a subset of the ImageNet dataset [45] like prior works [11], [22]. From the cuBLAS binary, we observed a total of 5436 candidate branches. Applying the filtering strategy in Figure 12 removed 75% of those branches at the page filter, and a further 98% at the kernel filter, leaving only 2 surviving kernels with 25 branches. From this reduced set, we try tampering with each branch, and locate a single branch that corrupts the models the most, with a minimal impact to inference latency. Our offline search process completes in under 100 benchmark runs across all stages, taking less than 2 hours. As shown in Table 4, across all ImageNet models, tampering a single branch instruction universally degrades the accuracy of all models from 57-80% down to 0.1%. As this attack causes <6.6% change in runtime, and the GPU code tampering is undetectable to users without access to device-side code, this attack is much stealthier than prior attacks tampering ML model code on CPUs [44], or model parameters on CPUs or GPUs [11], [22]–[24] that are both easily detectable.

TABLE 4: All evaluated models show substantial accuracy degradation, with latency delta within ±10% of the original.

Network	Base Acc.	Degraded Acc.	ΔLatency
AlexNet	56.68%	0.12%	+1.26%
VGG16	72.20%	0.10%	-4.32%
ResNet50	80.28%	0.10%	-6.62%
DenseNet161	77.20%	0.10%	-2.92%
InceptionV3	69.90%	0.10%	-6.64%

6.4. CPU-Side Privilege Escalation

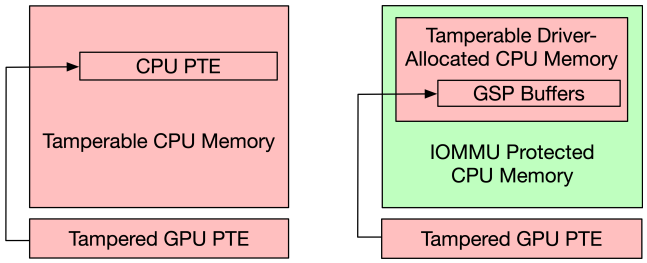
Thus far, GPUBreach has shown elevated privileges on the GPU-side with arbitrary read/write capabilities on GPU memory. Prior work has shown that compromised PCIe devices [46] can alter the host state via DMA and obtain escalated privileges on the host. In this section, we demonstrate how a compromised discrete GPU, such as in GPUBreach, can also tamper with driver memory on the CPU to escalate to root level privileges on the system.

Setting. We assume a workstation GPU with a single user, or a GPU in a containerized setting in the cloud, shared between containers of different users [19], [20]. We assume the attacker has arbitrary R/W privilege on the GPU memory via GPUBreach, which can write to GPU PTEs. As shown in Figure 2, GPU PTEs have a 2-bit Aperture field, which indicates the memory domain the PTE refers to. When the Aperture is set to the system memory mode (0b11), the PTE refers to CPU memory, and the PFN is interpreted as a CPU address. Thus, by overwriting GPU PTEs (aperture and PFN), the attacker can point to and access CPU memory.

We also assume the system enables the IOMMU, as recommended by all major vendors including NVIDIA and AMD [25]–[27]. IOMMU is a *standard defense* against DMA-based attacks, which prevents a malicious device from accessing arbitrary host memory. Thus a compromised GPU is restricted to only the IOMMU-permitted driver memory on the host (the IOVA region mapped by the IOMMU), as shown in Figure 13. With IOMMU enabled, when a PTE has the Aperture bits set to system memory mode, the PFN corresponds to the IOVA. Finally, we assume that the attacker can recover kernel address information, such as KASLR offsets through other side-channels [47]–[49].

Attack Overview. The goal of the attack is to corrupt the kernel-mode GPU driver, running on the CPU, within IOMMU-permitted regions, to gain root-privileges. The attack achieves this by corrupting trusted data-structures within IOMMU-permitted regions in the NVIDIA kernel-mode driver, that triggers memory-safety bugs in the driver, leading to arbitrary write primitives within the linux kernel.

The driver allocates shared buffers in CPU memory for GPU communication via DMA, including RPC queues, fault



(a) IOMMU off (b) IOMMU on

Figure 13: CPU memory accessible by a compromised GPU. Without IOMMU, the entire CPU memory is accessible from GPU. With IOMMU enabled, the GPU can only DMA into driver memory on CPU assigned to the device.

buffers, and semaphores. These buffers are written exclusively by privileged GPU components (GSP and GMMU) and read by the kernel driver. We find that these GPU-produced values are systematically trusted without validation, flowing into security-sensitive operations as unchecked memcpy sizes, loop bounds, and array indices. A compromised GPU can thus overwrite messages written by the GSP/GMMU in these buffers, before the driver consumes them, creating memory safety issues within the kernel, eventually leading to system-wide privilege escalation.

Through manual inspection, we find *dozens of* sites in the NVIDIA driver code where GPU-managed data structures are consumed without sanitization. These buffers can be overwritten by a compromised GPU as the IOVAs for these structures are stable even on system restarts. Below, we describe the most interesting kernel OOB write, achieved by tampering the GSP message queue, which results in root-level privilege escalation. We cover the other vulnerabilities in the latter part of this section and in Appendix C.

Overwriting GSP Message Queue. We tamper the GSP message queue structure in the GPU driver, which is used for device-to-host notifications, located at the offset `0x42000` from the base address of the IOVA region and searchable via the pattern of the queue's content. The queue consists of 63 entries, each 4 KB in size; a message can span multiple consecutive entries directed by an `elemCount` member in the first entry. Once the GSP writes a message into the message queue, it will increment the `rxAvail` counter by one. The driver checks if there are enough messages according to the value of `rxAvail` and `elemCount`. If there are enough messages, the driver copies messages from the GSP queue into a kernel staging buffer (maximum 16 elements). We observe an important metadata structure, `pMetaData`, is placed immediately after this staging buffer. It contains critical fields, like callback function pointers and other pointers used to access and modify data. GSP's input `elemCount`, is trusted and not sanitized before use. Hence, a malicious GPU can tamper this variable to any value larger than 16 causing OOB writes out of the staging buffer, which will overwrite members of `pMetaData`.

By supplying a crafted `elemCount` in the N-th entry of the GSP message queue, an attacker can force the driver to copy beyond the staging buffer and overwrite `pMetaData` with attacker-controlled data. This requires passing the driver's validation checks on the shared counter `rxAvail`. To pass the driver's validation checks, the attacker needs to inflate `rxAvail` to at least 17 to indicate sufficient entries are available. This can be achieved by flooding the queue with repeated `cudaDeviceGetAttribute` calls, causing the GSP to enqueue messages faster than the driver consumes them. We describe how we select the value of the queue index N in Appendix A.

Because the attacker controls all queue entries, they fully control the overwritten `pMetaData` (example payload in Figure 14). With this technique, the attacker has several gadgets in the GPU kernel driver to perform privileged operations, such as **arbitrary writes** and kernel-level execution via **arbitrary function calls** (Appendix B). We discuss one of the strongest primitives we found, a controlled *4-byte write to an arbitrary kernel address*, and use it to escalate to root privilege on the host.

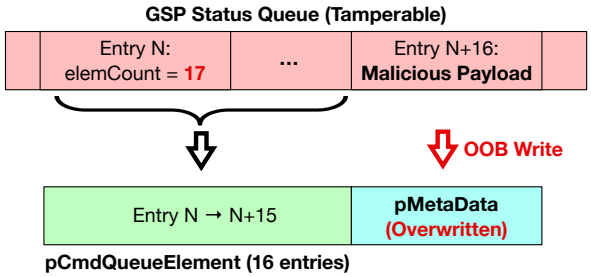


Figure 14: CPU-side privilege escalation. The compromised GPU injects `elemCount=17` and a malicious payload in the driver memory accessible via IOVA, that causes an OOB write in GPU inaccessible driver memory, that overwrites `pMetaData`. `pMetaData` contains data pointers used later, that when overwritten, result in arbitrary 4-byte writes in the kernel.

Arbitrary 4-byte Kernel Write & Privilege Escalation. After overwriting the `pMetaData` structure by triggering an element-wise copy from the GSP queue into the staging buffer, the driver calls `msgqRxMarkConsumed`. In this function, the driver writes the 4-byte value of `rxReadPtr` into the address stored in `pReadOutgoing`. By controlling the values of both members, the attacker can perform arbitrary 4-byte writes in kernel memory. Using this primitive, the attacker can modify the target process's credential structure (struct `cred`) in the kernel and set the euid to 0, to get root-level privileges. Appendix A provides more details about the exploit, like the value of the malicious payload, and the steps to ensure kernel stability after the exploit. Note that since the GPU message queue is corrupted in this exploit, a GPU reset is required to restore functionality. Crucially, the kernel remains in a stable state, enabling the attacker to execute arbitrary code with elevated privileges.

By exploiting DMA from the compromised GPU to the privileged GPU driver running on the CPU, we demonstrate the **first system-wide privilege escalation attacks using Rowhammer on discrete GPUs**, even with protections like IOMMU enabled.

Other OOBs by Tampering Fault Buffer. The fault buffer in the NVIDIA driver is a potent target for tampering. It contains over a dozen GPU-produced fields read by the driver using the `READ_HWVALUE_MW` macro, without runtime bound validation. We find that `UVM_ASSERT` macros, used with some of these reads, are debug-only checks that are removed in release builds. The fault buffer fields such as `fault_address`, `gpc_id`, `client_id`, and `fault_type`, are later used in address calculations, indexing operations, and control-flow decisions. If these values are maliciously overwritten, these can lead to OOB memory accesses. We describe some of these OOBs in Appendix C.

7. Discussion and Limitations

Applicability to Other GPUs. Our exploits are demonstrated on the NVIDIA RTX A6000 GPU, a widely used workstation and cloud GPU previously shown to be susceptible to bit flips [11]. This is because the underlying Rowhammer vulnerability is GPU-specific, with attack patterns varying based on the memory vendor, in-DRAM defenses, and memory technology. Our PTE massaging based exploitation technique, however, applies to any NVIDIA GPU vulnerable to Rowhammer. NVIDIA's security notice [28] indicates that GPUs across multiple generations (Ampere, Ada, Hopper, and Blackwell) may be affected by Rowhammer. We have verified that our CPU-side privilege escalation exploit, based on the driver vulnerabilities, remains effective across a wide range of driver versions (545.23.08–580.95.05), and is independent of the GPU model. We further validated it on two additional GPUs, a laptop RTX 3080-Laptop (Ampere) and a server L4 (Ada), using simulated bit flips in the absence of observable Rowhammer-induced bit flips on these GPUs. Future studies that may discover Rowhammer bit-flips on other GPUs could also apply our exploits to those GPUs.

Impact of ECC. Our exploits assume ECC is disabled, consistent with real-world usage of many GDDR-based GPUs. We observe that cloud providers continue to offer A6000 instances with ECC disabled by default, as enabling ECC incurs a 6.25% memory overhead and up to 10% slowdown on GDDR-based GPUs [11], [37]. Enabling system-level ECC or using GPUs with on-die ECC in DRAM chips (e.g., GDDR7 or HBM3) may reduce the threat, but not fully prevent our exploits. Recent Rowhammer attacks [8], [10] on CPU memories show that DRAM ECC mechanisms, which detect up to two-bit errors, fail to prevent Rowhammer attacks inducing three-bit errors or more, leaving systems exposed even with ECC enabled. Future work can evaluate such attacks under ECC-enabled configurations on GPUs.

UVM Requirement. Our exploits require 64KB and 4KB page frames for massaging PTEs, that are obtainable only through Unified Virtual Memory (UVM). UVM is supported on all NVIDIA GPUs since Pascal and is available in most multi-tenant environments, including Multi-Instance GPUs (MIG), Multi-Process Service (MPS), containerized deployments using Kubernetes [19], [20], and GPU confidential computing. Future works can explore these environments. Our exploits do not apply when UVM is currently disabled, such as in vGPU configurations supporting live migration, or when multiple vGPUs are time-sliced on a single GPU [50]. Table 5 lists the settings where our exploit is applicable.

TABLE 5: Applicability matrix of the attack.

	ECC	UVM	IOMMU	MPS	MIG	CC
Off	Yes	Yes	Yes	Yes	Yes	Yes
On	No	No	Yes	Yes	Unknown	Unknown

8. Mitigations

Prevent PT Massaging. Since our exploits rely on precisely massaging PTEs to desired locations, restricting fine-grained control over PT placement can prevent these attacks. For instance, the GPU driver can statically isolate page tables from data pages to avoid their co-location in memory, which is required by our Rowhammer attacks [51], [52]. While attacks like PTHammer [53] are theoretically possible, they are harder on GPUs due to higher memory latencies. Similarly, randomizing PT region locations can further hinder targeted placement. Additionally, limiting the use of small pages (4KB or 64KB) increases PT massaging memory requirements by 16-256×, making such attacks infeasible on most GPUs with under 100GB of RAM, albeit at the cost of up to 256× higher latency for access to UVM pages across the CPU-GPU boundary. Disabling UVM similarly hinders PT massaging, but limits application functionality [54], [55].

Input Sanitization in the GPU Driver. To prevent a compromised GPU from escalating privileges on the CPU, the GPU driver could treat all GPU-originated data consumed by it as untrusted. By routing inputs from the GPU through a staging buffer and enforcing bounds and sanity checks before consumption, the driver can prevent memory-safety vulnerabilities in the kernel due to a compromised GPU and block CPU-side privilege escalation in our exploits.

Adopt Principled Rowhammer Defenses. While enabling ECC can make attacks harder, adversaries can still craft successful exploits by injecting multi-bit-flips that surpass the ECC's detection capabilities [6], [8], [10]. Instead, integrity-based defenses that use Message Authentication Codes to detect PTE tampering [56]–[58] can fully stop our exploit. GPUs could also adopt principled in-memory Rowhammer mitigations, such as secure in-DRAM trackers [59]–[62], or Per Row Activation Counting (PRAC) mechanisms [63]–[68], to fully eliminate the underlying vulnerability.

9. Related Work

GPU Exploits. Prior GPU exploits have leveraged residual GPU memory to leak rendered webpages and ML model outputs [69]–[71]. GPU side-channel attacks like GPU.zip [72], Pixnapping [73], and NVBleed [74] leaked rendered pixels, while Spy in the GPU Box [75] and LeakyDNN [76] leaked ML model parameters. Other works [21], [77] demonstrated covert-channels through GPU micro-architectural optimizations. GPU code is also vulnerable to buffer overflows that enable control-flow hijacking and ML model tampering [78]–[80]. Recent work [81] showed that RPC queue metadata, used for CPU-GPU communication, can leak information in confidential computing settings. In contrast, GPUBreach introduces the first privilege escalation attacks on GPUs via Rowhammer, enabling arbitrary read/write of GPU memory and even elevated privileges on the CPU by tampering GPU driver data-structures, surpassing the capabilities of all prior GPU exploits.

CPU Privilege Escalation via Malicious Device. Prior works [82], [83] have shown that vulnerabilities in integrated mobile GPU drivers (ARM Mali, Qualcomm Adreno) can allow unprivileged applications to corrupt kernel memory and achieve privilege escalation on the CPU. Thunderclap [84] showed that malicious peripherals can exploit OS vulnerabilities via DMA accesses to achieve privilege escalation, even with IOMMU enabled, using a custom FPGA-based device attached to the victim system. Thunderspy [46] exploited the lack of firmware verification in Thunderbolt controllers: with physical access, attackers could reflash SPI firmware to bypass security checks, and achieve DMA-based privilege escalation. DMARacer [85] identified DMA-based race conditions in driver code, including TOCTOU bugs that can be exploited by malicious peripherals. In contrast to prior attacks that require driver bugs, custom hardware, or physical access, we show that a *discrete* GPU, compromised via Rowhammer from an unprivileged process, can tamper with trusted GPU-writable buffers in the privileged GPU driver, achieving CPU-side privilege escalation.

Other Rowhammer Exploits. Prior Rowhammer exploits targeted CPU-based memories. In DDR3, Flip Feng Shui (FFS) [14] used Rowhammer to compromise cryptographic code, while Drammer [34] achieved privilege escalation on mobiles. ThrowHammer [86] and Rowhammer.js [17] demonstrated these over the network and JavaScript respectively. TRRespass [2], Blacksmith [4] and SMASH [3] bypassed in-DRAM TRR defenses in DDR4, while ECCploit [6], ECC.fail [8], and Phoenix [10] defeated ECC in DDR3, DDR4, and DDR5. Zenhammer [7] showed AMD CPUs with DDR5 memories are vulnerable. Grand Pwning Unit [9] used integrated GPUs on mobile SoCs, sharing the CPU DRAM, to perform Rowhammer attacks on LPDDR3 DRAM. In contrast, we are the first to demonstrate system-wide privilege escalation using Rowhammer on *discrete* GPUs with GDDR6 memory. Prior works also use Rowhammer to flip bits in ML models on CPU DRAM [22]–[24], [44], [87]–[89] or GPU DRAM [11], degrading model accuracy. Our exploits go beyond this by enabling *arbitrary* leakage or tampering of ML model code and data on GPUs.

Rowhammer Mitigations. Most prior mitigations target CPUs. Software-based approaches such as GuardION [90], CATT [91], RIP-RH [92], ZebRAM [93], Siloz [52], and Citadel [51] isolate memory across security domains, while Copy-On-Flip [94] leverages ECC to detect and relocate attacked pages. These strategies could mitigate GPU-based attacks on page tables with NVIDIA driver support. Hardware-based defenses, including Rowhammer trackers [59], [60], [62], [95], [96], aggressor relocation [97]–[100], delayed activations [101], refresh-generating activations [102], and page table integrity mechanisms [57], that were developed for CPUs, could also be adapted for GPUs; however this can introduce varying performance costs depending on GPU latency and bandwidth constraints.

10. Conclusion

We present GPUBreach, the first Rowhammer-based privilege-escalation on NVIDIA GPUs with GDDR6 memories. By developing efficient GPU page-table massaging techniques, we tamper with GPU page tables using Rowhammer, achieving arbitrary GPU R/W capabilities. We demonstrate practical attacks including data leakage, stealthy code tampering, and privilege escalation on the CPU even in the presence of protections like IOMMU. Our work underscores the need for effective Rowhammer mitigations in GPUs, and a re-evaluation of the trust placed in the GPU within kernel-mode GPU drivers.

Acknowledgments

This research was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants under funding reference number RGPIN-2023-04796 and RGPIN-2018-05931, and an NSERC-CSE Research Communities Grant under funding reference number ALLRP-588144-23. David Lie is also supported by Tier 1 Canada Research Chair CRC-2019-00242. Any research, opinions, or positions expressed in this work are solely those of the authors and do not represent the official views of NSERC, the Communications Security Establishment Canada, or the Government of Canada.

Ethics Considerations

We disclosed our attack to NVIDIA's PSIRT on November 11, 2025, and subsequently also to Google, Microsoft, and AWS, and assisted with reproducing the issue and exploring mitigations. No users or production systems were affected by our experiments that were conducted on local machines within isolated environments, using proof-of-concept cryptographic workloads and open-source models. Upon completion of the coordinated disclosure process, we have publicly released all the code artifacts associated with this work to foster open science. Our artifacts are available at: https://github.com/sith-lab/gpubreach.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

[1] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014, p. 361–372.

[2] P. Frigo, E. Vannacci, H. Hassan, V. v. der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the many sides of target row refresh," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 747–762.

[3] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized many-sided rowhammer attacks from JavaScript," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1001–1018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/ridder

[4] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable rowhammering in the frequency domain," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 716–734.

[5] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-Double: Hammering from the next row over," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3807–3824. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kogler-half-double

[6] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 55–71.

[7] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer attacks on AMD Zen-based platforms," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1615–1633. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/jattke

[8] N. Kamadan, W. Wang, S. van Schaik, C. Garman, D. Genkin, and Y. Yarom, "ECC. fail: Mounting Rowhammer attacks on DDR4 servers with ECC memory," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 5679–5698.

[9] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 195–210.

[10] D. Meyer, P. Jattke, M. Marazzi, S. Qazi, D. Moghimi, and K. Razavi, "Phoenix: Rowhammer attacks on DDR5 with self-correcting synchronization," in *S&P (Oakland)*, 2026.

[11] C. S. Lin, J. Qu, and G. Saileshwar, "GPUHammer: Rowhammer attacks on GPU memories are practical," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 5719–5738.

[12] C. A. Haidar, Q. Payet, and M. Tibouchi, "Crowhammer: full key recovery attack on falcon with a single rowhammer bit flip," in *Annual International Cryptology Conference*. Springer, 2025, pp. 103–135.

[13] S. Amer, Y. Wang, H. Kippen, T. Dang, D. Genkin, A. Kwong, A. Nelson, and A. Yerukhimovich, "Pq-hammer: End-to-end key recovery attacks on post-quantum cryptography using rowhammer," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 3567–3582.

[14] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a needle in the software stack," in *25th USENIX Security Symposium (USENIX Security)*, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

[15] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.

[16] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," Google Project Zero, Mar. 2015, accessed: 2025-11-05. [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[17] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 2016, pp. 300–321.

[18] Z. Coalson, J. Woo, C. S. Lin, J. Qu, Y. Sun, S. Chen, L. Yang, G. Saileshwar, P. Nair, B. Fang, and S. Hong, "PrisonBreak: Jailbreaking Large Language Models with at Most Twenty-Five Targeted Bit-flips," *arXiv preprint arXiv:2412.07192*, 2025.

[19] Google, "Share GPUs across workloads with GPU time-sharing," https://cloud.google.com/kubernetes-engine/docs/how-to/timesharing-gpus, Accessed: 2025-01-22.

[20] Alibaba Cloud, "Container Service for Kubernetes: GPU Sharing Overview," https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/cgpu-overview/, 2025, Accessed: 2025-11-05.

[21] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, 2023. [Online]. Available: https://doi.org/10.1145/3576915.3616672

[22] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 497–514. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/hong

[23] F. Yao, A. S. Rakin, and D. Fan, "DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips," in *USENIX Security*, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yao

[24] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1211–1220.

[25] NVIDIA, "Understanding the iommu linux grub file configuration," https://enterprise-support.nvidia.com/s/article/understanding-the-iommu-linux-grub-file-configuration, 2026, accessed: 2026-04-17.

[26] AMD, "Input-output memory management unit (iommu)," https://rocm.docs.amd.com/en/docs-6.3.1/conceptual/iommu.html, 2024, accessed: 2026-04-17.

[27] Microsoft, "Iommu-based gpu isolation," https://learn.microsoft.com/en-us/windows-hardware/drivers/display/iommu-based-gpu-isolation, 2024, accessed: 2026-04-17.

[28] NVIDIA, "Security notice: Rowhammer – july 2025," https://nvidia.custhelp.com/app/answers/detail/a_id/5671, 2025.

[29] A. Nayak, B. Pratheek, V. Ganapathy, and A. Basu, "(Mis)Managed: A novel TLB-based covert channel on GPUs," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 872–885.

[30] NVIDIA, "Pascal MMU Format," https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[31] J. S. Kim, M. Patel, A. G. Yağlıkçı, H. Hassan, R. Azizi, L. Orosa, and O. Mutlu, "Revisiting rowhammer: An experimental analysis of modern DRAM devices and mitigation techniques," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 638–651.

[32] H. Luo, A. Olgun, A. G. Yağlıkçı, Y. C. Tuğrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "Rowpress: Amplifying read disturbance in modern dram chips," in *50th International Symposium on Computer Architecture (ISCA)*, 2023.

[33] I. E. Yuksel, A. Olgun, N. Bostanci, H. Luo, A. G. Yaglikci, and O. Mutlu, "Columndisturb: Understanding column-based read disturbance in real DRAM chips and implications for future systems," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025, pp. 975–994.

[34] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016. [Online]. Available: https://doi.org/10.1145/2976749.2978406

[35] H. Hassan, Y. C. Tugrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, "Uncovering In-DRAM RowHammer Protection Mechanisms:A New Methodology, Custom RowHammer Patterns, and Implications," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1198–1213. [Online]. Available: https://doi.org/10.1145/3466752.3480110

[36] C. S. Lin, J. Qu, and G. Saileshwar, "GPUHammer code repository - sith lab," https://github.com/sith-lab/gpuhammer, 2025, accessed: 2025-11-05.

[37] M. B. Sullivan, M. T. I. Ziad, A. Jaleel, and S. W. Keckler, "Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc," in *50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[38] NVIDIA, "Dgx b200 specifications," https://www.nvidia.com/en-us/data-center/dgx-b200/#specs, 2025, accessed: 2025-11-05.

[39] Linux, "proc_meminfo(5) — Linux manual page," https://www.man7.org/linux/man-pages/man5/proc_meminfo.5.html.

[40] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 565–581. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[41] NVIDIA Corporation, *NVIDIA cuPQC: SDK for GPU-Accelerated Post-Quantum Cryptography*, 2025, accessed: 2025-11-11. [Online]. Available: https://developer.nvidia.com/cupqc

[42] The Linux Foundation, "Post-quantum cryptography alliance brings accelerated computing to post quantum cryptography with NVIDIA cuPQC," https://www.linuxfoundation.org/press/post-quantum-cryptography-alliance-brings-accelerated-computing-to-post-quantum-cryptography-with-nvidia-cupqc, Jan. 2025, accessed: 2025-11-11.

[43] Q. Liu, J. Yin, W. Wen, C. Yang, and S. Sha, "NeuroPots: Realtime proactive defense against Bit-Flip attacks in neural networks," in *32nd USENIX Security Symposium (USENIX Security)*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/liu-qi

[44] S. Li, X. Wang, M. Xue, H. Zhu, Z. Zhang, Y. Gao, W. Wu, and X. S. Shen, "Yes, One-Bit-Flip matters! Universal DNN model inference depletion with runtime code fault injection," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 1315–1330. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/li-shaofeng

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vision*, vol. 115, no. 3, p. 211–252, Dec. 2015. [Online]. Available: https://doi.org/10.1007/s11263-015-0816-y

[46] B. Ruytenberg, "Breaking Thunderbolt Protocol Security: Vulnerability Report," 2020, public version. [Online]. Available: https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf

[47] W. Liu, J. Ravichandran, and M. Yan, "Entrybleed: A universal kaslr bypass against kpti on linux," in *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2023, pp. 10–18.

[48] D. Davoli, M. Avanzini, and T. Rezk, "On kernel's safety in the Spectre era (and KASLR is formally dead)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1091–1105.

[49] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 368–379.

[50] NVIDIA, "Comprehensive knowledge base on vgpu features across hypervisors," 2025, accessed: 2025-11-11. [Online]. Available: https://docs.nvidia.com/vgpu/knowledge-base/latest/vgpu-features.html

[51] A. Saxena, W. Wang, and A. Daglis, "Citadel: Rethinking memory allocation to safeguard against inter-domain rowhammer exploits," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025, pp. 1117–1131.

[52] K. Loughlin, J. Rosenblum, S. Saroiu, A. Wolman, D. Skarlatos, and B. Kasikci, "Siloz: Leveraging DRAM isolation domains to prevent inter-vm rowhammer," in *29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[53] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "Pthammer: Cross-user-kernel-boundary rowhammer through implicit accesses," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 28–41.

[54] P. S. Gali and B. Zaitlen, "Unified virtual memory supercharges pandas with rapids cudf," NVIDIA Technical Blog, Dec. 2024, https://developer.nvidia.com/blog/unified-virtual-memory-supercharges-pandas-with-rapids-cudf/.

[55] L. Engel, "Simplify system memory management with the latest nvidia gh200 nvl2 enterprise ra," NVIDIA Technical Blog, Feb. 2025, https://developer.nvidia.com/blog/simplify-system-memory-management-with-the-latest-nvidia-gh200-nvl2-enterprise-ra.

[56] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, M. Lipp, and D. Gruss, "Csi: Rowhammer–cryptographic security and integrity against rowhammer," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1702–1718.

[57] A. Saxena, G. Saileshwar, J. Juffinger, A. Kogler, D. Gruss, and M. Qureshi, "Pt-guard: Integrity-protected page tables to defend against breakthrough rowhammer attacks," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023, pp. 95–108.

[58] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. Qureshi, "Safeguard: Reducing the security risk from row-hammer via low-cost integrity protection," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 373–386.

[59] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet lightweight row hammer protection," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1–13.

[60] A. Jaleel, G. Saileshwar, S. W. Keckler, and M. Qureshi, "Pride: Achieving secure rowhammer mitigation with low-cost in-dram trackers," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1157–1172.

[61] M. Qureshi, S. Qazi, and A. Jaleel, " MINT: Securely Mitigating Rowhammer with a Minimalist in-DRAM Tracker ," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 899–914. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MICRO61859.2024.00071

[62] M. Marazzi, P. Jattke, F. Solt, and K. Razavi, "Protrr: Principled yet optimal in-dram target row refresh," in *IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 735–753.

[63] JEDEC, "JESD79-5C," https://www.jedec.org/document_search?search_api_views_fulltext=jesd79-5c, accessed: 2025-01-22.

[64] J. Woo, S. C. Lin, P. J. Nair, A. Jaleel, and G. Saileshwar, "Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025.

[65] M. Qureshi and S. Qazi, "Moat: Securely mitigating rowhammer with per-row activation counters," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 698–714.

[66] O. Canpolat, A. G. Yağlıkçı, G. F. Oliveira, A. Olgun, N. Bostancı, I. E. Yuksel, H. Luo, O. Ergin, and O. Mutlu, "Chronus: Understanding and securing the cutting-edge industry solutions to DRAM read disturbance," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025.

[67] S. Vittal, S. Qazi, P. Das, and M. Qureshi, "MoPAC: Efficiently Mitigating Rowhammer with Probabilistic Activation Counting," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025, p. 723–738. [Online]. Available: https://doi.org/10.1145/3695053.3730997

[68] T. Bennett, S. Saroiu, A. Wolman, and L. Cojocar, "Panopticon: A complete in-dram rowhammer mitigation," in *Workshop on DRAM Security (DRAMSec)*, vol. 22, 2021, p. 110.

[69] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 19–33.

[70] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu, "Vulnerable gpu memory management: Towards recovering raw data from gpu," in *arXiv preprint arXiv:1605.06610*, 2016. [Online]. Available: https://arxiv.org/abs/1605.06610

[71] T. Sorensen and H. Khlaaf, "Leftoverlocals: Listening to llm responses through leaked gpu local memory," in *arXiv preprint arXiv:2401.16603*, 2024. [Online]. Available: https://arxiv.org/abs/2401.16603

[72] Y. Wang, R. Paccagnella, Z. Gang, W. R. Vasquez, D. Kohlbrenner, H. Shacham, and C. W. Fletcher, "GPU.zip: On the side-channel implications of hardware-based graphical data compression," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 3716–3734.

[73] A. Wang, P. Gopalkrishnan, Y. Wang, C. W. Fletcher, H. Shacham, D. Kohlbrenner, and R. Paccagnella, "Pixnapping: Bringing pixel stealing out of the stone age," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2025.

[74] Y. Zhang, R. Nazaraliyev, S. B. Dutta, A. Marquez, K. Barker, and N. Abu-Ghazaleh, "NVBleed: Covert and side-channel attacks on NVIDIA multi-GPU interconnect," *arXiv preprint arXiv:2503.17847*, 2025.

[75] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Spy in the GPU-box: Covert and side channel attacks on multi-GPU systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589080

[76] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "Leaky DNN: Stealing deep-learning model secret with GPU context-switching side-channel," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 125–137.

[77] R. Nazaraliyev, Y. Zhang, S. B. Dutta, A. Marquez, K. Barker, and N. Abu-Ghazaleh, "Not so refreshing: attacking GPUs using RFM rowhammer mitigation," in *Proceedings of the 34th USENIX Conference on Security Symposium (SEC)*, 2025.

[78] M. Tarek Ibn Ziad, S. Damani, A. Jaleel, S. W. Keckler, and M. Stephenson, "Cucatch: A debugging tool for efficiently catching memory safety violations in cuda applications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 124–147, 2023.

[79] B. Di, J. Sun, and H. Chen, "A study of overflow vulnerabilities on GPUs," in *Network and Parallel Computing: 13th IFIP WG 10.3 International Conference, NPC 2016, Xi'an, China, October 28-29, 2016, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 103–115. [Online]. Available: https://doi.org/10.1007/978-3-319-47099-3_9

[80] Y. Guo, Z. Zhang, and J. Yang, "GPU memory exploitation for fun and profit," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 4033–4050. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/guo-yanan

[81] Z. Gu, E. Valdez, S. Ahmed, J. J. Stephen, M. Le, H. Jamjoom, S. Zhao, and Z. Lin, "NVIDIA GPU confidential computing demystified," *arXiv preprint arXiv:2507.02770*, 2025.

[82] J. Danisevskis, M. Piekarska, and J.-P. Seifert, "Dark side of the shader: Mobile GPU-aided malware delivery," in *Information Security and Cryptology – ICISC 2013*, H.-S. Lee and D.-G. Han, Eds. Cham: Springer International Publishing, 2014, pp. 483–495.

[83] B. Hawkes, "Attacking the Qualcomm Adreno GPU," Google Project Zero Blog, September 2020. [Online]. Available: https://googleprojectzero.blogspot.com/2020/09/attacking-qualcomm-adreno-gpu.html

[84] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," in *Proceedings of Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[85] B. Johannesmeyer, R. Isemann, C. Giuffrida, and H. Bos, "Dynamic detection of vulnerable DMA race conditions," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 4499–4513.

[86] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/tatar

[87] Y. Liu, L. Wei, B. Luo, and Q. Xu, "Fault injection attack on deep neural network," in *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD)*, 2017.

[88] M. C. Tol, S. Islam, A. J. Adiletta, B. Sunar, and Z. Zhang, "Don't knock! rowhammer at the backdoor of dnn models," in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[89] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Proflip: Targeted trojan attack with progressive bit flips," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021, pp. 7718–7727.

[90] V. Van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "Guardion: Practical mitigation of DMA-based rowhammer attacks on ARM," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 92–113.

[91] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Can't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 117–130.

[92] C. Bock, F. Brasser, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Rip-rh: Preventing rowhammer-based inter-process attacks," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 561–572.

[93] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "Zebram: comprehensive and compatible software protection against rowhammer attacks," in *OSDI 18*, 2018.

[94] A. Di Dio, K. Koning, H. Bos, and C. Giuffrida, "Copy-on-flip: Hardening ecc memory against rowhammer attacks." in *NDSS*, 2023.

[95] M. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. Ham, J. W. Lee, and J. Ahn, "Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[96] M. Qureshi, A. Rohan, G. Saileshwar, and P. J. Nair, "Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022. [Online]. Available: https://doi.org/10.1145/3470496.3527421

[97] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, "Aqua: Scalable Rowhammer mitigation by quarantining aggressor rows at runtime," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 108–123.

[98] M. Wi, J. Park, S. Ko, M. J. Kim, N. S. Kim, E. Lee, and J. H. Ahn, "SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling," in *HPCA*, 2023.

[99] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[100] J. Woo, G. Saileshwar, and P. J. Nair, "Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems," in *HPCA*, 2023.

[101] A. G. Yağlikçi, M. Patel, J. S. Kim, R. Azizi, A. Olgun, L. Orosa, H. Hassan, J. Park, K. Kanellopoulos, T. Shahroodi, S. Ghose, and O. Mutlu, "Blockhammer: Preventing RowHammer at low cost by blacklisting rapidly-accessed DRAM rows," in *HPCA*, 2021.

[102] M. Marazzi, F. Solt, P. Jattke, K. Takashi, and K. Razavi, "REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations," in *IEEE Symposium on Security and Privacy (SP)*, 2023.

Appendix A.
CPU Privilege Escalation Exploit Details

The adversary uses two threads to repeatedly overwrite entry N and $(N + 16) \bmod 63$ of the `pStatusQueue`, racing the GSP so the driver twice consumes the injected payloads. The attacker must still satisfy the sanity checks in Listing 3. To avoid an unstable kernel caused by the corrupted queue, the adversary should also terminate immediately after privilege escalation if possible.

```
if (_checkSum32(...) != 0)
// Retry if checksum fails.
if (pCmdQueueElement->seqNum != rxSeqNum)
// Retry if sequence number is wrong.
```

Listing 3: Checks in `GspMsgQueueReceiveStatus`.

Scanning the correct sequence number and index N. To satisfy the sequence-number check in Listing 3, the exploit must predict the exact sequence value the driver will expect on its next dequeue. Since the driver's `rxSeqNum` always advances to the sequence of the last successfully dequeued entry, we can recover this value by dumping the `pStatusQueue` via the tampered GPU PTE and locating the entry with the largest sequence number: let this entry be at index i with sequence $s_{\max}$. The driver will next expect $s_{\max} + 1$, which must be placed into entry $N = (i + 1) \bmod 63$. Likewise, the payload bytes that will later be copied into `pMetaData` must be written into the last entry the driver uses for payload consumption, namely index $(i+17) \bmod 63$, corresponding to the original $(i+16)$ payload entry shifted forward by one to align with the upcoming sequence.

```
// pMetaData is used as type msgqMetadata
NvU32 *pReadOutgoing;   NvU32 rxReadPtr;
msgqTxHeader rx {... NvU32 msgCount; }
msgqFcnNotifyRemote fcnNotify; void *
    fcnNotifyArg;
msgqFcnBackendRw fcnBackendRw;
msgqFcnCacheOp fcnFlush; msgqFcnBarrier
    fcnBarrier
```

Listing 4: A subset of fields in `pMetaData`

```
msgqMetadata *pQueue = (msgqMetadata*)handle;
pQueue->rxReadPtr += n; // n=17
if (pQueue->rxReadPtr >= pQueue->rx.msgCount)
    pQueue->rxReadPtr -= pQueue->rx.msgCount;
_backendWrite32(pQueue, pQueue->pReadOutgoing, &
    pQueue->rxReadPtr, ...);
```

Listing 5: 4-byte write in `msgqRxMarkConsumed`.

Writing the 4-byte value with crafted payloads. The definition of the structure pointed by `pMetaData` is in Listing 4. The attacker can control the 17th element in the queue to write the 4-byte value of `rxReadPtr` into the address stored in `pReadOutgoing` at Line 5 of Listing 5 by satisfying the following conditions: (1) set `pReadOutgoing` to point to a target location; (2) choose `rxReadPtr` and `rx.msgCount` so that the value actually written at the call-site equals a desired value after the driver's arithmetic from

Line 2 to Line 4. For example, to write 0, one needs to set `rxReadPtr=0xffffff9` and `rx.msgCount=10`; (3) set `fcnBackendRw` to 0 so the driver performs a write.

Improving kernel stability. Because the injected data corrupts the message-queue state, leaving malformed entries risks unpredictable behavior when the driver later parses them. To prevent further instability after privilege escalation, we deliberately crash the driver by overwriting the `fcnFlush()` (the first callback after the write) with an invalid address. When invoked, this pointer triggers an immediate module crash, terminating the driver before other subsystems can consume the corrupted queue and reducing the chance of a full kernel crash post-escalation.

```
if (pQueue->fcnFlush)
    pQueue->fcnFlush(pQueue->pReadOutgoing, 4);
if (pQueue->fcnBarrier) pQueue->fcnBarrier();
if (pQueue->fcnNotify)
    pQueue->fcnNotify(1, pQueue->fcnNotifyArg);
```

Listing 6: Callbacks in `msgqRxMarkConsumed` after the 4-byte write.

Appendix B. Arbitrary Function Calls

In addition to the arbitrary 4-byte write, the attacker can overwrite callback function pointers in `pMetaData`. Among them, the above `msgqRxMarkConsumed` invokes three callbacks after the 4-byte write if these pointers are configured (Listing 6), which enables the attacker to redirect execution under their control, for further exploitation.

Appendix C. Other Out-of-bound Accesses in the Driver

The following are representative fault buffer fields that can be tampered with to induce OOB accesses.

`gpc_id`. The fault buffer is shared between the GPU and CPU: the GPU writes fault entries via DMA, and the CPU driver reads them. When the driver parses the entry (Listing 7), `gpc_id` is used to calculate the `utlb_id` field without validation: the `UVM_ASSERT` is skipped in the release version. The computed `utlb_id` is then used as an array index in `batch_context->utlbs[]`, leading to OOB memory accesses when `gpc_id` exceeds the number of GPCs on the GPU. We list the OOB accesses below.

```
// In parse_fault_entry_common
buffer_entry->fault_source.gpc_id =
    READ_HWVALUE_MW(fault_entry...);
utlb_id = buffer_entry->fault_source.gpc_id...;
UVM_ASSERT(utlb_id < ...); // Ineffective
buffer_entry->fault_source.utlb_id = utlb_id;
// In fetch_fault_buffer_entries
current_tlb = &batch_context->utlbs[
    current_entry->fault_source.utlb_id];
++current_tlb->num_pending_faults; //OOB W
current_tlb->last_fault = current_entry; //OOB W
```

Listing 7: OOB examples caused by `utlb_id`.

`fault_address`. Similarly, the `fault_address` field is parsed from the fault buffer entry using the same mechanism. The `fault_address` is reconstructed from `ADDR_HI` and `ADDR_LO` fields (Listing 8). When the driver processes the fault, `fault_address` is used to calculate `page_index`, causing OOB accesses.

```
// In parse_fault_entry_common
NvU64 addr_hi, addr_lo;
addr_hi = READ_HWVALUE_MW(fault_entry, ...);
addr_lo = READ_HWVALUE_MW(fault_entry, ...);
buffer_entry->fault_address=(addr_lo+(addr_hi
    ...;
// In service_fault_batch_ats_sub
page_index = (fault_address - sub_batch_base) /
    PAGE_SIZE;
uvm_page_mask_set(read_fault_mask, page_index);
```

Listing 8: An OOB example caused by `fault_address`.

Appendix D. Leaking Model Weights from GPU DRAM

Our goal is to leak LLM weights using our arbitrary GPU-read primitive and identify the base model architecture. We assume the attacker and victim co-locate on the same GPU and victim runs a long-lived inference process.

Method. First, we build per-layer reference fingerprints (mean, standard deviation, fraction of zeros) for the base models by locating layer offsets (by adding unique paddings in test runs) and computing statistics over the dumps from known weights. For an unknown dump, we parse it and calculate similar statistics, and identify the best match via weighted cosine-similarity score.

TABLE 6: Cosine similarity score between LLM model fingerprints (base and fine-tuned models). Values closer to 1 (highlighted in green) indicate highly similar models.

	Variant	Llama2	Llama3	Mistral	Gemma
Llama2 (7B)	Meta*	0.99	-0.20	-0.05	0.29
	Nous	0.99	-0.20	-0.05	0.29
	Meta, Chat	0.99	-0.20	-0.05	0.29
Llama3 (8B)	Meta*	0.06	1.00	0.07	-0.02
	Nous	0.06	1.00	0.07	-0.02
	Meta, Instruct	0.06	1.00	0.07	-0.02
Mistral (7B)	v1.0*	0.94	-0.18	1.00	0.23
	Instruct v1.0	0.94	-0.18	1.00	0.25
	OpenHermes 2.5	0.94	-0.18	1.00	0.23
Gemma (7B)	Google*	-0.04	-0.20	-0.08	1.00
	Google, Instruct	-0.03	-0.16	-0.04	0.97

* Baseline models used to generate the reference fingerprint.

Evaluation. We test four 7B LLM families (∼16 GB parameters). A complete dump takes ∼1.5 minutes while fingerprinting (statistics + comparison) takes ∼15 minutes. Results show that across families, we can reliably identify the model families (within-family similarity >0.99) from fine-tuned models, using LLM weights dumped. Future works can explore attacks like recovering ML model functionality, and membership inference, from leaked LLM model weights.

E. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

This paper explores how attackers can use Rowhammer bit flips for privilege escalation attacks on an NVIDIA GPU. To accomplish this, they developed techniques for massaging page tables into targeted locations in the GPU's memory so that they could flip a bit inside of a page table on the GPU. Using this capability, they are able to write to memory on both the GPU and CPU, and demonstrate how to tamper with ML models, steal cryptographic keys, and gain escalated privileges on the CPU.

E.2. Scientific Contributions

- 5. Identifies an Impactful Vulnerability.
- 6. Provides a Valuable Step Forward in an Established Field.

E.3. Reasons for Acceptance

1) Identifies an Impactful Vulnerability: This paper shows how using Rowhammer to induce bit flips on a GPU can result in GPU-side privilege escalation, which can then be used for CPU-side privilege escalation. Additionally, the discovery of the GSP message queue OOB vulnerability in the NVIDIA driver is a notable contribution to software security research.
2) Provides a Valuable Step Forward in an Established Field: Whereas prior work was limited to corrupting GPU applications, this paper shows that the implications of bit flips on GPU memory are more severe than previously understood, with consequences for the host CPU.

E.4. Noteworthy Concerns

1) While the authors evaluated their privilege escalation exploit on multiple driver versions and GPUs using simulated bit flips, they only demonstrated bit flips and an end-to-end compromise on a single GPU.
2) Their attacks for degrading ML models and extracting keys from PQcrypto are not particularly impactful, as the arbitrary read/write primitive they present subsumes both.