GeForge: Hammering GDDR Memory to Forge GPU Page Tables for Fun and Profit

Junpeng Wan[1,2], Yanan Guo[3], Zhi Zhang[4], Zhuo Li[5], Dave (Jing) Tian[2], Zhenkai Zhang[1]

[1]*Clemson University,* [2]*Purdue University,* [3]*University of Rochester,*
[4]*University of Western Australia,* [5]*HydroX AI*

***Abstract*—Over the years, Rowhammer has been leveraged to mount a wide range of attacks against system main memory. While a recent study has revealed that GPU memory is similarly vulnerable, the security implications remain largely under-explored. To advance this line of research, we introduce `GeForge`, an end-to-end Rowhammer attack that exploits bit flips induced in GPU memory to achieve system-level compromise. At its core, `GeForge` corrupts GPU page tables to seize control of address translation, enabling arbitrary access to the entire GPU memory. Moreover, by exploiting a special mapping feature in the GPU page table, `GeForge` extends its reach to directly access host memory.**

To make `GeForge` practical under default system settings, we develop novel techniques that eliminate restrictive assumptions in prior work. Our techniques include a method for aligning offline-profiled physical address mappings to runtime GPU allocations and a memory massaging strategy that steers target GPU page table structures into vulnerable locations via the stock driver allocator. In addition, we improve the hammering pattern to trigger many more bit flips than prior work. With these approaches, we successfully mount `GeForge` on widely deployed NVIDIA GPUs, including both workstation-class and consumer-grade ones. We show that `GeForge` allows an attacker to arbitrarily read and modify data across GPU contexts. More crucially, we demonstrate that `GeForge` can help the attacker escalate privileges to root on the host system.

1. Introduction

As DRAM has continuously improved in performance and density, it has become increasingly vulnerable to an electrical disturbance phenomenon known as Rowhammer [23]. Over the last decade, Rowhammer has been exploited to mount a wide range of powerful attacks, including privilege escalation [6, 14, 46, 49], sandbox escapes [3, 9, 11, 15, 46], cryptographic key extraction [2, 26, 43], neural network degradation [29, 42, 54], and denial-of-service [16, 30, 55]. Despite this extensive body of work, nearly all demonstrations have targeted CPU-side memory systems. The question of whether Rowhammer can be triggered in GPU memory has long been unanswered.

Very recently, Lin *et al.* demonstrated that Rowhammer can indeed be triggered in modern GPUs with GDDR6 [29]. In their work, they achieved 8 bit flips across four GDDR6 banks on an NVIDIA workstation-class GPU, the RTX A6000, and they also showcased how to exploit those bit flips to degrade the performance of neural network models. While this is the first Rowhammer attack on a GPU, multiple critical limitations exist.

First, their work assumes that the mapping from GPU virtual addresses to GDDR6 rows and banks stays fixed if a large portion of GPU memory is allocated. Although they verified it in their own setup, we find that several environmental factors can actually shift the starting physical address of allocations, thus breaking this assumption. Second, their memory massaging approach hinges on the use of a customized GPU memory allocator, whose behavior differs significantly from the allocator in the NVIDIA driver. Therefore, under the default setting, their placement control of victim data will not work. Third, they only succeeded in flipping a few bits (8 in total across four GDDR6 banks) and leveraged them solely to disrupt machine learning (ML) inference. While their attacks are undoubtedly concerning, they do not establish or attempt to examine the kind of severe system-level threats that CPU-side Rowhammer exploits have demonstrated.

Considering the gaps remaining in [29], we aim to further this line of research with two objectives. ① We seek to exploit the bit flips triggered in GPU memory to mount more powerful attacks than the model degradation attack presented in [29]. ② We want to remove the restrictive assumptions made in [29] to enable attacks under ordinary runtime conditions. In this paper, we show that both goals can be achieved.

Specifically, we demonstrate that GPU page tables serve as a high-leverage Rowhammer target. Carefully corrupting their entries can culminate in full control of GPU address translation, allowing the attacker to arbitrarily read and write the entire GPU physical memory. However, we find that the implications can extend far beyond the GPU itself. Essentially, NVIDIA GPU page tables expose a mapping feature called the system aperture, which allows the GPU to directly access host memory. Therefore, an attacker with control over GPU page tables can craft system aperture mappings to compromise data integrity in host memory, ultimately achieving privilege escalation. Although highly powerful, realizing such an attack on commodity systems is non-trivial and requires overcoming several technical barriers.

An effective approach to identifying same-bank ad-

dresses of GPU memory is introduced in [29], but it is too costly (∼4 hours per bank) for use in online attacks. Fortunately, given a specific GPU model, all its cards, regardless of vendor, have an identical memory organization, and thus we can perform offline profiling once to recover which physical locations fall into the same bank and reuse this map thereafter. However, this approach requires determining the physical addresses of allocated memory at runtime — a challenge that prior work [29] sidesteps under an assumption we refute in Section 4. To address this challenge, we propose a page anchoring technique. The key observation is that NVIDIA GPUs map physical addresses to L2 cache sets in a highly non-linear fashion, so each page frame exhibits characteristic eviction-set patterns for its first few cache lines. Matching these patterns helps us pinpoint target physical addresses within allocated GPU memory.

The second problem is that even though the Rowhammer vulnerability has been proven to exist in GDDR6, it appears very hard to trigger with the many-sided hammering approach presented in [29], which activates each aggressor row uniformly per refresh interval. Inspired by the study in [17], we develop a more effective hammering strategy to overcome this barrier. In essence, rather than confining hammering patterns to a single refresh interval as in [29], we let them span multiple intervals; within each pattern, we tune the order and intensity of row activations to yield a non-uniform hammering schedule. Using this new approach, we can induce significantly more bit flips compared with prior work.

Despite many more bit flips, weaponizing them against GPU page tables is still far from straightforward: The stock NVIDIA driver reserves a low-memory region for storing page tables, around which we cannot allocate aggressor rows for hammering; accordingly, we must find a way to exhaust this pool first to force new table structures into areas accessible to us. Even then, we need a precise positioning method to trick the driver into landing GPU page table structures at vulnerable locations where bit flips occur. Moreover, random corruption may not produce a useful outcome, so we should have a strategy for maximizing the probability of converting opportunistic bit flips into deterministic control over address translation. Taking into account these problems, we devise a memory massaging technique that can effectively deplete the driver's original pool and steer page table structures into chosen locations. The technique pivots on two properties of CUDA's Unified Virtual Memory (UVM) — its ability to carve out large, unbacked swaths of virtual address space and its support for small page sizes with stride-controllable instantiation. Additionally, instead of last-level page table entries, we target entries at an intermediate level, which, as we show, makes it easier for a single bit flip to give us control over GPU address translation.

Having overcome these obstacles, we present `GeForge`, the first end-to-end Rowhammer attack on GPU page tables. Prior work triggered bit flips on NVIDIA's workstation-class A6000 but reported failures on other GPU models. In our work, not only do we successfully mount `GeForge` on the A6000, but also on a mainstream RTX 3060, the most widely deployed discrete GPU to date [35], showing its potential threat to commodity systems at scale. By manipulating GPU address translation, we launch attacks that breach confidentiality and integrity across GPU contexts. More significantly, we forge system aperture mappings in corrupted GPU page tables to access host physical memory, enabling user-to-root escalation on Linux. To our knowledge, this is the first GPU-side Rowhammer exploit that achieves host privilege escalation.

The main contributions of this paper are as follows:

- We devise a GPU page anchoring approach that leverages the non-linearity property of L2 cache addressing to localize target page frames within allocated GPU memory at runtime.
- We design refresh-synchronized, non-uniform hammering patterns that induce bit flips in GDDR6 more effectively than prior work, with which we observe more than 1,100 bit flips on an RTX 3060 and over 200 bit flips on an RTX A6000.
- We introduce a GPU memory massaging strategy that can effectively steer page table structures into chosen physical locations under the default driver settings and can create a memory layout favorable for exploiting a single bit flip to gain control over GPU page tables.
- We demonstrate the first end-to-end Rowhammer attack on GPU page tables, which enables arbitrary memory accesses across GPU contexts and further achieves host privilege escalation from an unprivileged user to root.

Availability: The source code of `GeForge` is available at https://github.com/stefan1wan/GeForge.

2. Background

2.1. GPU Architecture

GPUs achieve their computational power through massive parallelism. The basic compute unit in a GPU is the streaming multiprocessor (SM), which consists of multiple simple cores. A modern GPU typically contains dozens of SMs. Each SM organizes and executes threads in groups referred to as warps. For NVIDIA GPUs, each warp comprises 32 threads. The execution of threads in a warp follows the single-instruction, multiple-thread (SIMT) model (i.e., they run in lockstep).

Each SM has its own L1 cache, a portion of which can be configured as scratchpad memory. All SMs share an L2 cache. In NVIDIA GPUs, the cache line size is 128 B [57]. Beyond the cache hierarchy, GPUs use off-chip memory to store data.

As with a CPU's system memory, GPU memory is also based on DRAM. However, the DRAM used in GPUs is optimized for high bandwidth at the cost of longer access latency than its CPU counterpart. Currently, GDDR6 is the most widely used DRAM type in NVIDIA's consumer-grade and workstation-class GPUs. Unlike on the CPU side, where DRAM chips are assembled into dual in-line memory modules (DIMMs), GPU DRAM chips are soldered directly on

the board, and each of them is connected to the GPU through an individual memory controller. Since these DRAM chips are distributed around the GPU SoC, the latency of accessing them is not uniform across chips, a phenomenon referred to as a NUMA effect [29].

2.2. GPU Address Translation

Similar to the CPU side, GPU memory is virtualized and managed via paging. Each running GPU program, referred to as a GPU context, has a multi-level page table. When an SM generates a virtual address, the GPU memory management unit (GMMU) translates it to a physical address by consulting the corresponding page table. These page tables are maintained by the GPU driver and are inaccessible to unprivileged users.

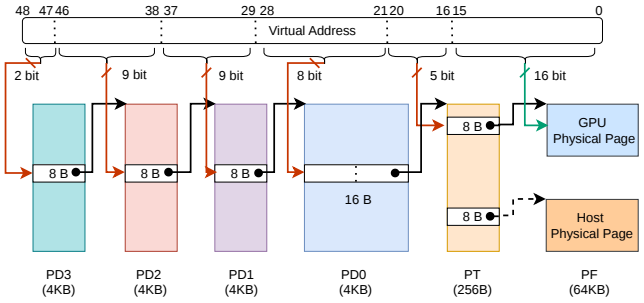


Figure 1: Address translation for 64 KB pages in Ampere GPUs.

NVIDIA GPUs support multiple page sizes, including 2 MB and 64 KB. By default, 2 MB pages are used for both GPU code and data. Yet, CUDA's Unified Virtual Memory (UVM) feature allows the allocation of 64 KB pages [56]. The page table hierarchy depth may vary depending on the page size. For example, in NVIDIA Ampere GPUs, a four-level page table is used for translating 2 MB pages, while a five-level one is used for 64 KB pages.

Figure 1 illustrates the five-level page table walk for 64 KB pages in Ampere GPUs such as the RTX 3060 and A6000. In these GPUs, a 49-bit virtual address space is provided. Given a virtual address, bits [48:47], [46:38], [37:29], and [28:21] index entries across the four levels of page directories (PD3, PD2, PD1, and PD0, respectively), while bits [20:16] index the page table (PT) to identify the page table entry (PTE) that contains the physical address of the 64 KB page frame. By contrast, Figure 9 in Appendix A shows the page table walk for 2 MB pages.

Note that NVIDIA GPU PTEs include an aperture field that specifies the memory domain of the mapping. When the aperture is set to "system", the GMMU treats the 64 KB page as residing in host memory. In this case, the PTE's address field is interpreted as a host physical (DMA) address, and the GPU services access the page by issuing PCIe transactions to that address. Such a system aperture is needed for CUDA features like zero-copy access, in which mapped page-locked host memory can be directly accessed by the GPU instead of being copied into local device memory first.

2.3. Rowhammer

CPUs typically use DDR memory (e.g., DDR4) while GPUs use GDDR memory (e.g., GDDR6); both are types of DRAM. Each DRAM chip consists of multiple banks, and each bank can be viewed as a two-dimensional array of cells organized in rows and columns. Each cell has a capacitor (and an access transistor). The capacitor is either charged or uncharged to represent a binary value (the charged state represents '1' for true-cells and '0' for anti-cells). As the capacitor gradually leaks charge, cells must be refreshed regularly to prevent data loss. For example, the refresh period must not exceed 64 ms for DDR4 and 32 ms for GDDR6.

Each bank also contains a row buffer that can hold the contents of a single row. To access a cell, the corresponding row must first be activated to load its contents into the row buffer, after which the access is served from the buffer. Thus, alternately accessing data in two rows of the same bank needs to toggle the row buffer contents back and forth, resulting in repeated activations of these two rows.

In 2014, Kim *et al.* showed that frequently activating ("hammering") a DRAM row can actually accelerate charge leakage in physically adjacent rows, causing stored bits to flip — a phenomenon known as Rowhammer [23]. The hammered rows are often called aggressor rows, whereas rows with flipped bits are usually called victim rows. Hammering is double-sided when two aggressor rows sandwich a victim row on both sides, and single-sided when the hammered rows do not form such a flanking pair. Double-sided hammering is generally more effective than its single-sided counterpart at triggering bit flips.

Starting with DDR4, DRAM vendors introduced target row refresh (TRR) to mitigate Rowhammer. TRR can effectively thwart single-sided and double-sided hammering by tracking frequently activated rows and proactively refreshing their neighbors. However, TRR implementations can only track a limited number of aggressor rows. Many-sided hammering exploits this limitation by spreading activations across multiple rows within the same bank, exceeding TRR's tracking capacity and successfully inducing bit flips [12]. GDDR6 also implements TRR mechanisms, which can still be bypassed by many-sided hammering [29].

3. Overview

In this section, we first specify the threat model and then describe the attack workflow of `GeForge`. The workflow outlines how `GeForge` leverages Rowhammer-induced bit flips to construct attack primitives, with implementation details provided in subsequent sections.

3.1. Threat Model

We assume an attacker with a non-privileged account on a system equipped with an NVIDIA GPU running the stock driver. The attacker can submit CUDA workloads using only ordinary CUDA APIs, without requiring special privileges or non-standard drivers. Moreover, when the GPU is otherwise

idle, the attacker can allocate large GPU buffers to occupy a substantial fraction of physical GPU memory.

We also assume that the GPU operates without ECC. This is inherently true for most consumer-grade GPUs like the RTX 3060, as ECC is not supported on them. For GPUs that do support ECC, like the workstation-class RTX A6000, ECC is disabled by default unless explicitly enabled.

We introduce two categories of attacks with `GeForge`: GPU-local attacks that compromise other concurrent GPU contexts, and host-directed attacks that achieve system-wide privilege escalation. While GPU-local attacks are feasible regardless of system configuration, our host compromise attack requires that the system's IOMMU is off or not enforcing isolation between the GPU and host memory. This assumption is actually not unrealistic in practice, as many consumer desktops and workstations ship with IOMMU disabled by default for compatibility and performance reasons.

For instance, a large number of Linux systems (including Ubuntu 22.04 LTS) operate without IOMMU enforcement by default unless explicitly enabled via kernel parameters. In fact, even on systems where IOMMU is initially on, users often disable it to avoid performance overhead or resolve compatibility issues with certain applications. It is reported that this configuration, whether disabled by default or subsequently disabled by users, remains widespread across commodity systems [32, 39].

3.2. Attack Workflow

`GeForge` proceeds through the following five steps:

① The attacker performs offline profiling in a controlled environment to recover the mapping from GPU physical addresses to GDDR banks and rows for the target GPU model.

② The attacker allocates a large GPU buffer in the attack environment and uses our page anchoring technique to locate a specific page frame within the allocation, which allows the offline-profiled address map to be applied at runtime.

③ The attacker then performs online memory templating to identify bit-flip-prone memory locations, along with the corresponding aggressor rows and hammering patterns.

④ The attacker utilizes our memory massaging strategy to steer GPU page table structures into vulnerable memory locations and then performs Rowhammer to corrupt a page directory entry, redirecting it to a forged page table under the attacker's control.

⑤ With the forged page table in place, the attacker constructs attack primitives that enable read and write access to arbitrary GPU and host physical addresses, which are then used to carry out end-to-end exploitation.

4. GPU Page Anchoring

As a prerequisite for `GeForge`, we must identify GPU memory addresses that fall in the same GDDR6 bank but different rows for hammering. In [29], an effective two-stage approach is presented: first isolate same-chip addresses using NUMA-like latency differences, and then exploit row buffer conflict timing to determine which of them are in the same bank. Yet, this approach takes approximately four hours per bank on GPUs, which is too costly for online attacks.

Fortunately, in contrast to CPUs where memory topology and controller policies vary with DIMM population and BIOS settings, GPU memory organization is fixed per model. We can thus profile a given GPU model offline to recover the mapping from physical addresses to GDDR6 banks and rows. Once this per-model map is in hand, `GeForge` can leverage it at runtime, provided we can pinpoint which GPU page frames in the allocation correspond to those profiled offline.

Note that, in [29], Lin *et al.* also employ offline profiling and reuse the results at runtime. However, rather than solving the problem of establishing correspondence between runtime page frames and offline-profiled ones, they simply sidestep it by arguing: (i) the GPU memory allocator of the NVIDIA driver tends to return physically contiguous chunks; and (ii) for large GPU memory allocations, the driver tends to return exactly the same page frames in the same order. Based on these two assumptions, they treat offsets within a sufficiently large buffer as directly reflecting physical addresses.

We have performed experiments to check both assumptions. While our results confirmed the first one (i.e., physical contiguity),[1] we found that the second is very unreliable in practice. Using the tool from [56], we in fact observed that several environmental factors, including driver and OS kernel versions, affect the starting physical address of allocations, causing shifts of tens of megabytes. Thus, the virtual offset approach developed in [29] cannot consistently apply the profiled map at runtime.

Because physical contiguity has been verified, reusing the offline profiling results requires only identifying which page of the allocated GPU memory corresponds to the first page frame used during offline profiling. Given that our threat model assumes the ability to allocate a large portion of GPU memory, we can strategically select a starting page frame that is guaranteed to fall within any sufficiently large allocation. We call this page frame an anchor. The challenge then reduces to locating this specific anchor page frame within the allocated region at runtime.

We find that, in NVIDIA GPUs, physical addresses are non-linearly mapped to L2 cache sets, and we exploit this property to develop a technique that can efficiently localize the anchor. Essentially, the non-linearity causes each page frame to exhibit characteristic eviction set patterns for its first few memory blocks. Using the INVALIDATE+COMPARE primitive [57], we can reliably test if an eviction set successfully achieves eviction. Note that these patterns are actually not globally unique, but due to the non-linearity, page frames that share a pattern are spaced distinctively. By combining the eviction set pattern with the inter-frame spacing, we can pinpoint the anchor.

1. Unlike on the CPU side, physical contiguity does not, by itself, directly enable same-bank row identification on the GPU side, because we find that it is extremely difficult to reverse-engineer the mapping function from physical addresses to GDDR6 banks. The function is not only highly non-linear but also mixes (nearly) all address bits.

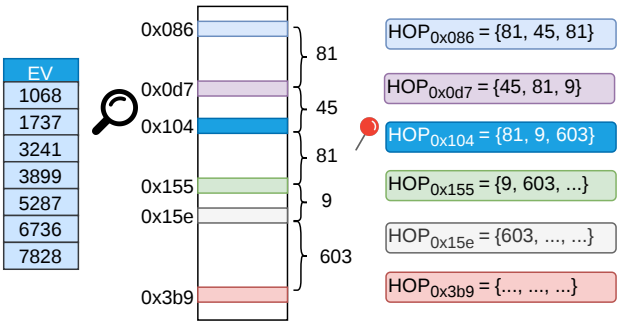


Figure 2: Locating the anchor page frame on the RTX 3060. The eviction set EV matches six page frames in the first 2 GB of GPU memory, and the inter-frame hop sequence uniquely distinguishes each candidate.

For example, suppose we offline-profile the RTX 3060 GPU model starting at the physical address `0x20800000`; namely, the 2 MB page frame `0x104` serves as the anchor. During offline profiling, we construct the eviction set EV for the first 128 B block of the anchor, as shown on the left of Figure 2. Each element x in EV represents the offset (in units of 128 B blocks) from the beginning of the page under test. In the first 2 GB of GPU physical memory, we find that this set evicts the first block of six page frames. However, as shown in Figure 2, the spacing between these matching page frames, measured in hops of 2 MB units, forms a distinctive pattern. At runtime, given a large contiguous allocation, we linearly scan its 2 MB pages. For each page p, we use INVALIDATE+COMPARE to test whether EV evicts p's first 128 B block. If successful, we verify whether the eviction set also works at page $p+81$, then at page $p+81+9$, and if needed, at page $p+81+9+603$. The first page p that matches both the eviction behavior and the complete hop sequence uniquely identifies the anchor page frame `0x104`.

5. GPU Memory Templating

With the bank/row mapping recovered offline and aligned to the runtime allocation via page anchoring, `GeForge` can proceed to GPU memory templating, whose goal is to identify Rowhammer-vulnerable bit locations of the target GPU. To this end, we build a fuzzer that can be tuned for templating different GPUs. In this section, we first introduce the hammering patterns used by the fuzzer and then describe its operation and configuration parameters.

5.1. Non-Uniform Hammering Pattern

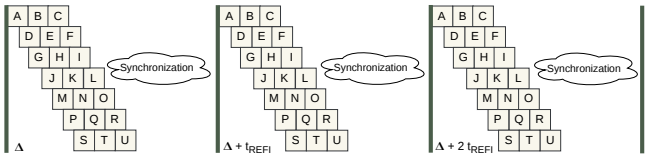
No matter whether Rowhammer attacks are mounted on the CPU or the GPU, achieving a very high row activation rate is essential. On CPUs, a single thread can often drive row activations close to the maximum rate allowed by DRAM timing constraints, because the memory access latency is comparable to the row cycle time t_{RC} (i.e., the minimum interval between two activations to the same bank). On GPUs, however, the end-to-end access latency from an SM to GPU memory is much larger than t_{RC}. As a result, a single GPU thread can generate only a limited number of row activations within one refresh window, leaving the memory controller idle for much of the time. To overcome this limitation, prior work [29] takes advantage of multiple warps to pipeline row activations: while some warps are stalled waiting for memory responses, others continue issuing new requests. This keeps the memory controller busy and raises the activation rate toward the theoretical maximum.

Even with a high row activation rate enabled by multiple warps, triggering Rowhammer bit flips in GPU memory still faces a critical challenge, that is, circumventing the TRR mitigation deployed in GDDR6 chips (see Section 2.3). To bypass TRR, hammering patterns must be many-sided at a minimum [12], and can further benefit from synchronization with DRAM refresh operations [9]. Indeed, the hammering pattern designed in [29] incorporates both properties, and successfully triggers bit flips in the GDDR6 memory of an RTX A6000 GPU.

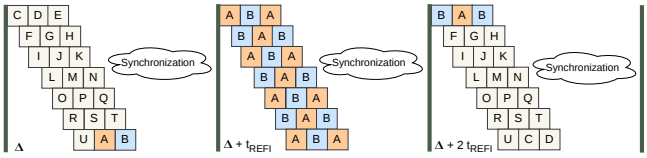
Figure 3a illustrates the hammering pattern of [29] using a 21-sided example. To hammer the set of 21 aggressor rows, the pattern launches 7 warps, each using 3 threads to target 3 distinct rows. The pattern is highly uniform: within each refresh interval t_{REFI}, each thread activates its designated row the same number of times in a fixed sequence, and the assignment of rows to threads remains unchanged across intervals. To synchronize hammering with DRAM refresh operations, the pattern inserts timing bubbles using dummy additions to ensure that each hammering round begins immediately after a refresh event.

Even though the hammering pattern of [29] can induce bit flips in GDDR6, its yield appears low: only eight bit flips across four banks have been reported in [29]. Inspired by Blacksmith [17], we suspect that this limited performance is, at least in part, due to the pattern's high uniformity, which likely constrains the intensity and order of the most effective aggressor row activations. We therefore take the hammering pattern of [29] as a vanilla baseline and extend it with the frequency-domain insights from Blacksmith [17] to construct a non-uniform hammering pattern.

Given n aggressor rows for n-sided hammering, our non-uniform pattern first selects two of them as a target pair. (The two rows in the target pair are at least one row apart.) Unlike the vanilla pattern, which always repeats the same access schedule every t_{REFI}, we construct the hammering pattern over X consecutive t_{REFI} intervals, where $X \geq 1$. Suppose W warps are used, each dedicating T threads to hammering, where $W \times T = n$. Then, across the X t_{REFI} intervals, the pattern will contain $W \times T \times X$ access slots for aggressor row activations. We first assign the target aggressor row pair to these slots. To control this assignment, we introduce two parameters: the phase ϕ, which specifies the offset of the target pair within the $W \times T \times X$ slots, and the amplitude A, which specifies how many times the target pair is repeated. After placing the target pair, we fill the remaining slots with accesses to the other aggressor rows in a round-robin fashion. The resulting non-uniform pattern will repeat every X t_{REFI} intervals throughout the hammering process.



(a) The vanilla hammering pattern of [29] over three t_{REFI} intervals.



(b) Our non-uniform hammering pattern over three t_{REFI} intervals.

Figure 3: Comparison between patterns for 21-sided hammering. Each warp is shown on a separate horizontal track (7 warps in total), and each square represents a thread (3 threads per warp). Letters A – U denote different rows in the same GDDR6 bank. Δ denotes the refresh-aligned starting point of the hammering schedule.

Figure 3b shows an example of our non-uniform hammering pattern that works well in practice to induce bit flips in GPU memory. The numbers of aggressor rows, warps, and threads are the same as those in the vanilla-pattern example of Figure 3a, namely, 21, 7, and 3, respectively. We set the non-uniform pattern to span three t_{REFI} intervals, where the target pair (i.e., aggressor rows A and B) starts at a phase of $\phi = 20$ and is repeated with an amplitude of $A = 13$. Compared to the vanilla pattern, where rows A and B are only activated three times over three refresh intervals, our non-uniform pattern activates them 13 times. This significantly increases the likelihood of triggering Rowhammer bit flips in their neighboring victim rows. Indeed, our evaluation confirms that this non-uniform hammering pattern can induce bit flips that the hammering pattern of [29] cannot trigger.

5.2. `GeForge` Fuzzer

To perform GPU memory templating, we build a fuzzer that generates non-uniform hammering patterns for a GPU's GDDR6 memory. Under our threat model (see Section 3.1), the attacker can allocate a substantial fraction of GPU memory when the device is otherwise idle (e.g., 11 of the 12 GB on the RTX 3060 and 47 of the 48 GB on the RTX A6000). The fuzzer reserves such a region using the standard CUDA API `cudaMalloc()`. Since the physical addresses of the allocated pages are not visible to the unprivileged attacker, the fuzzer first applies our page anchoring technique (see Section 4) to locate the anchor page frame within this large runtime allocation. With the anchor identified, the fuzzer can reuse the bank/row mapping obtained from offline profiling for the same GPU model.

During fuzzing, in each round, the fuzzer randomly selects a target bank and configures a set of execution parameters to generate a hammering pattern. A key parameter is the number of aggressor rows n for n-sided hammering. However, our fuzzer does not choose n directly. Instead, it selects two parameters: the number of warps W and the number of hammering threads per warp T, such that $n = W \times T$. Due to the timing constraints of GDDR6 and the bandwidth limit of the GPU's per-chip memory controller, we restrict W and T to $5 \leq W \leq 11$ and $2 \leq T \leq 5$ respectively, while ensuring that $20 \leq W \times T \leq 30$. Next, the fuzzer selects a distance d, where $2 \leq d \leq 8$, and sweeps through the target bank, advancing the starting aggressor row one step at a time. For each starting row, it derives the corresponding aggressor row set by spacing n aggressor rows at intervals of d.

Once a set of aggressor rows has been determined, the fuzzer designates its first two rows as the target pair and selects the number of consecutive t_{REFI} intervals X over which the non-uniform pattern spans. Empirically, we find the range $1 \leq X \leq 3$ to be the most effective[2] (on both the RTX 3060 and A6000), although X can in principle be larger. With X selected, the total number of row access slots is $W \times T \times X$. The fuzzer then selects the amplitude A to control how many slots the target pair consecutively occupies. We constrain A to $1 \leq A \leq 15$, while ensuring that the remaining slots (i.e., $W \times T \times X - 2A$) can still accommodate at least one access to each of the other aggressor rows. Finally, the phase ϕ is randomly chosen, subject to the constraint that the target pair's $2A$ consecutive slots fit within the pattern.

With a specific non-uniform hammering pattern created, the fuzzer treats all rows adjacent to the aggressor rows as victim rows. Before hammering, the fuzzer initializes every byte in each aggressor row to one data pattern and every byte in each victim row to the complementary pattern. We employ two pairs of data patterns, `0xFF`/`0x00` and `0x55`/`0xAA`, so that each hammering pattern is evaluated four times in total. The fuzzer then launches a CUDA kernel to execute the hammering pattern on the initialized data. Note that, although the non-uniform pattern spans X refresh intervals, dummy additions are still inserted as timing bubbles between successive intervals to synchronize hammering with each refresh event. After hammering, the fuzzer scans the victim rows for bit flips and records all observed ones as memory templating results. Algorithm 1 in Appendix E provides more details on the fuzzer.

Note that the distance d selected by the fuzzer can exceed 2, and from our templating results, we observe three possible aggressor-victim layouts. In many cases, the victim row is flanked by aggressor rows on both sides, yielding a double-sided layout; in other cases, it is adjacent to only one aggressor row, yielding a single-sided layout. Interestingly, we also observe a small fraction of flips induced by remote-sided aggressors located at a distance of 2 or more.

6. GPU Memory Massaging

Having identified vulnerable GPU memory locations via memory templating, we now turn to weaponizing them. On the CPU side, page tables have long been recognized as a highly lucrative Rowhammer target [24, 46, 49, 52, 60]. GPU page tables are similarly attractive, as carefully corrupting

2. If $X = 1$, the non-uniform pattern reduces to the vanilla one of [29].

them can grant arbitrary access to GPU physical memory or even host memory via the system aperture (see Section 2.2). We therefore focus on exploiting GPU Rowhammer to seize control over GPU page tables. Realizing this goal, however, requires addressing two key problems.

First, as shown in Figure 1, the GPU page table hierarchy comprises multiple levels of structures, which one should be targeted for maximum leverage? Second, because GPU page tables are allocated by the driver in low physical memory regions, how can the chosen structure be steered to vulnerable locations for hammering? We answer them in this section.

6.1. Target Structure and Bit Flip Candidates

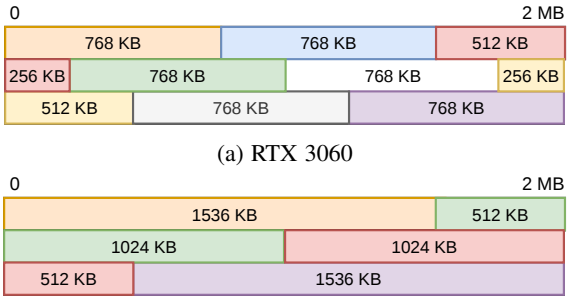
In principle, Rowhammer can be exploited to corrupt entries at any level of the GPU page table hierarchy, including PD3, PD2, PD1, PD0, and PT (see Figure 1). In `GeForge`, however, we deliberately target PD0 entries, as they arguably offer the best balance between leverage and exploitability.

Compared with PT entries, corrupting PD0 entries provides a more direct path to gaining control. A PT entry stores a pointer to a mapped page frame, so corrupting it becomes truly exploitable *only if* the redirected page frame already contains, or can be massaged to contain, another GPU page table structure. In contrast, a PD0 entry stores a pointer to a PT, and once a bit flip redirects this pointer into any accessible page frame, the page frame can directly serve as a forged PT to be filled with crafted PTEs. As our allocations occupy a large portion of GPU memory, a corrupted PD0 pointer is statistically very likely to land in memory under our control.

As for entries in PD1, PD2, and PD3, corrupting them could similarly redirect pointers into our allocated memory, allowing us to forge the downstream page table structures. However, the higher the level, the larger the virtual address range each entry covers. As shown in Section 6.2, steering a page directory allocation into a specific vulnerable physical location requires virtual memory accesses proportional to its coverage. Accordingly, targeting higher-level page directory entries demands controlling a much larger portion of the virtual address space, making memory massaging less practical. Hence, we choose PD0 as the target in `GeForge`.

With PD0 entry corruption as the goal, we next screen the vulnerable memory locations identified during templating to select suitable steering destinations for memory massaging. Not every discovered bit flip can be used for corrupting PD0 entries online. In this selection, we consider two constraints.

(1) Accessibility of aggressor rows. We observe that page directories, whether PD0, PD1, PD2, or PD3, are allocated as 4 KB chunks within 2 MB page frames. Hence, as detailed in Section 6.2, our memory massaging needs to choose a 2 MB page frame containing Rowhammer-vulnerable locations and steer the target PD0 allocation into it. Once a 2 MB physical page is employed by the driver to host page table structures, it is no longer accessible to users. We notice that, for many GPUs, a single 2 MB page frame can span multiple rows of the same GDDR6 bank. As a result, when selecting steering destinations, we must ensure that the candidate 2 MB frame does not encompass aggressor rows required for hammering the victim row in it.



(a) RTX 3060

(b) RTX A6000

Figure 4: Row stripe layouts across three consecutive 2 MB frames on two NVIDIA GPUs. Each color represents a distinct row stripe.

We emphasize that the severity of this aggressor row accessibility issue varies across GPU models, depending on the size of their GDDR6 row stripes. In GPU physical memory, the rows with the same row ID across all GDDR6 banks are laid out as one contiguous physical address region, which we call a row stripe.[3] A single row stripe thus occupies $S_R \times N_B$ bytes in total, where S_R is the per-bank row size and N_B is the total number of banks across all GDDR6 chips. Figure 4 illustrates how row stripes form three consecutive 2 MB page frames on the RTX 3060 and RTX A6000. On the RTX 3060, we find that its S_R is 4 KB (i.e., its GDDR6 chips operate in x16 mode [48]) and its N_B is 192, so its row stripe is 768 KB. As a result, a single 2 MB page frame on the RTX 3060 covers at least two consecutive rows in each bank. In contrast, on the RTX A6000, we find that its S_R is 2 KB (i.e., its GDDR6 chips are set in x8 mode [48]) and its N_B is 768, yielding a row stripe of 1536 KB. Such a stripe size actually allows the two aggressor rows adjacent to some victim rows to fall into different 2 MB page frames.

Consequently, for GPUs with a relatively small row stripe such as the RTX 3060, bit flips that require double-sided aggressors sandwiching the victim row cannot be exploited, because at least one aggressor row inevitably co-resides within the same 2 MB page and becomes inaccessible during massaging; in this case, we should retain only bit flips that can be triggered by a single aggressor row residing outside the victim's 2 MB page frame. On the other hand, for GPUs with a sufficiently large row stripe like the RTX A6000, a broader set of bit flips can be exploited, since double-sided aggressor configurations become viable alongside single-sided ones. In Appendix B, we give more details on the row stripes of both GPU models and their aggressor row accessibility.

(2) Validity of corrupted PD0 entry. Filtering for bit flips that remain triggerable online during massaging is necessary but not sufficient for exploitation, as some of these flips may corrupt a PD0 entry into an invalid state (e.g., causing its

3. In a row stripe, the data is not laid out as one bank's complete row followed by the next. Instead, each bank's row is divided into smaller chunks, and these chunks are interleaved across banks.

pointer to fall outside the legitimate physical memory range). We thus further refine the selection of bit flip candidates by considering the structure and semantics of PD0 entries.

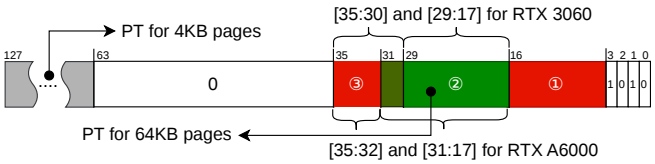


Figure 5: PD0 entry layout and exploitable bit flip regions.

As illustrated in Figure 5, each PD0 entry is 16 bytes in size and is logically divided into two 8-byte halves to support both 4 KB and 64 KB page sizes. The lower 8 B half points to the PT responsible for translating 64 KB pages, whereas the upper 8 B half points to the PT for 4 KB pages. Since 64 KB pages, but not 4 KB ones, can be instantiated from user space via UVM [56], the page table structures created during our memory massaging contain valid states only in the lower 8 B half of their PD0 entries, with the upper half remaining all zeros. Accordingly, we discard bit flips whose byte addresses fall within the upper 8 B half of a 16 B-aligned boundary.

For the remaining bit flips, we next need to examine their bit positions within the lower 8 B half of the PD0 entry. The least significant 4 bits in the 8 B serve as entry flags and must not be altered to preserve validity, so bit flips in this range are excluded. The following 32 bits form a pointer to the corresponding PT by storing bits [39:8] of its 40-bit physical address. We partition these 32 bits into three regions according to how a bit flip affects the resulting pointer:

- Region ① contains 13 bits, and bit flips in this region keep the resulting pointer valid but within the same 2 MB page frame as the original PT. As a result, such flips do not yield an exploitable redirection and should not be considered.
- Region ② is the one we focus on. A flip in this region can redirect the pointer to a different 2 MB page frame outside the original one while still keeping it within the valid GPU physical address space. The width of this region depends on the GPU's physical memory size. For example, on the RTX 3060 with 12 GB of GPU memory, it spans 13 bits, whereas on the RTX A6000 with 48 GB of GPU memory, it spans 15 bits.
- Region ③ covers the rest of the high-order bits of this 32-bit field. A $0 \rightarrow 1$ flip in this region can push the resulting address beyond the GPU's physical address space, rendering the PD0 entry invalid. A $1 \rightarrow 0$ flip, however, decreases the address and may still yield a valid pointer; nonetheless, we exclude this region to simplify candidate selection.

The remaining 28 bits of the 8 B are unused, so any bit flips in them have no effect and are discarded.

With both accessibility and validity criteria in place, we treat the retained GPU memory locations as candidate steering destinations and try them during massaging in ascending address order, starting from the lowest one.

6.2. Memory Massaging Procedure

Our GPU memory massaging procedure consists of five steps, as illustrated in Figure 6. Note that prior to massaging, the GPU memory allocated for templating has been released. Even though the assumption in [29] that allocations always return the same starting physical address does not hold across machines, we observe that on the same machine, the physical page frames backing a `cudaMalloc()` allocation follow a highly stable order across repeated allocations. Accordingly, after page anchoring and releasing the whole allocation, we can precisely determine the range of page frames covered by a new allocation, which allows us to pinpoint the specific page frame of interest during massaging. Below, we describe each step of the massaging procedure in detail.

(a) Isolate the steering destination's page frame. Given a candidate steering destination, we first construct a memory layout that isolates the 2 MB page frame containing it. Using `cudaMalloc()`, we sequentially allocate four contiguous chunks: chunk $\mathcal{A}$ spans all memory preceding the page frame of interest, chunk $\mathcal{B}$ is that page frame itself, chunk $\mathcal{C}$ covers the bulk of the remaining GPU memory after it, and chunk $\mathcal{D}$ reserves the last few tens of 2 MB page frames. As shown in Figure 6, this layout ensures that chunks $\mathcal{A}$ and $\mathcal{C}$ surround chunk $\mathcal{B}$. Moreover, the above-mentioned accessibility filtering guarantees that the aggressor row(s) required to trigger the bit flip at the steering destination reside within chunks $\mathcal{A}$ and/or $\mathcal{C}$.

(b) Deplete the default PD/PT placement region. We find that the NVIDIA GPU driver uses a low-memory region for placing page table structures (PDs and PTs) by default. This region lies far outside the range that our controlled allocations can reach, and thus no aggressor rows can be positioned adjacent to it for hammering. To make PD0 entries reachable by our attack, we need to exhaust this default region to force the driver to place page table structures into higher physical memory regions accessible to us.

This default region occupies roughly the lowest 100 MB of GPU physical memory, although its exact extent depends on where the driver begins allocating page frames (i.e., the same environment-dependent variability as discussed in Section 4). Draining this pool is very challenging, as its capacity can easily accommodate page table structures for the entire GPU memory when default 2 MB pages are used.

Fortunately, we can overcome this challenge by leveraging an interesting property of UVM: it allows 64 KB pages to be instantiated sparsely across a large virtual address space, causing the driver to create many separate yet mostly hollow page table structures that can efficiently deplete the default region. Specifically, we first use `cudaMallocManaged()` to reserve a very large virtual address space (e.g., 16 TB). Under UVM, this reservation alone does not consume GPU physical memory; GPU page frames are allocated only when the corresponding virtual pages are explicitly touched by the GPU. As illustrated in Figure 1, bits [37:29] of the virtual address form the index into PD1, indicating that two 64 KB pages whose virtual addresses differ by 2^{29} (i.e., 512 MB) are

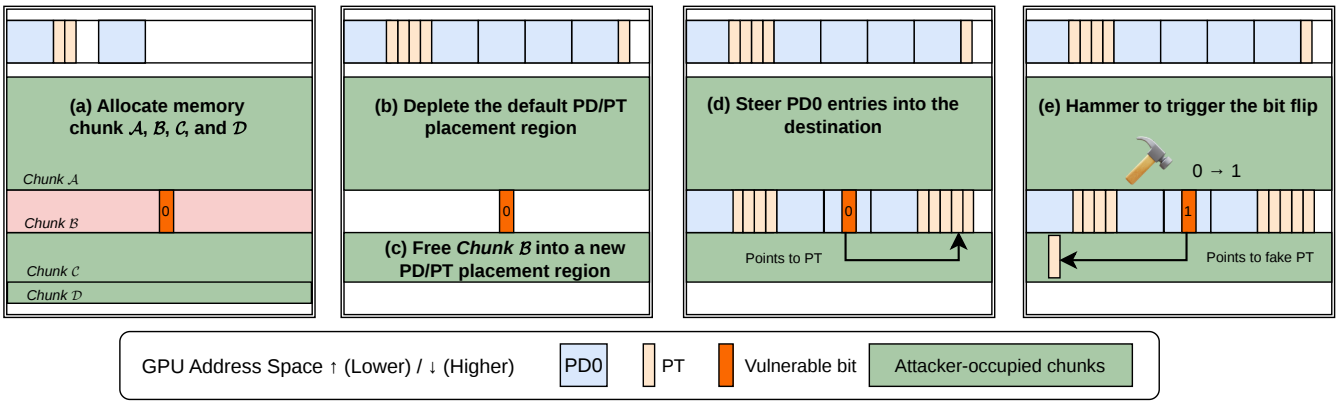


Figure 6: Main steps of `GeForge` memory massaging procedure.

mapped through different PD1 entries and therefore through different PD0s and PTs. As a result, by sequentially accessing pages in the reserved virtual address space with a stride of 2^{29}, we coerce the driver to instantiate, for each access, a new 4 KB PD0, a new 256 B PT, and a 64 KB data page.

Another advantage of this sparse touching pattern is that, as long as any free 2 MB page frames remain, the driver tends to place each such 64 KB page in a separate 2 MB frame, presumably in anticipation that later accesses may populate the other 64 KB pages in the same frame, allowing them to be coalesced into a single 2 MB page. Since step (a) has already occupied most of the GPU memory, newly allocated 2 MB frames will soon have to come from the low-memory region. Consequently, only tens of sparse accesses will be needed to deplete the free 2 MB page frames in that region.

However, exhausting all of the free 2 MB page frames in the low-memory region is still insufficient, because the 2 MB frame currently serving as the active page table structure pool may not yet be full. We thus continue accessing pages at the 2^{29} stride, causing newly created PD0s and PTs to be placed into that active PD/PT pool until it is filled. Note that continuing such accesses remains feasible even after free 2 MB page frames are no longer available, as the driver falls back to inserting new 64 KB pages into partially filled frames. Certainly, the number of continued accesses for filling the active pool is upper-bounded by $2\,\text{MB}/(4\,\text{KB} + 256\,\text{B}) \approx 482$.

Even though a few hundred strided accesses in the UVM-reserved virtual address space suffice to deplete the default PD/PT placement region, the success of our massaging process actually hinges on knowing the exact number, which is itself environment-dependent. We describe how to derive this number at runtime at the end of this section. Fortunately, it only needs to be determined once at the beginning and then can be reused throughout the entire procedure.

(c) Turn chunk $\mathcal{B}$ into a new PD/PT placement region. Once the free 2 MB frames in the default PD/PT placement region are certainly exhausted (the exact count of accesses is not needed here; 100 accesses suffice), we first free chunk $\mathcal{D}$ so that it absorbs subsequent data page allocations. Then, right when the active PD/PT pool reaches its exact capacity, we immediately free chunk $\mathcal{B}$, which contains the steering destination. When the driver next needs to place page table structures, e.g., in response to new accesses to previously untouched UVM pages, it must find a new free 2 MB page frame to serve as the PD/PT placement region. We find that, under this carefully constructed layout, the driver selects the just-freed chunk $\mathcal{B}$ for that role.

(d) Steer PD0 entries into the destination. With chunk $\mathcal{B}$ now serving as the PD/PT placement region, we continue leveraging UVM to populate it with new PD0s and PTs. The driver partitions this 2 MB region into 512 subpages of 4 KB each. The Rowhammer-vulnerable location (i.e., the steering destination) lies in one specific 4 KB subpage, and our first goal is to make a PD0 land in that exact subpage.

Notice that PD0s and PTs consume subpages at different rates: each new PD0 claims a 4 KB subpage, whereas each new PT is appended to the current subpage for PTs until it is full (i.e., each 4 KB subpage holds 16 PTs). Therefore, if we just access UVM pages at the 2^{29} stride without any further control, there is about a 94.1% probability that a PD0 can land in the subpage of interest. To ensure that the landing can always succeed, however, we exploit the fact that accesses to UVM pages with a narrower stride of 2^{21} (i.e., 2 MB) instantiate PTs without creating new PD0s. When the natural PD0 allocation sequence would otherwise skip the subpage of interest, we insert auxiliary PTs to advance PT placement to the next subpage. This shifts the PD0/PT allocation alignment by one subpage, allowing the next PD0 to claim the target subpage.

After a PD0 is positioned to land in the 4 KB subpage of interest, we switch to the 2^{21} stride. At this stride, each subsequent access populates an entry in the already-placed PD0 with a pointer to a newly instantiated PT.

(e) Hammer to trigger the bit flip. As the final step, we execute the designated hammering pattern on the aggressor row(s) in chunks $\mathcal{A}$ and/or $\mathcal{C}$ to corrupt the target PD0 entry, which currently stores the physical address of a legitimate PT. Upon a successful flip, the corrupted pointer is highly likely to land within our controlled memory region. This high probability stems from two factors: first, we have filtered bit flip candidates to ensure that the resulting address resolves to

a valid physical memory range; second, because the allocation in step (a) occupies the vast majority of GPU memory, the redirected pointer will almost certainly fall within chunk $\mathcal{A}$ or $\mathcal{C}$, both of which are under our control.

Because the direction of a Rowhammer-induced bit flip is physically deterministic (i.e., a vulnerable DRAM cell consistently flips either $1 \to 0$ or $0 \to 1$), there is a chance that the flip direction does not match the address modification we need (e.g., the original bit value is already '1' while the flip is a $0 \to 1$). In such a case, we repeat the memory massaging process with a new candidate steering destination.

How to determine the exact number needed in step (b). As mentioned above, we need the exact number N of strided UVM accesses in step (b), but this number is environment-dependent. To determine it at runtime, we again leverage the page anchoring technique from Section 4.

Specifically, prior to the actual massaging run, we carry out step (a) to allocate those four chunks, and we arrange for chunk $\mathcal{B}$ to begin at a known page frame whose first cache line's eviction set has been derived offline. Although the exact number N is unknown, it is empirically above 300 on all platforms we have tested. We thus start the search at this lower bound, setting N to 300. Following step (c), we first always perform 100 strided accesses and free chunk $\mathcal{D}$. Next, we perform the remaining $N - 100$ strided accesses and free chunk $\mathcal{B}$. After that, we access 32 additional UVM pages and test whether the eviction set of chunk $\mathcal{B}$ evicts the first cache line of any of these 32 pages. (We choose 32 conservatively; a smaller number suffices in practice.)

If a successful eviction occurs for one of the 32 pages, the page was certainly allocated in $\mathcal{B}$, meaning that the active PD/PT pool was not yet full when chunk $\mathcal{B}$ was freed and that the driver used $\mathcal{B}$ for data page placement. We then repeat the process with an incremented N. Otherwise, none of the 32 pages resides in $\mathcal{B}$. Since 32 is chosen as a conservatively large number, the only plausible explanation is that the default PD/PT region has just been exhausted after N accesses, and $\mathcal{B}$ has been claimed as the new PD/PT placement region. We therefore obtain the exact number.

7. End-to-End Exploitation

In this section, we present two classes of end-to-end `GeForge` attacks. The first encompasses GPU-local exploits that compromise GPU-resident data structures and contexts, while the second extends the attack to the host side, culminating in privilege escalation to a root shell. The presented attacks can be carried out on the platforms listed in Table 1, using RTX 3060 and RTX A6000 GPUs.

7.1. GPU-Local Attacks

Following the memory massaging procedure, we redirect a legitimate PT pointer to a location within our controlled memory region and forge a PT there. By setting the PTEs in the forged PT to point to arbitrary target GPU physical page frames, we can have access to GPU memory locations of our choice. Note that as the original PT maps 64 KB page frames instantiated by UVM, the target physical addresses must be aligned to 64 KB boundaries.

Armed with this arbitrary read/write primitive over GPU memory, we gain unrestricted access to previously inaccessible regions of GPU physical memory, enabling us to inspect or corrupt any data belonging to other GPU contexts. As a concrete demonstration, we consider attacking a GPU application that implements an image-processing pipeline, transforming an input RGB image into a single-channel sketch-style output. Specifically, the host loads an image in RGB format, allocates dedicated GPU buffers for the RGB input and grayscale output, and launches a CUDA kernel to perform the transformation. The kernel computes local gradients, edge magnitudes, and a tone-mapped base to synthesize realistic, pencil-like strokes. Upon completion, the result is transferred back to the host. For example, Figure 7a shows an input image, and Figure 7c displays the expected, legitimate output of the GPU program.

(a) Original (b) Recovered (c) Expected (d) Forged

Figure 7: GPU-local attack against an image processing application.

First, we use the arbitrary read/write primitive to dump a 40 KB memory region by reading from the GPU buffer that stores the original RGB image. We then reconstruct the image from the raw bytes. As shown in Figure 7b, the image portion recovered from the dumped 40 KB of data matches the top region of the original input. Moreover, we can leverage the primitive to manipulate the same GPU buffer prior to the launch of the CUDA kernel, overwriting its contents with falsified data. For instance, we modify the top portion of Figure 7a, and the output produced by the CUDA kernel is correspondingly altered, as shown in Figure 7d.

This demonstration confirms that `GeForge` can enable arbitrary read and write access across GPU contexts. Beyond this specific scenario, an attacker can leverage the same primitive to launch a wide range of high-impact attacks, such as stealing proprietary ML models or tampering with model parameters to manipulate inference outcomes.

7.2. Host-Side Attacks

While the arbitrary read/write primitive over GPU memory is already powerful on its own, the threat can be pushed one step further by reaching into host memory. In particular, as mentioned in Section 2.2, PTEs of NVIDIA GPUs support a *system aperture* feature that allows the GPU to access host physical memory, which `GeForge` exploits to launch host-side attacks.

The host-side attacks begin in the same way as the GPU-local attacks; that is, the attacker massages GPU memory

and exploits Rowhammer to corrupt a PD0 entry so that it is redirected to a forged malicious PT. Instead of using the forged PT to breach isolation between GPU contexts, the attacker populates its PTEs with the system aperture bit set. (This bit is the third least significant bit in each entry.) With this specific bit set, each PTE maps to a physical address in host memory rather than in the GPU's local memory. As a result, by pointing these PTEs to arbitrary host physical page frames, the attacker gains full read and write access to host memory. Note that, although these PTEs now map to host memory, each entry still preserves its original page size semantics, namely, it references a 64 KB page. Consequently, the host physical addresses in the forged PTEs must also be aligned to 64 KB boundaries.

We emphasize that the success of exploiting forged PTEs to access host memory depends on the system's IOMMU being either disabled or improperly configured. As noted in Section 3.1, this condition is not uncommon in practice. Indeed, on both of our test platforms (see Table 1), the IOMMU is disabled by default.

Privilege escalation. Forged PTEs with the system aperture bit set give us a primitive for accessing host memory. To demonstrate how this primitive can be leveraged for host-side attacks, we implement an end-to-end privilege escalation attack that elevates an unprivileged user to root on the host system. Specifically, we exploit the primitive to tamper with the host physical page frames backing the code segment of the C standard library (`libc.so.6`), allowing the injected code to propagate into subsequently launched SUID programs and enabling privilege escalation.

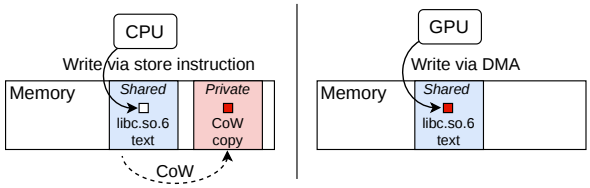


Figure 8: Comparison of writes to COW-protected pages via the CPU and GPU DMA.

Under Linux, shared libraries are typically mapped into processes as private file-backed mappings. Their underlying physical pages are shared across processes and are duplicated only when a process's write triggers copy-on-write (CoW). Our primitive, however, modifies host page frames via GPU DMA rather than through CPU-side memory accesses. Thus, the write of our primitive bypasses the CPU's MMU, does not trigger a page fault, and never invokes CoW, leaving the targeted library's page frames modified in place. In this way, the injected code can propagate to any processes that map the same library. Figure 8 illustrates the difference between CPU-side and GPU-side writes to shared library pages.

Building on this property, we use the primitive to overwrite a target function in `libc.so.6` with `amd64` machine code generated from compiled shellcode. A program that invokes the modified function can then execute the injected payload. If the target program is SUID and owned by root, the resulting execution can yield a root shell.

TABLE 1: Platform specifications.

	Platform A	Platform B
CPU	AMD Ryzen 5 5500	Intel Core i3-10100
Motherboard	ASUS PRIME B450M-A II	Asrock H570 Steel Legend
Host Mem. Size	32 GB	16 GB
GPU	NVIDIA RTX 3060	NVIDIA RTX A6000
GPU Maker	ASUS	DELL
GPU Mem. Size	12 GB	48 GB
GDDR6 Vendor	Samsung	Samsung
SKU	TUF-RTX3060-O12G-GAMING	900-5G133-0100-001
CUDA	12.0.140	12.3.52
NVIDIA Driver	570.133.07	
Linux Kernel	5.15.0-91-generic	

Our shellcode, generated using pwntools [13], is shown in Listing 2 (Appendix D). As modern shells drop elevated effective privileges to match the real user ID (ruid), directly invoking `execve("/bin/sh")` would only yield an unprivileged shell. To bypass this protection, the shellcode first invokes `setreuid(0, 0)` to align both the ruid and euid with root privileges, and then executes `execve("/bin/sh")`.

To identify a suitable target function and program, we first enumerate potentially exploitable root-owned SUID programs, as listed in Appendix D. We then use `ltrace` to analyze the library functions invoked by these programs and identify suitable injection points. To minimize side effects on the host system, we select an infrequently used function, `closelog`, as the target function. This function is invoked by the SUID program `/usr/bin/newgrp`, which we use as the target program. When `newgrp` executes, it spawns a new shell and calls `closelog` upon shell termination. Once `closelog` is overwritten with the shellcode, terminating the `newgrp` shell triggers the injected payload, ultimately yielding a root shell.

This demonstration confirms that, in the absence of effective IOMMU protections, `GeForge` can reliably escalate privileges by modifying shared library code across processes. With a root shell, an attacker can further carry out a wide range of malicious activities, such as injecting persistent backdoors and exfiltrating sensitive information.

8. Evaluation

In this section, we further evaluate the bit flips triggered by the non-uniform hammering patterns and attack reproducibility of `GeForge`. Table 1 summarizes the hardware configurations and software environments for our evaluation platforms, Platform A and Platform B. Specifically, Platform A is equipped with an NVIDIA RTX 3060 GPU, while Platform B uses an NVIDIA RTX A6000 GPU.

8.1. Bit Flips and Hammering Patterns

Our fuzzer identifies 1,171 unique bit flips across 72 banks on the RTX 3060, with the most vulnerable bank exhibiting up to 50 bit flips. Among them, the vanilla pattern triggers 149 flips, while the non-uniform pattern uncovers

an additional 1,059 flips; 37 flips are discovered by both patterns. A detailed breakdown of bit flips per bank is provided in Table 4 (Appendix E). Furthermore, we identify 202 bit flips on the RTX A6000 using the non-uniform pattern. These flips span 94 banks, with up to 11 flips observed in the most vulnerable banks, as shown in Table 5 (Appendix E). The RTX A6000 bit flips represent a substantial improvement over prior work [29], which reported only 8 bit flips across 4 banks. Additionally, after applying the two constraints described in Section 6.1, we retained 44 and 10 candidate memory massaging bit flips for the RTX 3060 and RTX A6000, respectively.

8.1.1. Bit Flip Directions. For the 1,171 observed flips on RTX 3060, 1,032 are $0 \rightarrow 1$ and 139 are $1 \rightarrow 0$. Similarly, for the 202 observed flips on RTX A6000, 179 are $0 \rightarrow 1$ and 23 are $1 \rightarrow 0$. This skew toward $0 \rightarrow 1$ is consistent with prior work [29], which likewise reports that most observed flips are $0 \rightarrow 1$ (6 of 8 in their study).

To explain this behavior, we refer to Kim *et al.* [23], who reported directional asymmetry in Rowhammer-induced flips and attributed it to the physical orientation of DRAM cells. Depending on the manufacturer's implementation, a logical 1 may correspond to a charged capacitor (a *true cell*) or a discharged one (an *anti-cell*). Because a Rowhammer disturbance leaks the stored charge, true cells are prone to $1 \rightarrow 0$ flips, whereas anti-cells tend to flip $0 \rightarrow 1$. In our GDDR evaluation, the predominance of $0 \rightarrow 1$ flips suggests that a logical 0 is represented by the charged state, meaning the victim cells predominantly function as *anti-cells*.

8.1.2. Distances between Aggressor and Victim Rows. In the evaluation, we observed that most observed bit flips are caused by either *double-aggressor* patterns (aggressor rows sandwiching the victim row) or *single-aggressor* patterns (one aggressor row adjacent to the victim row). On the RTX 3060, 205 of 1,171 flips are double-aggressor and 857 flips are single-aggressor; on the RTX A6000, 61 of 202 flips are double-aggressor and 132 flips are single-aggressor, as shown in Table 2. We observe more bit flips under single-aggressor patterns than double-aggressor patterns, because we configure the aggressor distance as ($2 \leq d \leq 8$), as described in Section 5.2. Under this configuration, double-aggressor patterns are used only when ($d = 2$), whereas single-aggressor patterns are used for the remaining values (i.e., 6 out of 7 cases).

However, the remaining bit flips (109 on the RTX 3060 and 9 on the RTX A6000) do not fall into either of the aforementioned categories. We classify these as *remote-aggressor* flips, since the aggressor and victim rows are separated by a distance of 2 or 3, as Table 2 shows.

Previous work, Half-Double [24] and BLASTER [27], also show that remote-aggressor patterns can induce bit flips, as activating remote rows triggers refreshes in adjacent rows (`dist=1`), which ultimately cause the flips. However, in their work, the direct activations of adjacent rows are required to trigger the flips even at low frequency. In contrast, the bit flips we observe on GDDR6 in the RTX 3060 and RTX A6000 occur solely from non-adjacent aggressor rows, i.e., rows with `dist` ≥ 2.

TABLE 2: Distribution of bit flips by aggressor-victim row distance on the RTX 3060 and RTX A6000.

Aggressor-victim Distance	RTX 3060		RTX A6000	
	Flips	Pct.	Flips	Pct.
double-aggressor ($dist = 1$)	205	17.51%	61	30.20%
single-aggressor ($dist = 1$)	857	73.19%	132	65.35%
remote-aggressor ($dist = 2$)	14	1.20%	1	0.50%
remote-aggressor ($dist = 3$)	95	8.11%	8	3.96%
remote-aggressor ($dist \geq 4$)	0	0%	0	0%
Total	1,171	100%	202	100%

8.1.3. Non-uniform Hammering Patterns. Based on the results above, we show that the non-uniform pattern outperforms the vanilla pattern in both efficiency and bit flip coverage.

For efficiency, we ran the fuzzer from Algorithm 1 (Appendix E) for 11 hours with each pattern on the same 64 banks of the RTX 3060 using identical parameters. The non-uniform pattern identified 68 unique bit flips, a roughly 51.1% increase (23 additional flips) over the 45 flips found using the vanilla pattern.

In terms of bit flip coverage, the non-uniform pattern induced 202 bit flips across 94 banks on the RTX A6000. This significantly outperforms prior work utilizing the vanilla pattern, which reported only 8 flips across 4 banks [29]. To further compare bit flip coverage, we randomly selected 10 flips induced by the non-uniform patterns on the RTX 3060. Each pattern spanned two to three refresh intervals with an average amplitude of 7.5. Specifically, three were triggered by double-aggressor patterns, five by single-aggressor patterns, and two by remote-aggressor patterns. We attempted to reproduce these flips using the vanilla pattern under identical aggressor rows and parameters. Ultimately, only a single flip (originally categorized as a remote-aggressor instance) was successfully triggered. These results further confirm that non-uniform hammering patterns expose bit flips beyond what the vanilla pattern can achieve.

8.2. Attack Reproducibility and Generality

As previously discussed, we successfully identified bit flips and executed our attack primitives and exploits on the RTX 3060 in Platform A. To evaluate the portability of these attacks across different host environments, we transferred the identical RTX 3060 GPU to Platform B.

8.2.1. Bit Flips Reproducibility. We randomly selected 10 bit flips to reproduce, and our results show that all of them originally observed on Platform A can be reliably triggered on Platform B. This demonstrates reproducibility across different host architectures (Intel and AMD) using the same

GPU. Moreover, this implicitly indicates that the row mapping remains stable; otherwise, such reproducibility would not be possible.

8.2.2. Attack Primitives Reproducibility. After reproducing the exact memory-massaging bit flip candidate from Platform A, we successfully executed the GPU/host attack primitives and corresponding exploits on Platform B. This demonstrates the cross-host reproducibility of the `GeForge` attack primitives.

When migrating the attack primitives from Platform A to Platform B, we observe that the first physical address allocated via `cudaMalloc()` shifts from `0xA200000` to `0xA400000`. To reproduce the attack primitives, we did a minor adjustment to the memory massaging procedure: shrinking Chunk $\mathcal{A}$ by 2 MB ensures that Chunk $\mathcal{B}$ still maps to the same physical page containing the target bit flip. In contrast, prior work [29] requires re-identifying the entire memory mapping in this case, which is significantly more time-consuming.

8.2.3. Attack Methodology Generality. To demonstrate the platform-agnostic nature of our attack chain, we reproduce an end-to-end exploit chain on Platform B with the RTX A6000, in addition to the RTX 3060 described in Section 7. Leveraging the 202 unique bit flips identified during memory templating, we select 10 exploitable candidates based on the constraints in Section 6.1. Using one of these candidates, we successfully replicate the memory massaging procedure, triggering a targeted bit flip that redirects a PT pointer to a memory region under our control. Finally, we reconstruct the GPU-local attack primitive and confirm unauthorized read/write access to the RTX A6000's physical memory space.

9. Discussion

In this section, we first discuss three mitigation mechanisms against `GeForge` and then discuss the current limitations of our work and directions for future research.

9.1. Mitigation

Error Correcting Code (ECC). ECC is designed to correct bit errors, thereby mitigating an attacker's ability to reliably weaponize Rowhammer-induced bit flips. However, not all GPUs support ECC. For example, consumer-grade models such as the RTX 3060 lack it entirely. Furthermore, ECC does not cover all fault patterns; an adversary can bypass its protection by inducing multiple bit flips within the same ECC codeword, as studied in prior work on main memory [7]. Consequently, ECC should be viewed as a hardening measure rather than a comprehensive defense against GPU Rowhammer.

Input-Output Memory Management Unit (IOMMU). When the IOMMU is enabled, it translates DMA addresses issued by the GPU and checks the permissions of accesses, effectively limiting the GPU's DMA to host-side page frames that the OS or driver has explicitly mapped for the device. As a result, forged system aperture PTEs alone cannot grant access to arbitrary host physical memory. Note that, although effective against host-side attacks, IOMMU protection does not prevent the GPU-local attack primitives of `GeForge`. Therefore, cross-GPU-context attacks remain a viable threat.

Refresh Management (RFM). Compared to TRR, RFM provides a more robust mitigation against Rowhammer attacks. While TRR operates mainly as an opaque, in-DRAM mechanism independent of the memory controller [12], RFM requires strict hardware collaboration. Under RFM, the memory controller tracks the rolling accumulated activations per bank. Once a specific bank reaches its predefined activation threshold, the memory controller suspends standard read/write operations to that bank and issues an RFM command. This provides the DRAM with a dedicated time window (t_{RFM}) to internally identify the aggressor row and refresh the adjacent victim rows.

While RFM is defined in both DDR5 and GDDR6 standards to theoretically prevent Rowhammer attacks, its practical deployment remains highly inconsistent. As reported in [33], Intel and AMD memory controllers fail to issue the necessary RFM commands for DDR5. A similar issue exists in the GPU domain; despite GDDR6 specifications supporting RFM [18, 34], the successful bit flips we observe on the RTX 3060 and RTX A6000 indicate that current GPU memory controllers either configure RFM insufficiently or remain ineffective against targeted access patterns.

9.2. Limitations and Future Work

Other GPU models. We evaluated five additional NVIDIA GPUs beyond the RTX 3060 and RTX A6000 for susceptibility to Rowhammer-induced bit flips: one RTX 3080, one RTX 4060, two RTX 4060 Ti, and one RTX 5050, as listed in Table 3 (Appendix C). Except for the RTX 3080, which uses GDDR6X, all other cards are equipped with GDDR6. For the RTX 4060 Ti, we tested two GPU cards with different memory vendors: one with Samsung GDDR6 and one with SK Hynix GDDR6.

Despite these efforts, we did not observe bit flips on these models. This absence may stem from differences in TRR mitigation policies in those GDDR chips (e.g., counter-based or sampling-based aggressor detection) or other vendor-specific mitigations. A broader exploration across additional GeForce models remains open for future work.

IOMMU evasion. Enabling the IOMMU can neutralize the exploitation of the system aperture mapping, ensuring that even if a GPU page table is taken over, the attacker can no longer directly access arbitrary host memory addresses. However, Thunderclap [32] demonstrates that malicious peripherals can exploit driver-pinned buffers, shared mappings, and other OS-authorized memory regions to circumvent the IOMMU isolation. In future work, we plan to systematically probe GPU DMA mappings to investigate whether analogous IOMMU vulnerabilities exist.

High-Bandwidth Memory (HBM). In this work, GeForge targets GDDR6 memory, which is standard in consumer-grade and workstation-class GPUs. In contrast, server-grade GPUs (e.g., NVIDIA A100 and H100) utilize HBM. HBM employs built-in defenses such as ECC and transparent row-remapping [8]. Although these mechanisms mitigate susceptibility, recent studies demonstrate that HBM remains vulnerable to Rowhammer [37]. This suggests that HBM-based GPUs may also be exploitable, an investigation we leave to future work.

10. Related Work

In early Rowhammer attacks, the canonical hammering patterns were single-sided [23], double-sided [46] (which requires two aggressor rows sandwiching a victim row), and one-location [14] (which relies on a close-page policy to repeatedly activate the same row). These patterns were often sufficient to induce bit flips in DDR3 and LPDDR3 [15, 23, 26, 46, 49, 58, 60]. However, starting with DDR4, memory chips incorporate in-DRAM TRR to detect frequently activated rows and proactively refresh their neighbors, rendering these hammering patterns largely ineffective.

To bypass TRR, TRRespass proposes the first effective hammering pattern that activates more than two rows in the same bank, i.e., many-sided hammering, and demonstrates bit flips on several DDR4 modules [12]. SMASH later shows that synchronizing many-sided hammering with DRAM refresh commands plays an important role in circumventing TRR [9]. SledgeHammer further amplifies the effectiveness of many-sided hammering by performing it in parallel across multiple banks [22]. All of these patterns are uniform in the sense that they activate each aggressor row the same number of times in sequence. In contrast, Blacksmith is the first to make hammering patterns non-uniform by varying activation order, regularity, and intensity, which significantly outperforms uniform patterns [17]. Half-Double, on the other hand, hammers far aggressor rows and leverages TRR-induced refreshes of nearby rows to indirectly help trigger bit flips in the victim row [24].

In recent work, Phoenix reverse-engineers long-horizon TRR behavior in DDR5, spanning hundreds to thousands of t_{REFI} intervals, and accordingly crafts refresh-timed, multi-interval hammering patterns with self-correcting refresh synchronization, achieving bit flips on tested DDR5 chips [33]. This work underscores that refresh-aware, long-horizon, and synchronized hammering is essential for defeating TRR in the most modern memory chips.

RowPress is a DRAM read-disturb phenomenon similar to Rowhammer, as it flips bits in victim rows [20, 31]. However, unlike Rowhammer, which relies on repeatedly opening and closing aggressor rows, RowPress induces bit flips by keeping rows open for an extended period.

Besides CPU-initiated memory accesses for hammering, Throwhammer [47] and Nethammer [30] launch Rowhammer attacks through network devices; Thunderhammer does so via PCIe and Thunderbolt devices [10]; and JackHammer launches attacks from FPGA boards [51]. Nevertheless, their targets remain the system's main memory. By contrast, GPUHammer is the first to target the GDDR6 memory of an NVIDIA GPU with a Rowhammer attack [29].

With respect to Rowhammer defenses, prior studies can be broadly divided into software-only mechanisms [1, 4, 25, 50, 61] and hardware-based techniques [28, 36, 38, 40, 41, 44, 45, 53, 59]. In commodity DRAM chips, however, vendors typically use well-established mitigations such as TRR and ECC, although prior work has called their effectiveness into question [7, 17, 21, 24, 33].

Although Rowhammer can enable many attacks, one of the most powerful is privilege escalation [3, 5, 6, 9, 11, 14, 15, 24, 46, 49, 52]. To achieve this, the attacker needs to place certain security-critical data structures, such as page tables, into vulnerable memory locations, a process known as memory massaging. Various massaging techniques have been developed by exploiting built-in features of software or hardware [3, 5, 6, 9, 11, 14, 19, 26, 43, 46, 47, 49, 60]. For example, Seaborn *et al.* leverage the mmap interface to spray page-table pages in the memory [46]. To our knowledge, GeForge is the first work that demonstrates how to massage GPU page tables and exploit Rowhammer-induced bit flips in GPU memory to achieve privilege escalation.

11. Conclusion

We presented GeForge, which exploits GPU Rowhammer to corrupt GPU page tables and seize control of GPU address translation, thereby enabling a range of attacks. To make GeForge practical under default system settings, we introduced a page anchoring technique to localize target GPU page frames within a runtime allocation, developed an efficient non-uniform hammering strategy that triggers many more bit flips than prior work, and devised a memory massaging technique to steer GPU page table structures into vulnerable memory locations. Building on these techniques, we demonstrated that GeForge can break isolation between GPU contexts. Furthermore, we showed that, when IOMMU enforcement is absent or ineffective, corrupted GPU page tables can be leveraged to access host memory and ultimately achieve user-to-root privilege escalation. Overall, our results broaden the security implications of GPU Rowhammer and underscore the need for stronger architectural and system-level defenses for modern GPU platforms.

Acknowledgment

This work was supported in part by the National Science Foundation under grants CNS-2443671, OAC-2530649, and CNS-2145744. The authors thank the anonymous reviewers and shepherd for their comments and suggestions that helped us improve the quality of the paper.

Ethics Considerations

We reported our findings to the NVIDIA Product Security Incident Response Team (PSIRT), and NVIDIA PSIRT

acknowledged receipt but did not respond further. Moreover, we ensured that all experiments were conducted without involving any third-party systems or non-consenting users.

LLM Usage Considerations

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

References

[1] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[2] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *Annual Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2016, pp. 602–624.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 987–1004.

[4] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi, "CATT: Software-only Mitigation against Rowhammer Attacks," in *USENIX Security Symposium*, 2017.

[5] W. Chen, Z. Zhang, X. Zhang, Q. Shen, Y. Yarom, D. Genkin, C. Yan, and Z. Wang, "HyperHammer: Breaking free from kvm-enforced isolation," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025, pp. 545–559.

[6] Y. Cheng, Z. Zhang, S. Nepal, and Z. Wang, "CATTmew: Defeating Software-only Physical Kernel Isolation," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[7] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory against Rowhammer Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 55–71.

[8] S. Cui, A. Patke, H. Nguyen, A. Ranjan, Z. Chen, P. Cao, B. Bode, G. Bauer, C. Di Martino, S. Jha *et al.*, "Characterizing GPU Resilience and Impact on AI/HPC Systems," *Preprint arXiv:2503.11901*, 2025.

[9] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript," in *USENIX Security Symposium*, 2021.

[10] R. Dumitru, J. Wan, D. Genkin, R. Kennell, Y. Yarom *et al.*, "Thunderhammer: Rowhammer Bitflips via PCIe and Thunderbolt (USB-C)," *Preprint arXiv:2509.11440*, 2025.

[11] 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 (S&P)*, 2018.

[12] P. Frigo, E. Vannacc, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 747–762.

[13] Gallopsled, "Pwntools: CTF Framework and Exploit Development Library," https://github.com/Gallopsled/pwntools, accessed: 2025-11-02.

[14] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 245–261.

[15] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-induced Fault Attack in JavaScript," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016, pp. 300–321.

[16] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor Via Rowhammer Attack," in *Workshop on System Software for Trusted Execution (SysTEX)*, 2017, pp. 1–6.

[17] P. Jattke, V. Van Der Veen, P. Frigo, S. Gunter, and K. Razavi, "Blacksmith: Scalable Rowhammering in the Frequency Domain," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 716–734.

[18] JEDEC, "Graphics double data rate 6 (GDDR6) SGRAM standard," *JEDEC Solid State Technology Association*, 2023.

[19] S. Ji, Y. Ko, S. Oh, and J. Kim, "Pinpoint Rowhammer: Suppressing Unwanted Bit Flips on Rowhammer Attacks," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2019, pp. 549–560.

[20] J. Juffinger, S. R. Neela, M. Heckel, L. Schwarz, F. Adamsky, and D. Gruss, "Presshammer: Rowhammer and Rowpress Without Physical Address Information," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer, 2024, pp. 460–479.

[21] 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 *USENIX Security Symposium*, 2025.

[22] I. Kang, W. Wang, J. Kim, S. van Schaik, Y. Tobah, D. Genkin, A. Kwong, and Y. Yarom, "SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism," in *USENIX Security Symposium*, 2024, pp. 1597–1614.

[23] 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 *International Symposium on Computer Architecture (ISCA)*, 2014, pp. 361–372.

[24] 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 *USENIX Security Symposium*, 2022, pp. 3807–3824.

[25] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[27] Z. Lang, P. Jattke, M. Marazzi, and K. Razavi, "Blaster: Characterizing the Blast Radius of Rowhammer," in *Workshop on DRAM Security (DRAMSec)*, 2023.

[28] E. Lee, Y. Kim, S. Khan, and J. W. Lee, "TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[29] C. S. Lin, J. Qu, and G. Saileshwar, "GPUHammer: Rowhammer Attacks on GPU Memories are Practical," in *USENIX Security Symposium*, 2025.

[30] M. Lipp, M. Schwarz, L. Raab, L. Lamster, M. T. Aga, C. Maurice, and D. Gruss, "Nethammer: Inducing Rowhammer Faults through Network Requests," in *IEEE European Symposium on Security and Privacy Workshops ((EuroS&PW))*, 2020, pp. 710–719.

[31] 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 *International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–18.

[32] A. Markettos, C. Rothwell, B. Gutstein, A. Pearce, P. Neumann, S. Moore, and R. Watson, "Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2019.

[33] D. Meyer, P. Jattke, M. Marazzi, S. Qazi, D. Moghimi, and K. Razavi, "Phoenix: Rowhammer Attacks on DDR5 with Self-Correcting Synchronization," in *IEEE Symposium on Security and Privacy (S&P)*, 2026.

[34] 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 *USENIX Security Symposium*, 2025.

[35] I.-G. News. The RTX 3060 was the Most Used Graphics Card by Steam Gamers this October 2025. https://news.instant-gaming.com/en/articles/15844-the-rtx-3060-was-the-most-used-graphics-card-by-steam-gamers-this-october-2025. Accessed: 2025-11-13.

[36] A. Olgun, O. Mutlu, and colleagues, "ABACuS: All-Bank Activation Counters for Scalable and Low-Overhead RowHammer Mitigation," in *USENIX Security Symposium*, 2024.

[37] A. Olgun, M. Osseiran, A. G. Yağlıkçı, Y. C. Tuğrul, H. Luo, S. Rhyner, B. Salami, J. G. Luna, and O. Mutlu, "An Experimental Analysis of Rowhammer in HBM2 DRAM Chips," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, 2023, pp. 151–156.

[38] Y. Park, W. Kwon, J. H. Ahn, E. Lee, J. W. Lee, and T. J. Ham, "Graphene: Strong yet Lightweight Row Hammer Protection," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[39] C. Peglow and T. Eisenbarth, "Security Analysis of Hybrid Intel CPU/FPGA Platforms Using IOMMUs Against I/O Attacks," *Master's thesis of University of Lübeck*, 2020.

[40] M. Qureshi and S. Qazi, "MOAT: Securely Mitigating Rowhammer with Per-Row Activation Counters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[41] 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 *International Symposium on Computer Architecture (ISCA)*, 2022.

[42] A. S. Rakin, Z. He, and D. Fan, "Bit-flip Attack: Crushing Neural Network with Progressive Bit Search," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1211–1220.

[43] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *USENIX Security Symposium*, 2016, pp. 1–18.

[44] A. Saxena and M. Qureshi, "START: Scalable Tracking for Any Rowhammer Threshold," in *High-Performance Computer Architecture (HPCA)*, 2024.

[45] A. Saxena, G. Saileshwar, P. J. Nair, and M. Qureshi, "AQUA: Scalable Rowhammer Mitigation by Quarantining Aggressor Rows at Runtime," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[46] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," in *Black Hat USA*, 2015.

[47] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX Annual Technical Conference (ATC)*, 2018.

[48] M. Technology, "Micron Datasheet 16Gb: 2 Channels x16/x8 GDDR6 SGRAM (MT61K512M32)," https://www.mouser.com/datasheet/2/671/Micron_08232024_gddr6_sgram_16gb_brief_1578719-3484493.pdf, accessed: 2025-11-02.

[49] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 1675–1689.

[50] 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 *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.
[51] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient Rowhammer on heterogeneous FPGA-CPU platforms," *Preprint arXiv:1912.11523*, 2019.
[52] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *USENIX Security Symposium*, 2016, pp. 19–35.
[53] A. G. Yağlıkçı, 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 *High-Performance Computer Architecture (HPCA)*, 2021, pp. 345–358.
[54] 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 Symposium*, 2020, pp. 1463–1480.
[55] S. Zeitouni, D. Gens, and A.-R. Sadeghi, "It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs," in *Annual ACM/IEEE Design Automation Conference (DAC)*, 2018, pp. 65:1–65:6.
[56] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2023, pp. 960–974.
[57] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, "Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive," in *USENIX Security Symposium*, 2024, pp. 2101–2118.
[58] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, and G. Karsai, "Triggering Rowhammer Hardware Faults on ARM: A Revisit," in *Workshop on Attacks and Solutions in Hardware Security (ASHES)*, 2018, pp. 24–33.
[59] Z. Zhang, Z. Zhan, D. Balasubramanian, B. Li, P. Volgyesi, and X. Koutsoukos, "Leveraging EM Side-Channel Information to Detect Rowhammer Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.
[60] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PThammer: Cross-user-kernel-boundary Rowhammer through Implicit Accesses," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 28–41.
[61] Z. Zhang, Y. Cheng, M. Wang, W. He, W. Wang, S. Nepal, Y. Gao, K. Li, Z. Wang, and C. Wu, "SoftTRR: Protect Page Tables against Rowhammer Attacks using Software-only Target Row Refresh," in *USENIX Annual Technical Conference (ATC)*, 2022, pp. 399–414.

Appendix A. Address Translation

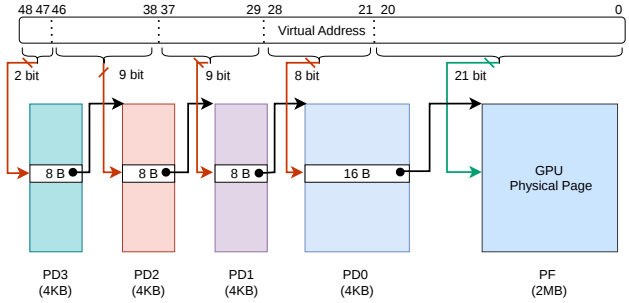


Figure 9: Address translation for 2 MB pages in Ampere GPUs.

Appendix B. Row Stripes and Aggressor Row Accessibility

To determine the row stripe size for both GPU models, we need to obtain their total number of banks, N_B, as defined in Section 6.1. The NVIDIA RTX 3060 GPU has 12 GB GDDR6 memory, which features six 2 GB GDDR6 Ball Grid Array (BGA) chips connected via a 192-bit bus with 32-bit per chip. Each chip implements two independent 16-bit channels, with each channel comprising 16 banks. Consequently, the GPU contains a total of 192 memory banks ($N_B = 6 \times 2 \times 16 = 192$). Similarly, the NVIDIA RTX A6000 is equipped with 48 GB of GDDR6 memory distributed across twenty-four 2 GB BGA chips. Physically, each chip contains two independent channels, with 16 banks per channel, yielding a total of 768 memory banks ($N_B = 24 \times 2 \times 16 = 768$).

As shown in Figure 4, we assume that each row stripe (e.g., 768 KB for the RTX 3060 and 1536 KB for the RTX A6000) occupies a contiguous region of physical GPU memory. This assumption is consistent with prior work [29], which implicitly assumes a linear mapping between physical addresses and successive row IDs. Under this model, as the physical address increases, the corresponding row ID increases monotonically.

To further illustrate the accessibility of aggressor rows, we first use the NVIDIA RTX 3060 as an example, whose row stripe layout is shown in Figure 4a. Suppose the fourth row stripe (green) contains the vulnerable bit. During the memory massaging step, the attacker must release all 2 MB pages, making it impossible to construct aggressor patterns involving the fifth row stripe (white), thereby preventing effective double-sided Rowhammer. However, the attacker can still control 512 KB of the third row stripe (red), so single-sided Rowhammer remains possible. For the NVIDIA RTX A6000 (row stripes shown in Figure 4b), we assume that the vulnerable bit resides in the third row stripe (red), specifically within the 1024 KB portion of the second 2 MB page that will be released. In this case, the preceding row stripe (green) has 512 KB located in the first 2 MB page, while the following row stripe (purple) is fully contained

within the third 2 MB page. As a result, the attacker can still construct double-sided aggressor patterns using these two row stripes.

Appendix C. GPU Models

TABLE 3: GPU models and memory specifications.

GPU Model	DRAM Size	DRAM Type	DRAM Vendor
RTX 3060	12 GB	GDDR6	Samsung
RTX 3080	10 GB	GDDR6X	Micron
RTX 4060	8 GB	GDDR6	SK Hynix
RTX 4060 Ti	8 GB	GDDR6	Samsung/SK Hynix
RTX 5050	8 GB	GDDR6	Samsung
RTX A6000	48 GB	GDDR6	Samsung

Appendix D. Exploitation

Listing 1 lists all root-owned SUID programs on Platform A. Listing 2 is our shellcode.

Listing 1: Setuid root programs

```
/usr/bin/gpasswd; /usr/bin/su; /usr/bin/chfn;
/usr/bin/mount; /usr/bin/chsh; /usr/bin/umount;
/usr/bin/newgrp; /usr/bin/fusermount3; /usr/bin/passwd;
/usr/bin/sudo; /usr/bin/pkexec;
/usr/libexec/polkit-agent-helper-1; /usr/sbin/pppd;
/usr/lib/dbus-1.0/dbus-daemon-launch-helper;
/usr/lib/snapd/snap-confine;
/usr/lib/openssh/ssh-keysign; /usr/lib/xorg/Xorg.wrap
```

Listing 2: Shellcode

```
xor edi, edi /* 0 */

/* call setreuid('rdi', 'rdi') */
push SYS_setreuid /* 0x71 */
pop rax
mov rsi, rdi
syscall
/* execve(path='/bin///sh', argv=['sh'], envp=0) */
/* push b'/bin///sh\x00' */
push 0x68
mov rax, 0x732f2f2f6e69622f
push rax
mov rdi, rsp
/* push argument array ['sh\x00'] */
/* push b'sh\x00' */
push 0x1010101 ^ 0x6873
xor dword ptr [rsp], 0x1010101
xor esi, esi /* 0 */
push rsi /* null terminate */
push 8
pop rsi
add rsi, rsp
push rsi /* 'sh\x00' */
mov rsi, rsp
xor edx, edx /* 0 */
/* call execve() */
push SYS_execve /* 0x3b */
pop rax
syscall
```

Appendix E. GeForge Fuzzer and Bit Flips

Algorithm 1 illustrates the workflow of our Fuzzer. Table 4 summarizes the bit flips observed across 72 banks on the RTX 3060 GPU, while Table 5 covers the bit flips for 94 banks on the RTX A6000 GPU. On both tables, each entry represents the total number of bit flips found in a single bank.

Algorithm 1 Workflow of `GeForge` Fuzzer.

Require: `rounds`

for each `i` = 1 to `rounds` **do**
  Reserve a large chunk of GPU memory with `cudaMalloc()`.
  Select the target bank randomly.
  Select a hammering pattern type, either vanilla or non-uniform. For the non-uniform pattern, choose X, A, and ϕ.
  Select n aggressor rows with W warps and T threads.
  Select data pattern pairs from `0xFF`/`0x00` and `0x55`/`0xAA`.
  Generate `d`, the distance between two aggressor rows, and `group_step`, the distance between the starting row IDs of two groups of aggressor rows.
  Load the mappings for the target bank and parse `max_rows` as the number of rows in the bank.
  Compute sweep offsets `sweep_offsets` $\leftarrow 0 :$ `group_step` $:$ `max_rows`.
  for each sweep start `s` in `sweep_offsets` **do**
    Select the aggressor row set $\mathtt{S} \leftarrow \{\, s+i\cdot \mathtt{d} \mid i = 0, \ldots, n-1 \,\}$
    Select victims $\mathtt{V} \leftarrow \{\min(\mathtt{S}) - 2, \ldots, \max(\mathtt{S}) + 2\} \setminus \mathtt{S}$
    for `r` = 1 to 5 **do**
      Initialize the rows in `S` and `V` with chosen data patterns, e.g., `0xFF` for rows in `S` and `0x00` for rows in `V`.
      Assemble the hammering pattern with aggressor rows in `S` as Section 5.1 describes, then execute the hammering pattern.
      Evict the L2 caches.
      Read all chunks from rows in `V` and compare with the original data pattern.
      Append deviations (virtual address, expected pattern, observed pattern, row ID) to log.
    end for
  end for
  Release the allocated GPU memory and introduce a brief delay to allow the GPU to process the deallocation.
end for

TABLE 4: Bit flips per bank on RTX 3060 (1,171 bit flips on 72 banks).

4	2	5	5	21	40	12	12	19	3	9	7	19	8	21	16	9	4
11	10	18	47	23	12	16	19	5	8	20	5	25	13	7	20	12	9
44	19	23	33	21	35	15	26	19	16	46	26	27	42	19	31	15	27
2	10	14	8	7	8	17	24	16	16	22	10	12	20	1	1	1	2

TABLE 5: Bit flips per bank on RTX A6000 (202 bit flips on 94 banks).

7	3	1	3	1	1	2	3	4	1	3	2	1	2	1	2	1	3	1	1
1	7	11	2	1	1	2	11	6	2	3	4	2	3	2	4	1	4	2	1
1	1	1	4	1	1	3	2	1	1	2	2	5	1	1	1	2	3	1	2
2	3	2	5	1	2	1	1	1	3	1	3	2	1	1	2	1	2	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	1						

Appendix F.
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.

F.1. Summary

This paper presents GeForge, an end-to-end Rowhammer-based attack targeting modern NVIDIA GPUs equipped with GDDR6 memory. The paper demonstrates that carefully engineered hammering patterns can induce bit flips in GPU memory structures, including GPU page tables, and that these bit flips can be exploited to obtain arbitrary read and write access to GPU physical memory. Leveraging a specific page table entry (PTE) feature—the aperture field—the attack further enables GPU access to host CPU memory, thereby breaking isolation between GPU and host.

Building on these primitives, the paper demonstrates a sequence of downstream exploits, including cross-context GPU memory access, manipulation of victim GPU workloads, and ultimately a host privilege escalation that results in root access on a Linux system. The paper claims three primary contributions: (i) inducing higher bit-flip rates on GPUs than prior GPU Rowhammer work, (ii) introducing new exploitation techniques that leverage GPU page table semantics to bridge GPU and CPU memory, and (iii) demonstrating practical, end-to-end exploitation scenarios that show the security implications of GPU Rowhammer attacks.

F.2. Scientific Contributions

- 1. Independent Confirmation of Important Results with Limited Prior Research.
- 4. Addresses a Long-Known Issue.
- 5. Identifies an Impactful Vulnerability.
- 6. Provides a Valuable Step Forward in an Established Field.

F.3. Reasons for Acceptance

1) The paper provides independent confirmation of important results with limited prior research, building on GPUHammer from Lin et al.
2) The paper addresses a long-known Issue, demonstrating that Rowhammer is a serious threat to GPU systems.
3) The paper identifies an impactful vulnerability, especially in demonstrating a pivot from GPU memory to the attached host.
4) The paper provides a valuable step forward in an established field, especially in showing that so many more GPU DDR6 bits are flippable.

F.4. Noteworthy Concerns

1) The attack, while impressive, is performed with ECC and IOMMU disabled. Enabling either may significantly complicate the most powerful attack variant that escalates privilege to the host.
2) The attack was performed in a desktop-class system, which is often single-tenant. In a single-tenant system the privilege escalation capability is not as consequential.