ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Not so Refreshing: Attacking GPUs using RFM Rowhammer Mitigation

Ravan Nazaraliyev*[1], Yicheng Zhang[1], Sankha Baran Dutta[2], Andres Marquez[3], Kevin Barker[3], Nael Abu-Ghazaleh†[1]

[1]*University of California, Riverside*
[2]*Brookhaven National Laboratory*
[3]*Pacific Northwest National Laboratory*

Abstract

Graphics Processing Units (GPUs) have become a critical part of computing systems at all scales. In this paper, we demonstrate new side channel attacks targeting the Graphics DDR (GDDR) memory chips. While several studies have demonstrated attacks on CPU memory chips, revealing potential security vulnerabilities, these attacks do not easily transfer to GPU memories, due to differences in the microarchitecture and operational characteristics of GDDR memory and GPU memory controllers, as well as the distinct computational model of GPUs. We reverse-engineer the mapping of physical addresses to GDDR physical bank addresses and show that existing row buffer timing attacks on these systems are ineffective due to row buffer management policies. Instead, our attacks target the Refresh Management (RFM) feature engineered into modern memories to mitigate Rowhammer vulnerabilities. We identify RFM-based timing leakage where repeated accesses to the same bank trigger refresh events, leading to measurable differences in access times. We exploit this leakage to first construct covert channel attacks on a shared GPU, achieving a bandwidth of over 50 KBps per bank with a low error rate of 0.03%. We demonstrate two end-to-end side-channel attacks on discrete GPUs with GDDR6: application fingerprinting and 3D object rendering fingerprinting within *Blender*, achieving F1 scores of up to 95% and 98%, respectively. Additionally, we implement three side-channel attacks on GPU-based SoCs using LPDDR5 memory: application fingerprinting, web fingerprinting, and video fingerprinting, achieving high F1 scores. Finally, we present a Denial of Service (DoS) attack, where the attacker leverages the RFM blocking to slow down applications by over 4.8× on average.

1 Introduction

GPUs are becoming increasingly important across a broad range of data-intensive applications, ranging from graphics rendering [35], machine learning training and inference [47, 73] to scientific computation [53]. Due to their highly parallel computational models, GPU memories are architected to meet these high-bandwidth demands, in ways different from conventional CPU-based systems. For instance, GDDR chips are used in commodity GPUs (*e.g.*, GeForce RTX 40 series [58]), while High Bandwidth Memory (HBM) chips are used in data center GPUs (*e.g.*, A100 [57]).

For the past decade, Dynamic Random Access Memory (DRAM) has proven vulnerable to *Rowhammer* attacks [16, 23, 24, 37, 49, 50, 69, 75, 77]. These attacks exploit a vulnerability in DRAM where repeatedly accessing (or "hammering") a specific row can cause electrical interference that causes bit flips in adjacent rows. This vulnerability can be leveraged to gain root privileges, or otherwise compromise victim applications [69]. Subsequently, groups of researchers adapted rowhammer attacks to different architectures [16, 23, 24], and systems including networks [75] and mobile phones [77]. Since these attacks can be launched remotely from software, they form a substantial threat to modern computing systems.

The first widely adopted mitigation of Rowhammer attacks is the idea of Target Row Refresh (TRR) [36, 39], where the system (the memory controller or the DRAM chip) tracks the most heavily accessed rows and triggers a refresh of their neighbors when the accesses reach a pre-defined threshold. Different companies had their own proprietary implementations of TRR, and new attacks demonstrated gaps in TRR protection often based on the limited number of rows that can be tracked [16, 23]. More recently, the JEDEC community, a non-profit DRAM industry trade organization, introduced a standard mitigation mechanism known as Refresh Management (RFM) [17, 30, 31, 34, 38, 44]. RFM shares the overall philosophy of TRR in identifying heavily accessed memory rows and refreshing them, but does some of the tracking and actions at the granularity of banks or sub-banks to support scalability. Moreover, under some circumstances, RFM blocks the bank until the next refresh operation. Modern DDR manufacturers widely implement this RFM solution to pro-

*rnaza005@ucr.edu
†naelag@ucr.edu

tect GDDR chips [26]; it is also used in a number of other DRAM standards including DDR5, HBM3, LPDDR4/4X, and LPDDR5 [27–30]. The GDDR6/6X [26] standard is the first generation of GDDR to use RFM. We discuss RFM and reverse engineer some of its operational details in § 3.

In this paper, we show that the use of RFM to mitigate Rowhammer unintentionally introduces new security problems including covert channels, side channels, and denial of service vulnerabilities. RFM uses activation counters to track the number of activations sent to a DRAM bank. These counters trigger different actions such as refreshing target rows, or even temporarily blocking the bank until the next refresh operation. The root of the vulnerabilities is that the RFM counters are associated with the physical bank and therefore *shared* among applications with memory mapped to each bank; allocation of physical pages naturally allows pages of different applications to share banks. Refresh actions lead to detectable timing variations, leading to timing leakage. Moreover, denial of service vulnerabilities can arise when a malicious application drives bank counters to initiate blocking, severely limiting access to those banks.

The RFM-based leakage is a different source of leakage than prior attacks on DDR such as DRAMA [65]. We show that DRAMA style attacks are challenging to carry out in GDDR due to the short row buffer timeout (limiting row-buffer contention leakage), and physical memory allocation policies (preventing row-buffer co-residency among processes). Specifically, on GDDR memories, the memory controller has a small time-out interval before closing a row after access to the row is complete. The high global memory access latency and small time-out interval for open row buffer severely limit leakage through the row buffer, interfering with the DRAMA row buffer contention attacks. Moreover, the allocation of physical pages does not allow the sharing of a single DRAM row between applications, therefore eliminating the potential for DRAMA [65] row buffer co-residency based attacks. Our attacks exploit a different phenomenon; rather than detecting the timing difference between row buffer hits and misses, we detect the timing difference when bank blocking occurs, which leaks the number of bank-level activations.

To successfully launch our attacks, we overcome several challenges originating from the GPU execution model and memory hierarchy, as well as from the nature of GDDR. First, we reverse engineer DRAM address translation from physical addresses to bank addresses and develop approaches so that virtual-to-physical address translation is not needed. We also reverse engineer RFM operation and interactions between the memory controller and DRAM. Using this information, we construct attacks on multiple NVIDIA GeForce RTX GPUs and an NVIDIA SoC (Jetson AGX Orin). On the discrete GPUs, we demonstrate a high-quality covert channel bandwidth of over 50 KBps per bank (bandwidth can be further increased by communicating across multiple banks in parallel), with a low error rate. We next demonstrate two side channel attacks on NVIDIA GeForce RTX GPUs with GDDR6: (1) application fingerprinting; and (2) inferring rendered characters in a 3D rendering application, both with over 90% F1 scores. In addition, we implement three side channel attacks on the Jetson AGX Orin which uses LPDDR5 memory: application fingerprinting, web fingerprinting, and video fingerprinting, all with F1 scores exceeding or approaching 90%. Moreover, we present a slowdown attack where a malicious program intentionally causes expensive memory blocking to slow down other applications over 4.8 times on a selected benchmark.

In summary, the contributions of this paper are:

- We reverse-engineer GDDR physical page to DRAM bank addressing on several generations of NVIDIA GPUs. We also propose deterministic physical page allocation approaches that overcome the need to know the virtual to physical mapping when developing RFM conflict sets.
- We reverse engineer RFM operation and identify opportunities for timing leakage. This includes finding conflict sets and identifying timing behavior and interactions between the memory controller and DRAM.
- We demonstrate an RFM leakage-based covert channel of high bandwidth with a low error rate on two generations of NVIDIA GPUs.
- We demonstrate five side-channel attacks on discrete GPUs or SoCs to identify co-located CUDA applications, 3D character rendering, CPU applications, browsed websites, and played videos, achieving F1 scores exceeding or approaching 90%.
- We demonstrate a slow-down attack, where a malicious process can intentionally push RFM to block banks and slow co-located applications (by an average factor of 4.8x).

Although we explore RFM in the context of GDDR6, it is also part of the DDR5, HBM3, LPDDR4/4X, LPDDR5 [27–30] standard and is likely to impact CPU-based and mobile systems as well. It is also interesting that the source of the vulnerability is a *feature designed to improve the security of the system*; it is important to think about security holistically.

2 Background and Threat Model

In this section, we first provide an overview of GDDR memories and RFM protection mechanisms. We also introduce our threat model.

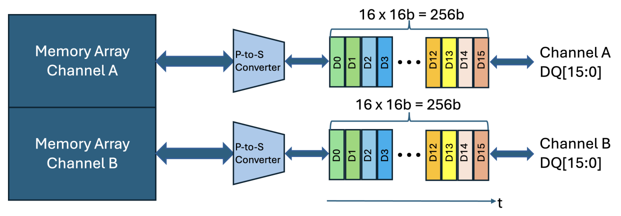


Figure 1: GDDR6 channel organization.

2.1 GDDR

Contemporary DRAM systems are organized into a multi-tiered hierarchy comprising memory channels, Dual In-line Memory Modules (DIMMs), ranks, and banks. On a CPU-based system, ranks are composed of multiple chips with typical numbers of four or eight per rank. Each DRAM chip is subdivided into banks, which are further segmented into rows and columns. Graphics DDR chips are typically mounted directly on the circuit board surrounding the GPU chip, with each GDDR chip interfacing with the GPU chip through a dedicated 32-bit wide channel. A distinctive aspect of this architecture is that each GDDR chip has its own independent memory channel, managed by its own discrete Memory Controller (MC) embedded within the GPU chip.

GDDR6 DRAM chips differ from GDDR5 chips by incorporating what are known as independent channels. As shown in Figure 1, a single memory bank is divided into two independent channels. Each of the two 16-bit wide channels can independently process memory requests, creating a cumulative 32-bit data path. GDDR chip architecture shares similarities with other DDR memories, particularly in terms of bank organization, and the use of row buffers, rows, and columns. The row buffer plays a key role in facilitating data communication with external components. When an *ACTIVATE* command is received, the row buffer activates an entire row, after which specific columns of bits are accessed and read from the buffer. GPU threads can access any address within the allocated global memory space. When a thread requests data from global memory, the access is first checked in the cache hierarchy, including the local L1 cache and the shared L2 cache. If the data is not found in these caches, the request is forwarded to a memory controller, which decodes the requested physical address into bank, channel, row, and column values. A memory coalescing unit combines accesses to the same cache line across different threads of a single warp executing the same instruction to reduce the number of memory accesses.

When a row is active within the row buffer and a different row needs to be accessed, the memory controller issues a *PRECHARGE* command to the GDDR chip. This command closes the currently active row, thereby allowing a new row to be activated. Closing the row and precharging the row buffer for the new access incurs some overhead. Conversely, if a subsequent data request is for the same row that is already active, it results in a row buffer hit. The memory controller's handling of the row buffer is contingent upon its policy. Typically, after activating a row, memory controllers maintain the row buffer open for a timeout interval so that the next requests hitting the same row can be directly served from the row buffer [6, 20]. Modern memory controllers are typically equipped with *adaptive page policies* [20] which dynamically adjust the behavior of the row buffer based on the pattern of memory accesses. Prior DRAM attacks such as *DRAMA* [65], rely on controllers that use open-page or adaptive page policy. The effectiveness of this attack heavily relies on the timeout interval for the open row buffer.

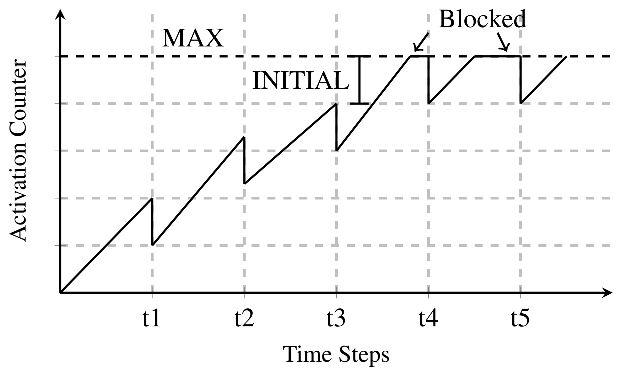


Figure 2: The RFM mechanism.

2.2 Refresh Management

To mitigate Rowhammer attacks on DRAM chips, the idea of refreshing frequently-accessed rows was proposed soon after the attack was introduced [36, 39], culminating in a class of solutions called in-DRAM Target Row Refresh (TRR). Different DRAM vendors had different proprietary implementations of TRR that varied in terms of which and how many rows are monitored, and when to trigger a refresh. Recent attacks have shown that many of the TRR implementations are vulnerable to many-sided Rowhammer attacks [16]. These attacks generate access to many rows. As a result, the TRR logic cannot track all frequently accessed rows, leaving some rows vulnerable to the attack, and resulting in bit flips. Moreover, TRR issues victim row refreshes at during periodic *REFRESH* periods (typically around 7.8 μs), an attacker can send a pattern of row activations during this period with no TRR protection. In a similar vein, Jattke et al. [23] showed that non-uniform access patterns that work around the refresh schedule and include random accesses to fool the TRR logic can induce even more bit flips than uniform many-sided attacks. These patterns trick TRR into refreshing random rows while the attacker activates their target rows.

Refresh Management (RFM) was introduced in new sets of JEDEC standards for LPDDR4/4X, HBM3, and GDDR6 [26–28]. RFM is implemented across both the memory controller

(MC) and DRAM chips, effectively sharing the operational responsibilities [17, 34]. The protection mechanism in RFM leverages ideas similar to TRR, generating refresh operations to suspected victim rows, but with important differences in implementation and granularity to help close some of the attacker opportunities. In addition to using a sampler that tracks heavily accessed rows, RFM incorporates a counter that records the number of row activation requests sent to a memory region, such as a memory bank [26] or sub-bank [34]. RFM provides memory controllers (MCs) with an *RFM* command, offering an external interface to DRAM chips controllable by the MC.

RFM aggressively approaches mitigating Rowhammer attacks [17]. Typically, RFM uses two threshold values for activations for each memory region [26, 34]: an initial or intermediate threshold and a higher maximum threshold. When the activation counter for a memory region reaches the initial threshold, the memory controller (MC) can issue an RFM command, prompting the DRAM chip to initiate refreshes of *victim* rows, similar to TRR, but triggered on demand rather than periodically. If the activation counter reaches the maximum threshold, the corresponding memory region will no longer accept further memory accesses. This blocking can occur due to exceeding the high threshold on the number of activation requests within the memory region. Moreover, *RFM* commands can be delayed for performance reasons by the memory controller even when the activation counter exceeds the initial threshold [26], leading to blocking until the RFM command is received. When refreshes occur, the access counters are reduced; when they fall below the blocking threshold, the memory region becomes accessible again.

An illustration of RFM operation is depicted in Figure 2. At time steps t_1, t_2 and t_3, the (MC) issues *RFM* commands, leading to refreshing the hot-rows, and a reduction in the counter by a designated fixed value [26]. Although the plot illustrates this reduction as instantaneous, the process is time-consuming because RFM allocates an additional time window for the completion of refreshes [7, 26]. Following t_3, the counter increases to its maximum permissible limit (the maximum threshold), at which point the memory region becomes blocked, preventing further activation until the counter is reduced at t_4 after a refresh cycle or an RFM command. If memory accesses continue to the bank, the counter may increase again and result in the bank being blocked, leading to significantly high access latency.

Although the RFM mechanism is standardized, as depicted in Figure 2, proprietary implementations vary in different ways. For instance, an AMD patent [34] describes how designs can differ in how they track sub-banks. Additionally, some implementations may employ a single threshold, where the initial and maximum thresholds are identical, or configure the thresholds differently. Furthermore, the mechanism can be configured to rely on using the standard *REF* commands instead of issuing *RFM* commands, as *REF* is often faster [34].

To summarize, RFM introduces two types of behaviors: (1) every time the initial threshold is reached (when an RFM command triggers refreshes), and (2) when the maximum threshold is reached (leading to bank/sub-bank blocking). While the overall standard is described in detail [26, 30, 31], significant variations are possible [17] both in the operation of the memory controllers (*e.g.*, , different thresholds and optimizations to merge refreshes), and the DRAM vendors (*e.g.*, , distinct implementation of hot row tracking and refreshing in response to RFM commands) that need to be reverse engineered for each GPU system. Nevertheless, the delays observed for the two types of refresh are detectable by applications that access a bank undergoing either an RFM refresh or blocking. These delays are the primary sources of leakage we target in our side channel attacks, while the blocking behavior is the vector enabling the slowdown attacks.

2.3 Threat Model

Our threat model is similar to previous works [41, 52, 80, 83, 84], where a *spy* CUDA application and a *victim* CUDA or graphics application share a GPU. This scenario is feasible in cloud settings, where multiple tenants can share the same physical GPU through virtualization [12, 42, 74, 81]. Additionally, prior research has demonstrated practical techniques for achieving co-location in cloud systems [4, 67, 85]. The attacker does not require superuser privileges. Instead, they perform experiments to reverse engineer the GDDR address mapping functions that hash physical addresses to GDDR bank addresses, as a preliminary step for the attacks.

Our attacks can be generalized to other platforms that use RFM. We expect versions of the attack can be developed for systems that use other memories. We demonstrate attacks on Jetson AGX Orin 64GB (LPDDR5) [32], where an attacker runs on the GPU and spies on CPU applications, without special support or privileges.

Experimental Platform. We perform evaluations on four NVIDIA GPUs with GDDR6 memories from the three major vendors: two GeForce RTX 4060 GPUs with GDDR6 from Samsung and SK Hynix respectively, one GeForce RTX 3080 with GDDR6 from Micron, and one GeForce RTX 3070 Ti with GDDR6 also from Micron.

3 Attack Overview and Challenges

In this work, we present three types of end-to-end attacks: covert and side channels, and a slow-down attack.

Research challenges. Successfully carrying out these attacks requires us to solve four challenges: (1) Since the attacker does not know the virtual to physical address mapping, we need to uncover or control the virtual to physical address mapping; we describe how to get physical addresses for a given virtual address with a root privilege for reverse engineering and a method to control physical memory allocation when there is no root privilege for the attacks; (2) How to efficiently bypass the complex GPU cache hierarchy such that we can generate memory accesses that reach the DRAM chips; (3) Identifying which physical addresses map to the same bank to cause or measure contention: this requires us to understand the mapping between physical memory addresses and the DRAM banks; (4) Understanding the RFM leakage behavior, RFM sub-bank partition to locate addresses and distinguish this leakage from row buffer contention. The side channel attacks and slow-down attacks, share similar requirements in terms of deriving address groups that map to different banks. We discuss these four challenges and how we overcome them in the remainder of this section.

3.1 Virtual-to-Physical Address Translation

In this section, we first show how to obtain a physical address corresponding to a virtual address in a user application with a root privilege. Then using this information, we provide a method to deterministically allocate physical pages when the application does not have root privilege.

Obtaining physical address. We reverse engineer the physical memory allocation in order to enable the attacker to control their physical memory allocation. We instrument the Unified Virtual Memory (UVM) module in the open-source NVIDIA driver to collect virtual to physical mappings to reverse engineer the allocation process [21]. When a process is using Unified Virtual Memory, the `cudaMallocManaged()` API first allocates memory on the CPU side (similar to `malloc`). Once this memory is accessed by the GPU, a page fault occurs and at this time, the allocation in the GPU global memory is performed. CUDA also supports asynchronous prefetch of data from system memory to device memory using `cudaMemPrefetchAsync()`. This call allocates memory matching the prefetch size on the GPU memory, registering a physical space of 2MB to the requested process. We discover that `block_gpu_pte_write_2m` method implemented inside the UVM driver registers the physical page.

We instrument the driver to track the physical memory allocation events and discover that physical memory allocation is sequential. Moreover, we confirm that using `cudaMalloc()` (an alternative to `cudaMalloManaged()` to allocate memory when UVM is not used), the allocation happens with a granularity of 2MB, aligned at a 2MB boundary as well even when the allocation request is smaller than this size. GPUs (as well as CPUs) often use an optimization called coalescing TLBs to expand the reach of the TLB structures [66]. The 2MB consecutive allocation promotes TLB coalescing, allowing these consecutive pages to share a single TLB entry. Allocation is also consecutive for allocation sizes larger than 2MB.

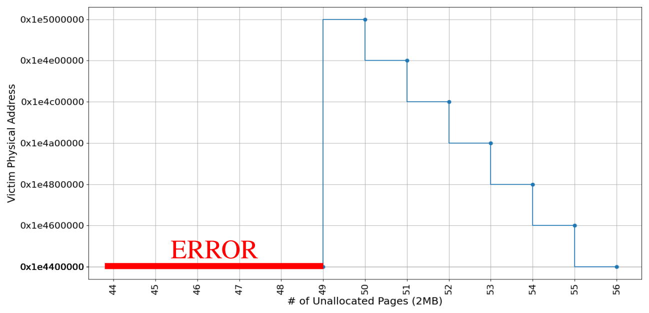


Figure 3: Deterministic physical allocation in RTX 4060.

> ***Observation 1****: Allocations in GPU global memory happen in 2MB pages. The first free 2MB space (2MB-aligned) is assigned to the requesting process, that is, allocations happen in logically sequential order.*

The consecutive allocation of 2MB chunks simplifies the virtual to physical mapping component of the attack in the following way: the groups of addresses that we derive to hash into the same bank from a consecutively allocated region continue to hash together to the same bank even when mapped to a different physical location, or even when used by a different program. The address hashing patterns are preserved given that these pages are consecutive in physical memory. We verified this observation empirically; thus, we can derive groups of addresses for each bank and continue to use them regardless of the virtual to physical mapping; groups of addresses continue to hash to the same bank, even though the bank may change for different virtual to physical mappings.

Forcing deterministic allocation. We already pointed out how to obtain a physical address from the driver. However, without root privilege, we can neither modify the driver nor access the kernel log messages. The attacker processes (trojan and spy) need to know the physical address to select the correct set of addresses for a successful attack (especially, a covert channel).

Since it is hard to derive the virtual-to-physical mapping, we offer a generic method for controlling the physical allocation. Using Observation 1, we realize that we can force our process to allocate a specific part of memory: GPU giving contiguous memory allocations enables us to build a deterministic allocator. We first show this across different processes. As visualized in Figure 3, if the whole memory is occupied by the first process, then the second process gets an **ERROR** on any allocation. Only if the first process does not allocate the last 49 pages, the second application can successfully launch and allocate memory. The reason behind this observation is that 48 pages (each 2MB) corresponding to 96MB is required to launch the process, then the required 2MB page is allocated. As the first process does not allocate

Algorithm 1 Obtaining Last Page (2MB)

```
1: $vector < uint32\_t* > Pages;$
2: $uint32\_t * Last\_2MB\_page;$
3: while True do
4:     $uint32\_t * temp;$
5:     $err = cudaMalloc((void **)\&temp, 2MB);$
6:     if $err \neq cudaSuccess$ then
7:         $break;$
8:     end if
9:     $Pages.push\_back(temp);$
10: end while
11: $N \leftarrow Pages.size();$
12: for $i \leftarrow 0$ to $N-1$ do
13:     $cudaFree(Pages[i]);$
14: end for
15: $Last\_2MB\_page = Pages[N-1];$
```

the last physical pages, the second process obtains the last page (*e.g.*, PA=$0x1e5000000$) first. We use this behavior to propose a method to allocate the last available page. Since the GPU allocates starting from the lower pages, we know for sure that the last pages are available. Algorithm 1 shows how to utilize this observation in a single application. The process continuously allocates all physical pages available on global memory, then releases the prior pages and only keeps the last page. This knowledge is especially crucial in covert channels where, before communication starts, the sender and receiver should agree upon a bank. Note that the physical addresses (*e.g.*, PA=$0x1e5000000$) on the plot are stable, i.e. the last pages do not change and they correspond to the available global memory. With the physical pages not changing over time, the attacks become stable. Moreover, this gives the attacker another opportunity that they do not need to rely on reverse engineering results. An attacker can easily find address pairs for the last page(s) and launch her attacks without knowing the physical address.

3.2 Bypassing GPU Caches

Bypassing L1 cache. A preliminary challenge is to bypass the complex GPU cache hierarchy such that memory accesses reach DRAM, rather than be served through the caches. For global memory (as opposed to private memory), GPUs bypass the use of the L1 cache to avoid coherence issues where shared global data is accessed and modified in L1 caches of different GPU SM cores [14]; even on GPU systems that cache global memory accesses in L1 cache, this can be disabled using a compiler option [1].

Bypassing L2 cache. We also bypass L2 cache accesses using the `discard` instruction for all of our attacks. NVIDIA introduced this new PTX instruction in PTX 7.4 [56] to invalidate data in the L2 cache. The `discard` instruction gets an address at an alignment of 128 bytes, matching the cache line granularity, and invalidates the data in that cache block in the L2 cache. The `discard` instruction is only supported on Ampere and Ada architectures.

In the absence of `discard`, it is still possible to do the reverse engineering (and, in principle, the attacks although we do not present complete end to end attacks). Reverse engineering on earlier GPUs without discard can use an eviction based approach to bypass the caches.We did not invest in deriving eviction sets, but instead periodically flush the cache (i.e., evict all sets) since we are able to do so efficiently. Specifically, to flush the L2 cache, we linearly access a buffer at a 32-byte stride that is equal to or larger than the size of the L2 cache [21]. To reduce the cache eviction time, we parallelize the accesses using parallel threads.Flushing the L2 cache remains a costly operation and we increase the number of memory operations between flushing in the following ways. The NVIDIA GPU caches are sectored [15, 76], with each cache line partitioned into four 32-byte sectors. When a thread accesses any 32 bytes within a cache block, only those 32 bytes get cached in the L2 cache. Consequently, for a given cache block, we can generate 4 different accesses, each for one sector, without discarding or flushing the cache line. Furthermore, since the row buffer operates at the page granularity, we can use many cache blocks within the same page to access the same row without having to flush the already-used addresses. We note that during cache flushing, accesses to the four sectors within the same cache line are coalesced by the GPU into a single memory operation; coalescing does not occur during the attack since we use a single thread to access the different sectors. These two optimizations together, substantially reduce the number of times we need to flush the cache, without harming the cache flush time, improving the bandwidth on GPUs that do not support discard.

3.3 DRAM Bank Address Translation

Having bypassed the caches, the next challenge is to identify how different physical addresses are mapped to DRAM banks to enable the identification of addresses that reach the same bank. Knowledge of address mapping enables the attacker to efficiently generate bank-specific addresses. We use *a timing side channel* that relies on the observation that when two addresses map to the same bank, they cause the row buffer for the first accesses to be closed before the timeout value; when we access the row again, we will experience a higher access time needed to reactivate that row. We then use groups of addresses that map to the same bank to derive the hashing functions that map addresses to banks. Our approach learns from prior works to find physical address to bank mapping in the context of DDR [5, 19, 65], but these works are not directly usable due to important differences in how GDDR is structured.

Identifying addresses mapping to the same bank. We use *a timing side channel* that exploits the difference between row buffer hit and conflict (explained in § 2) to identify addresses

[1]Compiler option "-Xptxas -dlcm=cg" bypasses the L1 cache.

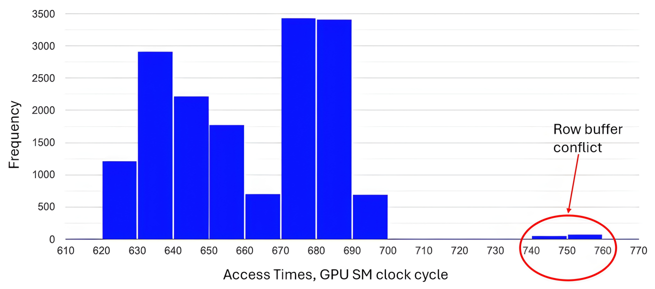


Figure 4: Access times distribution on RTX 4060.

that share the same bank. Specifically, pairs of addresses targeting the same bank will have higher latency when issued in alternating patterns due to row buffer conflicts. The experiment to identify whether two addresses share the same bank alternates accesses repeatedly and then takes the minimum access time to eliminate the noise in the measurement. If this time exceeds a threshold, we identify the addresses as mapping to the same bank.

In our experiments, we allocate global memory on the GPU and select the addresses to group from this allocation. The address-hashing functions, which determine the bank each address resides in, may utilize any of the bits in the physical address. Thus, we ensure that we have addresses across the range of the physical address space. By repeating this experiment for large numbers of addresses, we identify groups of addresses that each map to the same bank.

Deriving address mapping hash functions. The memory controller uses address-hashing functions to map physical addresses to chip/bank/channel/row/column (channel in the case of GDDR6). These addresses are constructed to try to distribute the memory pages to banks in a way that reduces bank contention. We assume the functions use linear XOR operations of selected bits of the address, as is typical with address and index hashing functions [5]. After grouping addresses into banks using the timing side channel, we use an open source tool [65] to reconstruct the address hashing functions; the tool derives functions using different bit combinations that explain the mapping of the addresses recovered from the timing analysis for each group of addresses that shares a common bank.

To find out the cache blocks in the same row, we perform two experiments. In the first experiment, we compare the first block with all other blocks in 2MB space. This gives us the locations of cache blocks that have row buffer conflicts with the first block. However, this experiment would not yield the addresses that are in the same row as the first block. In the second experiment, we take an address that caused row buffer conflict with the first block in the first experiment and compare that address against all other blocks in the 2MB memory space. Based on the address patterns, we observe that each 1KB memory is divided into 2 parts: cache blocks 0-1-4-5 and cache blocks 2-3-6-7 are in the same row respectively. We conjecture that this structure is due to the independent channel configuration of GDDR6. We also tested a GPU with GDDR5 memory where each 1KB segment stayed in the same row since GDDR5 does not possess channels. This channel configuration doubles the number of banks while decreasing the row size of a physical bank. This (see Figure 1) behavior becomes clear when we group the addresses and get twice the number of banks as presented in the GDDR6 datasheets [45]. A histogram of the memory access latency is shown in Figure 4, where we can clearly observe the higher times due to row-buffer conflicts.

Observation 2*: If the 2MB page is divided into 1KB pieces, cache blocks 0, 1, 4, and 5 map to the same row. Cache blocks 2, 3, 6, and 7 map to another row. This sub-page granularity arises due to the use of independent channels in GDDR6 memory modules.*

We use Observation 2 when grouping the addresses into banks. Since a 1KB range is in the same physical bank (but different channels inside the bank), we can ignore the channel function and work on only 1KB pieces. Once we find out the address mapping functions for chips/banks, in the next phase we can identify the channel address mapping function which decides which 4 cache blocks map into which channel.

Reverse engineering results. GPU address translation uses complex XOR functions, that is, the masked bits are not limited to some parts of the physical address, rather they use several different bits to distribute addresses into physical banks. The reversed engineered functions are shown in Figure 9. We do not label the functions as the timing side channel does not provide information related to chips or banks. It only differentiates groups of addresses in the same bank. The channel function can be directly identified once the bank is identified.

3.4 Characterizing RFM Leakage

We conduct experiments to understand the RFM behavior observable from user applications (CUDA Kernels running on the GPU), as well as the granularity of the sharing of the RFM counters. The experiments compare three different cases: (a) when there is no contention; (b) when there is row buffer contention; and (c) when RFM is triggered. This experiment is important to see how a malicious process can trigger RFM blocking, or on the other hand, observe and characterize RFM-related delays in memory accesses. The leakage differs from prior row buffer-based attack (DRAMA) [65], which relies on row buffer contention or sharing. The figure also highlights some of the unique features of RFM leakage where accesses to addresses corresponding to the same RFM region (the region having the same RFM activation counter) increase their shared counters and cause delays through refresh or blocking when access thresholds are reached.

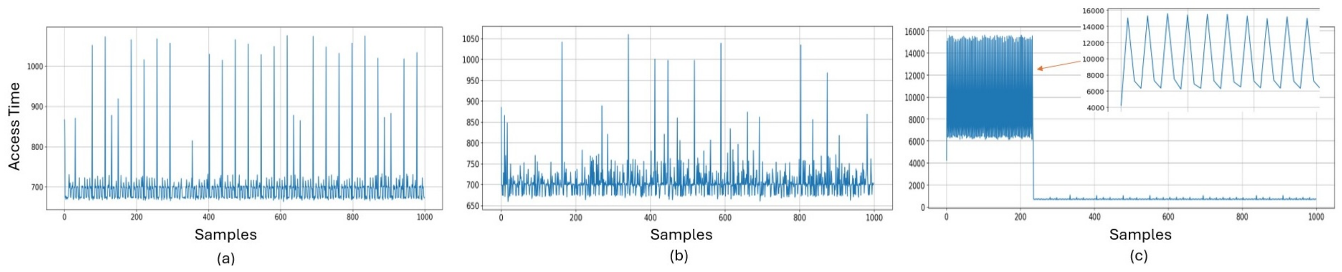


Figure 5: Different scenarios of contention comparisons: (a) refers to the scenario where the observer sees no contention; (b) refers to the case where the observer sees fluctuating row buffer contention; (c) refers to the case where the observer experiences RFM blocking. The x-axis refers to the access time (clock cycle) for kernel *A*. The y-axis shows the measurement sample.

Three cases share the same experiment setup with different address comparisons. The experiment setup consists of two CUDA kernels (*A* and *B*), each running only a single thread, to simplify address mapping. Kernel *B* accesses a given global memory address repetitively, while Kernel *A* observes memory access delays. *A* accesses its own address (primary address) and measures the time for this access to finish, while *B* accesses its address in different places in memory (secondary addresses), chosen to trigger different RFM and row buffer contention behavior. Since we already reverse-engineered the index mapping functions, we are able to select the addresses to help identify the RFM bank division structure.

Figure 5 illustrates the observed delays under three different scenarios where *B* accesses: (a) a different address in a different DRAM bank, leading to no contention with the primary address A; (b) a different row address in the same bank but in a different sub-bank; and (c) a different row address in the same bank and sub-bank.

No contention. Figure 5 (a), shows that when the *B* is activating an address in a different bank from *A*, *A* observes no contention. We ignore the spikes in the plot since they typically correspond to the periodic refreshes. We conclude that *global memory accesses in different RFM sub-banks or different banks do not cause contention to each other.*

Row buffer contention. In Figure 5 (b), we observe *fluctuating* contention in the DRAM access time of *A* while *B* keeps activating a row in the same bank, but different RFM sub-bank. This variability comes from the row buffer conflicts: contention happens when the memory controller does not yet close the row buffer after *B*'s accesses and *A's* access conflicts in the row buffer. The reason for the *fluctuation* is that in some cases, the GPU memory controller has already closed the row buffer, in others, the row buffer is open. This behavior (further explained in § 12) is due to the timeout interval being small with respect to high global memory access times. Therefore, *the attacks based on row buffer contention become ineffective in GPUs.*

RFM blocking. Figure 5 (c) shows high DRAM access times of *A* while *B* repeatedly activates a row in the same RFM sub-bank. We observe similar behavior even when *B* accesses the same row *A* accesses due to RFM blocking. The decreasing times reflect resuming accesses once blocking stops due to decreasing counter values after refresh. *We conclude that the RFM timing side channel is different from row buffer contention in the following ways: RFM leakage introduces a substantial increases in access time; RFM leakage can also occur across the same row.*

Observation 3*: In NVIDIA RFM design, the DRAM banks are divided into sub-banks, each sub-bank having a different set of RFM counters. RFM has a completely different leakage than row buffer timing channel: Row buffer contention is incremental for a single access time while RFM increases an access time several times and RFM leakage can occur across the same row.*

The experiment demonstrates that addresses within the same RFM sub-bank as the primary address induce specific contention, which we refer to as RFM leakage. We can set a threshold and compare different addresses to obtain address pairs that cause RFM contention, hence understanding how RFM divides banks into sub-banks. For example, if we try to group cache blocks in the same bank into RFM groups on RTX 4060, we get 3 distinct ones. This highlights that the bank is divided into 3 sub-banks, each sub-bank having a different RFM activation tracker counter. In RTX 3070 Ti and 3080, we observe a bank is divided into 2 sub-banks. This design of the RFM mechanism is similar to AMD's Refresh Management for DRAM Memory Controller patent [34]. We designed an experiment to measure how long the blocking delays could be observed after blocking is triggered. The experiment uses a thread to access a memory bank until blocking is triggered. The other thread samples DRAM access time as in Figure 5. The experiment delays the start of sampling to see how long the blocking behavior is observable: we find that it is no longer observable after 3 microseconds. This makes it difficult to detect blocking after a kernel switch.

4 Covert Channel Attack

In this section, we exploit the findings in § 3 to build a fast and resilient covert channel. We begin by illustrating the design of the covert channel and the synchronization mechanism, followed by our evaluation results.

4.1 Covert Channel Design

In the § 3.4, we observe that if a specific memory region or sub-bank receives row activations, the counter corresponding to that region increases and the RFM mechanism takes the actions according to the activation counter. The communication channel based on RFM leakage uses the ability of sender and receiver processes to control the number of activations sent to DRAM and measure the access time to specific addresses in DRAM. The receiver senses **delayed** accesses to its address when the sender accesses alternating rows to increase the number of activations in the memory region. Conversely, when the sender is inactive, the receiver experiences a **normal** global memory access time.

Covert channel synchronization. Synchronization between the sender and receiver processes in a covert channel is crucial for controlling bandwidth and minimizing the number of erroneous bits. We utilize 3-phase synchronization in our covert channel. First, the sender signals the receiver with a pre-agreed pattern of data [2], indicating the sender's readiness to start communication. In the second phase, the receiver responds with pre-agreed data to signal its readiness to receive. Once the sender gets this signal, the handshake is complete. The sender starts to transfer the bits to the receiver. To further reduce the error rate, the sender attaches the pre-agreed data pattern at the start of the message, allowing the receiver to easily identify the beginning of the message.

The receiver and sender kernels are located in two host processes. To successfully build the covert channel between the two processes, they should execute concurrently on the GPU. With the help of Multi-Process Service (MPS) [61], multiple CUDA kernels can execute in parallel, sharing GPU resources. Multi-process execution minimizes the overhead from GPU context switching and boosts performance, especially when a single process does not fully exploit the GPU's capacity. Our covert channel is based on timing leakage from RFM in DRAM. We utilize MPS to support parallel execution.

Deterministic physical allocation. Implementing this covert channel poses several challenges. Firstly, the address spaces for the sender and receiver processes are different, and neither party is aware of the other's address space. Additionally, the user can only see the virtual addresses and lacks virtual-to-physical address mappings. To overcome these challenges, we use Algorithm 1 for a deterministic physical allocation. Algorithm 1 continuously allocates 2MB pages on the GPU until running out of available memory. When the covert channel starts, the receiver uses the Algorithm 1 to keep allocating 2MB pages sequentially until the GPU's memory is exhausted. This ensures the receiver process secures the last available 2MB page in the physical address space. After that, the sender process begins its memory allocation, obtaining the 2MB page immediately ahead of the receiver's page. Recall that the GPU allocates memory sequentially (Observation 2 in section 3.1): we use algorithm 1 to allocate the last physical page to the the receiver and then the page before that to the sender. As a result, the processes obtain known physical addresses allowing communication on pre-agreed set of banks. We note that this is not essential since the active banks could alternatively be probed to establish initial synchronization.

[2] The pre-agreed data pattern is long series of continuous highs followed by a small sequence of continuous lows.

Algorithm 2 Receiver for Covert Channel

```
 1: // $Samples_{\text{receiver}}[N]$ is an array of $N$ samples to record the access times
 2: // $T$ is the threshold to differentiate bits '1' and '0'
 3: // $H$ is the number of consecutive highs to form a 1
 4: // $L$ is the number of consecutive lows to form a 0
 5: Synchronization();
 6: for $i \leftarrow 0$ to $N$ do
 7:     $start \leftarrow clock()$;
 8:     $*ReceiverRow$;
 9:     $end \leftarrow clock()$;
10:     $Samples_{\text{receiver}}[i] = end - start$;
11:     $discard(ReceiverRow)$;
12:     // Check for the end of communication pattern
13: end for
14: // Using T, H and L decode samples into message bits
```

Designs for sender and receiver. Algorithm 2 and Algorithm 3 show the design of the receiver and sender, respectively. The receiver, once the synchronization has been completed, starts measuring the access time to its global address. The receiver starts decoding the collected samplings into bit representation once the receiver receives the 'end' character, a data pattern representing the end of the communication. The receiver checks for this pattern every time in the iteration. Once the receiver observes this pattern in the message, it finishes receiving, and then decode the message. We define different duty cycles for bits '1' and '0', namely H and L in Algorithm 2. In the sender process, we observe that activating 4 rows ($*row0, *row1, *row2, *row3$) is sufficient to cause considerable timing difference for the receiver to differentiate between high and low. Both sender and the receiver uses **discard** instruction to bypass L2 cache. The access pattern of the sender corresponds to a many-sided Rowhammer attack.

4.2 Covert Channel Evaluation

We follow the process described in § 3.4 to find address pairs that reside within the same RFM sub-bank. The receiver requires 1 row ($*ReceiverRow$), while the sender needs 4 rows ($*row0, *row1, *row2, *row3$). The level of high is determined by the number of activations (R in Algorithm 3) in the sender code and the duty cycle of low bit is determined by

Algorithm 3 Sender for Covert Channel

```
 1: // $Message_{sender}[N]$ is an array of $N$ bits used to send a message
 2: // $R$ is the number of iterations to send the ACT command
 3: // $K$ is the number of clock cycles to stay idle
 4: Synchronization();
 5: for $i \leftarrow 0$ to $N$ do
 6:     if $D_{sender}[i] == 1$ then
 7:         for $i \leftarrow 0$ to $R$ do
 8:             *row0; *row1; *row2; *row3;
 9:             discard(row0); discard(row1);
10:             discard(row2); discard(row3);
11:         end for
12:     else
13:         start ← clock();
14:         end ← start;
15:         while end - start < K do
16:             end ← clock();
17:         end while
18:     end if
19: end for
```

the idle time (K in Algorithm 3). The error rate is determined using the Levenshtein edit distance [46]. For evaluation, the sender sends a message composed of 10000 bits. The message is made up of a randomly generated sequence of equal numbers of '1' and '0' bits.

Results. We implement the covert channel on GeForce RTX 3080, 3070 Ti, and two 4060 GPUs from different vendors. We evaluate the performance of the covert channels using two metrics: bandwidth and error rate. As shown in Table 1, the covert channel achieves a bandwidth of up to 55 KBps with an error rate of 0.68% on an RTX 4060 GPU equipped with GDDR6 from SK Hynix. On the RTX 3070 Ti, the covert channel's bandwidth and error rate change to 44.1 KBps and 0.98%, respectively.

Table 1: Covert channel: Bandwidth (KBps)/Error rate(%).

	Bandwidth	**Error rate**
RTX 4060 8GB (SK Hynix)	55.0	0.68
RTX 4060 8GB (Samsung)	52.7	0.75
RTX 3080 12GB (Micron)	50.8	0.03
RTX 3070 Ti 8GB (Micron)	44.1	0.98

Evaluation on RTX 3080. We specifically evaluate the covert channel bandwidth and error rate on the GeForce RTX 3080 12GB. The evaluation is done by varying the duty cycle of low bit. As presented in Figure 6, the bandwidth is lowest at clock cycle 4000 and the error rate is also low. This is because the duty cycle is stable. Towards clock cycle 3500 the error rate increases due to decrease in the duty cycle. The duty cycle becomes unstable at 3500 and decreases by 1 stabilizing at 2900. At clock cycle 2900, the channel has the best performance with an error rate of 0.03% and bandwidth of 50.8 KBps. Then as the duty cycle decreases more, the channel destabilizes, and both bandwidth and error rate increase.

Parallelization of covert channel. Since different pairs of addresses cause RFM contention to each other. We utilize

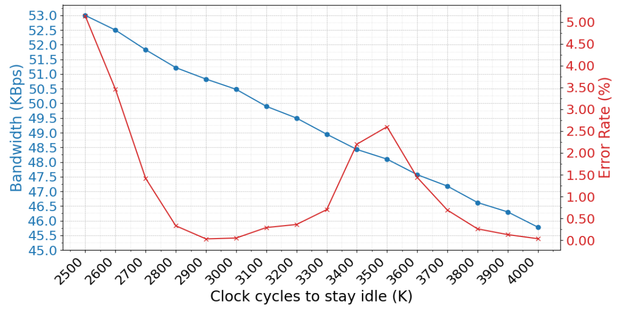


Figure 6: Covert channel evaluation on GeForce RTX 3080.

independent sets of pairs to further increase the bandwidth of the channel. The GPU architecture offers flexible parallelization for CUDA applications. For example, the RTX 4060 GPU has 24 streaming multiprocessors (SMs), with each SM containing multiple warp schedulers. Each warp scheduler can schedule a warp (32-thread) at a time with a specific scheduling policy. CUDA interface allows the developer to run CUDA kernels on different SMs by launching different thread blocks. An example of this is $<<< 2,32 >>>$, which launches 2 different blocks of 32 threads (a warp) on distinct SMs, allowing for parallel execution. Using this method, we implement the receiver and sender with 2 thread blocks each, resulting in a total of 4 thread blocks running concurrently. This approach doubles the bandwidth to **110 KBps** on RTX 4060 with SK Hynix GDDR. However, this increased bandwidth comes with a higher error rate, which rises to 5%.

5 Side Channel Attack

In this section, we utilize the leakage vectors identified in § 3.4 to develop two side-channel attacks: an application fingerprinting attack on CUDA applications and a 3D rendering fingerprinting attack on graphic workloads.

We assume a scenario where a spy application operates in the background, persistently monitoring RFM leakage. The spy uses **discard** instruction to bypass the L2 cache. Meanwhile, a victim user conducts various activities on the shared GPU. Analyzing the traces of GDDR memory footprints allows us to correlate them with the victim's actions and infer secrets about the applications in use. We assume the shared GPU enables Multi-Process Service (MPS) [61].

5.1 Attack 1: Application Fingerprinting

In this attack, we demonstrate that an attacker can infer specific CUDA applications by exploiting RFM leakages. We use 20 CUDA applications from the Rodinia benchmark [8] and NVIDIA CUDA samples [60] as the victim. Rodinia is a benchmark suite designed for heterogeneous computing, targeting both CPUs and GPUs. Table 6 lists 20 CUDA applications across various domains.

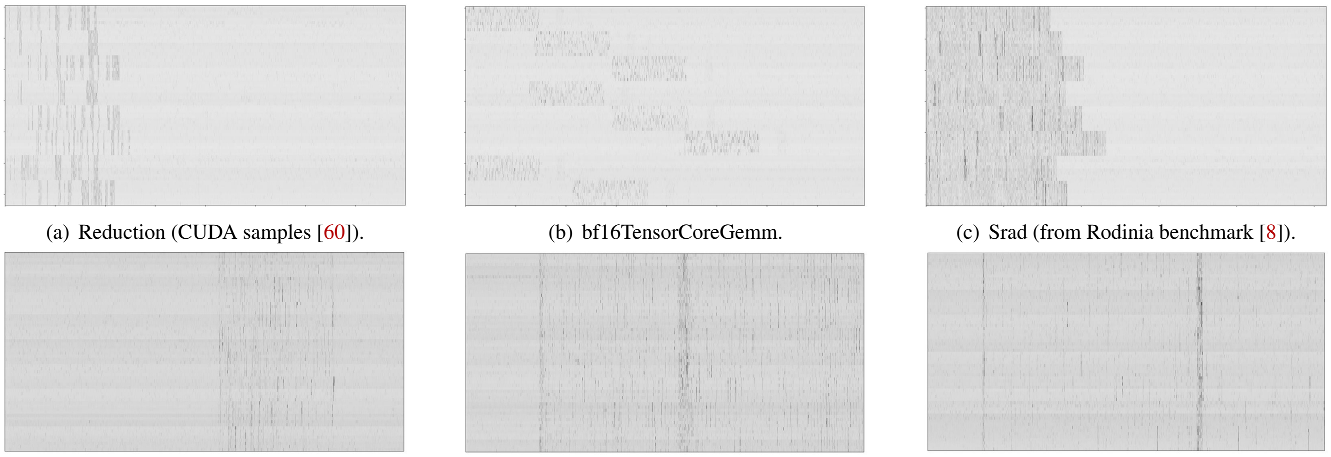

(a) Reduction (CUDA samples [60]). (b) bf16TensorCoreGemm. (c) Srad (from Rodinia benchmark [8]).
(d) Character 1 (Blender Studio [11]). (e) Blender character 2. (f) Blender character 3.

Figure 7: Memorygrams show distinct patterns (y-axis shows time for 64 banks of RTX 4060 over time x-axis).

Experimental setup and data collection. We profile GDDR memory footprint traces of a CUDA-based spy while the victim runs CUDA applications from the selected applications. For each application, we gather 100 trace samples. In each sample, the spy application runs 8 blocks of CUDA kernels, and within each kernel, 8 banks are profiled sequentially, resulting in a total of 64 GDDR banks being profiled. Between each sampling, there is a delay of approximately 100 ns, letting the spy have a raw sampling rate of ~300,000 samples per second. The spy application collects 3 million data points for each bank. However, since this number of data points is too large for the memorygrams to capture distinguishable patterns, we limit the spy to only saving 200,000 data points. We observe that this number is sufficient to reveal the patterns.

Observing memorygram distinguishability. Fig. 7(a)- 7(c) shows the traces of RFM leakages from 64 banks of an RTX 4060 GPU when three different selected applications (Reduction, bf16TensorCoreGemm and srad benchmarks) are executed. The data samples of RFM timing measurements are normalized from 0 to 1. Following established parlance [63, 70], we refer to these traces as memorygrams. By observing the memorygrams of three applications, we can easily distinguish these applications.

Classification and results. Since we have 20 applications, our dataset consists of 2,000 memorygrams. To classify these memorygrams into specific applications, we use the CNN model ResNet-152 [18] with pre-trained weights. Because our memorygrams are grayscale images, we customize the first layer of ResNet-152 to have a single input channel instead of three. Additionally, the last layer is modified to output 20 classes. We resize the memorygrams to 224 by 224 pixels and train the model using PyTorch [64] version 2.1.2, employing a 10-fold cross-validation method [40]. For each fold, we compute three performance metrics: F1 score (F1), Precision (Prec), and Recall (Rec). In our training setup, we utilize the *CrossEntropyLoss* function as our criterion to handle multi-class classification tasks. For optimization, we employ the *Adam* optimizer with a learning rate of 0.0001 and a weight decay of 1e-5. Table 2 presents the average and standard deviation across the 10 folds for 4 experimental machines. For RTX 4060 GPUs, our attacks achieve F1 scores of over 94% and 93% for two GDDR6 manufacturers: Samsung and SK Hynix, respectively. For RTX 30 series, we can correctly fingerprint the victim's application with F1 scores of 88.6% for the RTX 3080 and 95.4% for the RTX 3070 Ti GPU.

5.2 Attack 2: 3D Rendering Fingerprinting

We follow up the application fingerprinting attack with a proof-of-concept demonstration showing that a spy can steal 3D graphic content rendered by the victim by sniffing RFM leakage. First, we introduce the background of 3D rendering on GPUs. Then, we illustrate our attack methods and present evaluation results.

3D rendering on GPU. 3D rendering transforms a 3D scene into a photorealistic 2D image, an intensive process that relies heavily on GPUs for speed, especially with large scenes [22]. This study focuses on Blender [10], an open-source toolkit for rendering images from 3D scenes. Blender's rendering process involves five main steps: scene loading, ray casting, path tracing, shading, and output. First, Blender loads the 3D

Table 2: Application fingerprint performance: F1 (%), Precision (%), and Recall (%).

	F1	**Prec**	**Rec**
	$\mu(\sigma)$	$\mu(\sigma)$	$\mu(\sigma)$
RTX 4060 8GB (Samsung)	94.4 (7.4)	95.3 (6.7)	94.9 (6.4)
RTX 4060 8GB (SK Hynix)	93.7 (4.8)	95.6 (3.3)	94.0 (4.3)
RTX 3080 12GB (Micron)	88.6 (8.5)	90.5 (8.0)	89.0 (8.1)
RTX 3070 Ti 8GB (Micron)	95.4 (4.5)	96.2(4.0)	95.7 (3.9)

Table 3: 3D rendering fingerprint performance: F1 (%), Precision (%), and Recall (%).

	F1	Prec	Rec
	$\mu(\sigma)$	$\mu(\sigma)$	$\mu(\sigma)$
RTX 4060 8GB (Samsung)	96.0 (5.4)	97.1 (4.0)	96.4 (4.9)
RTX 4060 8GB (SK Hynix)	98.3 (2.4)	98.9 (1.6)	98.4 (2.4)
RTX 3080 12GB (Micron)	96.7 (4.2)	97.4 (3.5)	97.0 (3.8)
RTX 3070 Ti 8GB (Micron)	94.3 (5.5)	95.1 (5.0)	94.8 (5.0)

models and all parameter settings into GPU memory. The GPU then handles a vast number of calculations for ray casting, path tracing, and shading. For ray tracing acceleration, Blender uses a Bounding Volume Hierarchy (BVH) supported by NVIDIA GPUs [2]. After the GPU completes all computations, the rendered 2D images are returned to the host CPU.

Observing 3D character distinguishability. We observe that different 3D characters rendering introduces different delays on the GPU's GDDR which the memorygram can capture. Fig. 7(d)- 7(f) show the memorygrams of RFM leakages when three different Blender characters are rendered on an RTX 4060, respectively.

Experimental setup and data collection. Similar to our previous attack, we collect RFM leakage via a CUDA-based spy program while the victim renders 3D graphic workloads in Blender. The spy application collects 1 million data points for each bank. Since the blender takes longer time than the applications, we increase the delay between measurements to ~30 μs downgrading the raw sampling rate to ~30,000 samples per second. For the victim 3D characters, we choose forty 3D characters from the Blender Studio open movies [11]. Each character is rendered with the same parameter settings, including the exact same background scene, camera angle, and rendering resolution. We test on Blender version 4.2.0. For each character, we collect 50 memorygrams, generating a dataset of 2,000 memorygrams in total.

Classification and results. We use a pre-trained ResNet-152 model [18], customizing the first layer to have a single input channel and modifying the output layer to handle 40 classes. During the training stage, we use the *CrossEntropyLoss* function as the criterion and the *Adam* optimizer with a learning rate of 0.0001 and a weight decay of 1e-5. Table 3 presents the performance of 3D rendering fingerprint attacks, including the average and standard deviation across 10 folds. For NVIDIA RTX 4060 GPUs, we achieve F1 scores of over 96% and 98% for GDDR6 memory from Samsung and SK Hynix, respectively. The average F1, precision, and recall scores for the RTX 3080 are 96.7%, 97.4%, and 97.0%, respectively. For the RTX 3070 Ti, we obtain average F1, precision, and recall scores of 94.3%, 95.1%, and 94.8%, respectively.

Effect of bank order: Tthe classifier for the side channel attacks is robust to the bank order, and is able to fingerprint accurately over different runs of the victim where the victim's physical memory allocation is likely to be different. We believe that this is due to the virtual to physical mapping over the 2MB pages causing shifts of the banks but preserving their relationship to other banks (an effect observed by prior GPU side channel studies [13]). This causes a shift in the patterns, but preserves them, enabling accurate classification.

6 GPU-to-CPU Attacks on LPDDR5

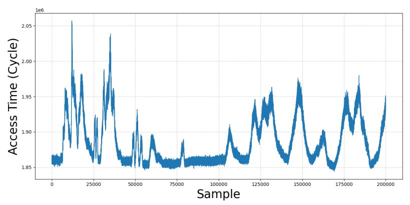
Modern Systems-on-Chip (SoC) devices are used in mobile devices and embedded systems to provide high-performance energy-efficient computational support. In an SoC, it is common for various modules (CPU, GPU, and accelerators) to share a memory system. For instance, the NVIDIA Jetson AGX Orin integrates an ARM Cortex-A78 CPU, an NVIDIA Ampere GPU, and a Deep Learning accelerator, all of which share an LPDDR5 memory [32]. Because the memory is shared, memory access patterns from one SoC module can be captured by an attacker residing on another. We confirmed that RFM leakage is also present on LPDDR5 memory (See Figure 12, similar to Figure 5(c)).

In this section, we demonstrate that it is indeed possible for a spy residing on the GPU to capture memory footprints originating from applications running on the CPU on the NVIDIA Jetson SoC, using RFM-based leakage. Specifically, we conduct three side-channel attacks: an application fingerprinting attack, a website fingerprinting attack, and a video fingerprinting attack. In all these attacks, the victim process runs on the CPU, accessing system memory shared between the GPU and CPU, while the spy process accesses memory to detect RFM leakage.

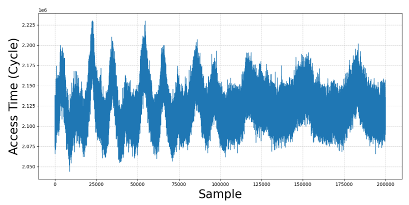
Attack 3: CPU Application Fingerprinting: We select 25 benchmarks from the SPEC2017 Benchmarks [72], that we were able to build and run successfully on the Jetson's CPU. During the attack, the spy selects 1024 cache lines and accesses them while bypassing the GPU caches. The cache lines are selected based on fixed offsets within an allocated 2MB page selected to sample different memory banks. The attacker measures the access times for each bank and adds them up for each sampling period (this makes the signal independent of the bank order) and tracks these values over time. We illustrate some of the collected application traces in Fig. 8(a)- 8(c).

For the classifier, we use a sliding window with a window size of 5,000 and a stride of 1,000 to compute the 12 statistical features (Table 5). We use the features to train classification models: XGBoost [9] with the following parameter settings: $n\_estimators = 100$, $max\_depth = 6$, and Light Gradient Boosting Machine (LightGBM) [33] with the following parameter settings: $n\_estimators = 100$, $max\_depth = -1$, and $num\_leaves = 31$. We use the 10-fold cross-validation method to avoid overfitting. LightGBM achieves the highest F1 score (93.5%), precision (94.1%), and recall (96.7%), outperforming XGBoost in all metrics (Table 4).

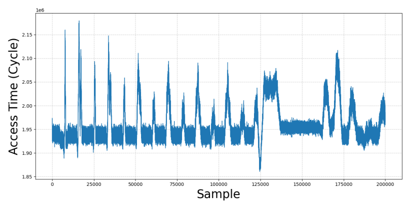
Attack 4: Website Fingerprinting: For this attack, the spy process selects 64 banks to monitor. The attacker collects 200



(a) cactuBSSN_r_507 [72]



(b) fotonik3d_r_549



(c) bwaves_s_603

Figure 8: RFM signal for three applications (y-axis is sum of all bank access times) over time (x-axis).

Table 4: CPU Application, website and video fingerprinting performance: F1 (%), Precision (%), and Recall (%).

		F1	**Prec**	**Rec**
		$\mu(\sigma)$	$\mu(\sigma)$	$\mu(\sigma)$
Attack 3	XGBoost	92.5 (1.3)	93.0 (1.2)	92.7 (1.5)
	LightGBM	**93.5 (1.3)**	94.1 (1.2)	93.7 (1.4)
Attack 4	XGBoost	89.9 (1.2)	90.5 (1.1)	90.5 (1.1)
	LightGBM	**92.1 (1.3)**	92.6 (1.3)	92.6 (1.3)
Attack 5	XGBoost	84.9 (1.8)	85.3 (2.0)	86.0 (1.7)
	LightGBM	**87.6 (1.8)**	88.1 (1.7)	88.3 (2.0)

traces for each web page accessed using Chromium 131. We select the top 50 websites from the Alexa top 1 million list [3]. Fig. 10(a)- 10(c) illustrate the sum of all bank access times on the Jetson's LPDDR5 when 3 websites (`linkedin.com`, `facebook.com`, and `taobao.com`) are browsed by the victim user. Using a similar classification methodology as Attack 3, Table 4 shows the performance of 50-class website fingerprinting. LightGBM achieves the highest F1 score (92.1%), precision (92.6%), and recall (92.6%).

Attack 5: Video Fingerprinting: For this attack, the spy process selects 16 banks. This attack is similar to website fingerprinting, but the victim plays different YouTube videos. We chose 20 random videos (mostly documentaries) for fingerprinting and classification. The attacker collects 100 traces for each video, with each trace comprising 300,000 data points. Fig. 10(d)- 10(f) illustrate the sum of all bank access times on the Jetson's LPDDR5 for three selected videos. Using a similar classification methodology, Table 4 summarizes the performance of the 20-class video fingerprinting attack across 10 folds. LightGBM attains an F1 score of 87.6%, precision of 88.1%, and recall of 88.3%, surpassing XGBoost's respective scores of 84.9%, 85.3%, and 86.0%.

7 Slow-down Attack

This section describes how an attacker can utilize the RFM features to slow down another user's workload sharing the same memory as the attacker's process. Such attackers, called *Memory Performance Hog (MPH)* in [48], can lead to several potential harmful outcomes. The slow-down can degrade the performance of an application, causing them to miss performance targets or increasing billing on a cloud systems. It is difficult to detect MPHs since they do not increase hardware utilization, unlike attacks on other shared resources.

In § 3.4, we show that by creating excess activations in DRAM rows and increasing the activation counter, we observe an increased amount of access time due to RFM command issuance by the memory controller and RFM blockage of sub-banks. An attacker, besides the covert and side channel attacks, can use this behavior to slow down other tenants/processes running concurrently on the same GPU. From § 3.4, we know that each bank in GDDR chips is divided into several sub-banks. The slow-down attacker also using the method in § 3.1, gets the last physical page for the attack. After the attacker groups the addresses according to different banks, she selects different addresses from RFM sub-banks. To ensure that the attacker only slows down due to RFM, she utilizes a small number of GPU hardware resources. The attacker after getting its DRAM addresses, launches the kernel which occupies one and a half SMs (in total 6 warps amounting to 192 threads). Each thread is activating a single row in a single sub-bank.

We use Blender benchmark as the victim from Openbenchmarking [62] to show how much slow-down an attacker can get by activating **only a single address** in different RFM sub-banks in each physical bank and this access pattern corresponds to single-sided Rowhammer attack. We experiment this on RTX 4060 with Samsung GDDR6. Normally, without any attacker to slow down, all parts of this benchmark take 23 minutes [55] to complete in total. However, when the attacker is active along with the victim, the benchmark takes 1 hour and 52 minutes [71] to complete which is over **4.8x** slow-down on average ranging up to 7 times individually.

8 Potential Mitigation

Refresh management is designed to mitigate Rowhammer but potentially creates exploitable vulnerabilities as we explored. It is important to find a balance between these goals. We offer three potential mitigations.

Partitioning. Partitioning provides a security isolation between users/processes. The RFM counters can be implemented per partition, hence different tenants will not affect each other. Although this would increase the cost, require additional hardware modification, and bring in the issue of how to partition fairly, etc., it is an effective way to provide security. The partition may include a bank/sub-bank or even a single chip to be completely devoted to a single tenant. NVIDIA provides partitioning of GPUs up to several separate GPU instances by using Multi-Instance GPU (MIG) [59]. This mechanism allows each user to have a different memory controller, therefore possibly eliminating the RFM leakage. However, MIG is supported by a limited number of GPUs (A30, A100, H100, and H200).

Alternative Rowhammer mitigations. JEDEC introduced Refresh Management in 2020 and since that time it has been implemented by different companies. Although it provides a way to solve the Rowhammer problem, there are other proposals by the researchers without relying on blocking the access as in RFM. For example, Rubix [68] disrupts the spatial correlation in the line-to-row mapping by using encrypted addresses for memory access, significantly reducing the occurrence of hot rows by 2 to 3 orders of magnitude.

New DDR5 specifications [25] come with a new Rowhammer solution called Per-Row Activation Counter (PRAC) which tracks each row in DRAM banks. Although the specification does not give detailed information on how it works, we believe the attacks described here can be extended to DDR5 systems due to 2 reasons: When a row activation counter reaches a threshold, victim rows need to be refreshed and PRAC allows additional time window for the internal refresh management which is similar to *RFM* commands; CPUs allow (DDR5 systems) sharing of a single row of DRAM bank and this gives the attacker an opportunity for finer granularity in timing leakage than what is described in this paper.

System-wide analysis. The attacks rely on system features. The high number of cache misses and DRAM accesses is the feature of these attacks, but it can also mean some other attacks, like Rowhammer, row-buffer attacks, etc. Our attacks use *discard* and *clock* instructions in PTX. As long as the timing can be measured, the side channel attacks will be possible. Even if the *clock* is disabled, the attacker can still build her own timer [13]. If the *discard* is disabled, then eviction sets can be utilized, although the bandwidth and error rate will probably suffer due to excess accesses of eviction set elements. As a result, the side channel attack, slow-down attack, and reverse engineering can still be implemented.

9 Related Work

DRAM timing side channels. Pessl et al. 2016 [65] presented covert and side channel attacks on the shared row buffer of DRAM memory modules. DRAMA presented a covert channel and side-channel attack. The threat model for the covert channel assumes parties (the receiver and the sender) occupy different rows in the same DRAM bank. We show that since the row buffer time-out period is small compared to high memory access latency, the covert channel becomes challenging to implement on GDDR memories. The threat model for the side channel assumes the spy has a different row as well as a shared row with the victim process. On GeForce devices, as we observe, it is not possible to share a DRAM row between two different processes. However, our attacks exploit a completely different leakage vector on GDDR: RFM contention leakages (see § 12). DramaQueen [78] proposed a method to infer access patterns that depend on secret information by weakening the assumptions in DRAMA [65]. DRAMD [43] is a variant of the one-row DRAM attack that further enhances the success rate of side channel attacks on Intel SGX. Jain et al. [21] proposed Fractional GPUs (FGPUs), a tool that also discovers DRAM addressing functions on NVIDIA GPUs of its time. However, this tool is complex and incompatible with the latest GPU generations, such as Ampere or Ada Lovelace. Additionally, to obtain the physical address from the driver, they added a new IOCTL (input-output control) method, but the new drivers do not support this.

GPU covert and side channel attacks. Naghibijouybari et al. [51] introduced the first covert channel attacks on GPGPUs by exploiting the contention in different micro-architectural resources. Later, Naghibijouybari et al. [52] demonstrated side channel attacks using the CUDA memory management APIs and performance counters. Nayak et al. [54] demonstrated a covert channel exploiting the shared last-level TLBs. Ahn et al. [1] demonstrated a covert channel based on the shared, on-chip interconnect channels in GPUs. Wei et al. [80] used context-switching penalties to deduce details related to the Deep Neural Network (DNN) model. Dutta et al. [13] investigated prime-and-probe covert and side channels in remote multi-GPU L2 caches. Later, Zhang et al. [82] presented covert and side channel attacks based on performance counter and contention mechanisms in multi-GPU interconnects. Our work is the first to exploit the GPU off-chip memory microarchitecture (GDDR) as well as showcase the vulnerability of commercial Rowhammer protection mechanisms.

10 Concluding Remarks

In this paper, we demonstrated new attacks that use the RFM standard designed to protect modern memory chips from Rowhammer attacks. The standard triggers refresh operations and eventually bank blocking based on shared counters of accesses at the granularity of the bank or the sub-bank: these operations lead to detectable timing differences. We demonstrated that this timing leakage can be used to construct high bandwidth covert channels and side channels on GDDR6 GPU systems. We also showed that driving banks to the blocked state can result in a substantial slowdown for other applications. GDDR systems are not vulnerable to DRAMA [65] style attacks because the memory controller has a different page policy, making our attacks the **first** microarchitectural memory attack on GDDR systems. We also presented side-channel attacks on a Jetson platform using LPDDR5, showing that RFM leakage threatens other DDR systems that use RFM.

Our current attacks are limited in granularity by the row buffer size (the counters count accesses within a sub-bank of the row buffer) and the need to reach counter thresholds where timing leakage is observed when refreshes are triggered or when blocking occurs. As a result, it is possible to detect aggregate behavior leading to different levels of access to different pages mapped to different sub-banks. We believe that some finer-grained attacks may still be possible provided that they lead to different page-level access behavior but these attacks require further reverse engineering and finer-grain attack strategies, which we hope to explore in future studies.

Acknowledgements

We sincerely thank the anonymous shepherd and reviewers for their valuable suggestions for this work. This work was partially supported by a Department of Energy CENATE grant, and National Science Foundation grant CNS-2053383.

References

[1] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 565–577, 2021.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the conference on high performance graphics 2009*, pages 145–149, 2009.

[3] Alexa. Top 1M sites (2024). `https://www.alexa.com/topsites`.

[4] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. Malicious co-residency on the cloud: Attacks and defense. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[5] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 19–24, 2018.

[6] Matthew Blackmore. A quantitative analysis of memory controller page policies. Master's thesis, Portland State University, 2013.

[7] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Oğuz Ergin, and Onur Mutlu. Understanding the security benefits and overheads of emerging industry solutions to dram read disturbance. *arXiv preprint arXiv:2406.19094*, 2024.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[9] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proc. ACM SIGKDD Conf. on Knwl. Discov. and Data Min. (KDD)*, pages 785–794, 2016.

[10] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[11] Blender Online Community. *Blender Studio Character library*, 2023.

[12] Khaled M Diab, M Mustafa Rafique, and Mohamed Hefeeda. Dynamic sharing of gpus in cloud systems. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 947–954. IEEE, 2013.

[13] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[14] NVIDIA Forums. Memory Statistics - Caches. `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticscaches.htm`.

[15] NVIDIA Forums. Memory Transactions. `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/sourcelevel/memorytransactions.htm`.

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

[17] Adam Hastings and Simha Sethumadhavan. Wac: A new doctrine for hardware security. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 127–136, 2020.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[20] Intel. Performance Differences for Open-Page / Close-Page Policy. https://cdrdv2-public.intel.com/826015/826015_Perf_Diff_Open_Pg_Rev0-9.pdf, 2024.

[21] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019.

[22] Milan Jaroš, Lubomír Říha, Petr Strakoš, and Matěj Špet'ko. Gpu accelerated path tracing of massive scenes. *ACM Transactions on Graphics (TOG)*, 40(2):1–17, 2021.

[23] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734. IEEE, 2022.

[24] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. Zenhammer: Rowhammer attacks on amd zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 2024)*, 2024.

[25] JEDEC. Double Data Rate 5 (DDR5 SDRAM) SDRAM Standard, Document JESD79-5C.

[26] JEDEC. Graphics Double Data Rate (GDDR6) SGRAM Standard, Document JESD250D.

[27] JEDEC. High Bandwidth Memory (HBM3) DRAM Standard, Document JESD238A.

[28] JEDEC. Low Power Double Data Rate 4 (LPDDR4) Standard, Document JESD209-4E.

[29] JEDEC. Low Power Double Data Rate 5 (LPDDR5) Standard, Document JESD209-5C.

[30] JEDEC. Near-term DRAM Level Mitigation, Document JEP300-1, 2021.

[31] JEDEC. System Level Rowhammer Mitigation, Document JEP301-1, 2021.

[32] Leela S Karumbunathan. Nvidia jetson agx orin series, 2022.

[33] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[34] Jing Wang Kevin M. Brandl, Kedarnath Balakrishnan and Guanhao Shen. Refresh management for dram. https://patents.google.com/patent/US11222685B2/en, January 2022.

[35] Mark J. Kilgard and Jeff Bolz. Gpu-accelerated path rendering. *ACM Trans. Graph.*, 31(6), nov 2012.

[36] Dae-Hyun Kim, Prashant J. Nair, and Moinuddin K. Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2015.

[37] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651. IEEE, 2020.

[38] Michael Jaemin Kim, Jaehyun Park, Yeonhong Park, Wanju Doh, Namhoon Kim, Tae Jun Ham, Jae W. Lee, and Jung Ho Ahn. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1156–1169, 2022.

[39] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[40] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[41] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*, pages 19–33. IEEE, 2014.

[42] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *2011 International Conference on Parallel Processing*, pages 733–742. IEEE, 2011.

[43] Zhiyuan Lv, Youjian Zhao, Chao Zhang, and Haibin Li. Dramd: detect advanced dram-based stealthy communication channels with neural networks. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1907–1916. IEEE, 2020.

[44] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 735–753. IEEE, 2022.

[45] Micron. GDDR6 SGRAM. https://www.mouser.com/datasheet/2/671/mict_s_a0009021659_1-2290912.pdf?srsltid=AfmBOopCWmIZWGKhUzM747iQaJ-eXJ-TKUQpQcp_mt0vp7DXNntih6qT, 2016.

[46] Frederic P Miller, Agnes F Vandome, and John McBrewster. Levenshtein distance: Information theory, computer science, string (computer science), string metric, damerau? levenshtein distance, spell checker, hamming distance, 2009.

[47] Tergel Molom-Ochir and Rohan Shenoy. Energy and cost considerations for gpu accelerated ai inference workloads. In *2021 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pages 1–5. IEEE, 2021.

[48] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in Multi-Core systems. In *16th USENIX Security Symposium (USENIX Security 07)*, Boston, MA, August 2007. USENIX Association.

[49] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[50] Onur Mutlu, Ataberk Olgun, and A Giray Yağlıkcı. Fundamentally understanding and solving rowhammer. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 461–468, 2023.

[51] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 354–366, 2017.

[52] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2139–2153, 2018.

[53] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[54] Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. (mis) managed: A novel tlb-based covert channel on gpus. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 872–885, 2021.

[55] Benchmark without Slow-down. https://openbenchmarking.org/result/2409051-NE-WITHOUTSL23.

[56] NVIDIA. Parallel Thread Execution. https://docs.nvidia.com/cuda/parallel-thread-execution/.

[57] NVIDIA. A100 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf, 2022.

[58] NVIDIA. GeForce RTX 40 series. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/, 2023.

[59] NVIDIA. MIG User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2024.

[60] NVIDIA. NVIDIA CUDA samples. https://github.com/NVIDIA/cuda-samples, 2024.

[61] NVIDIA. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, June, 2024.

[62] Openbenchmarking. Blender benchmark. https://openbenchmarking.org/test/pts/blender.

[63] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in*

Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[65] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.

[66] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE, 2012.

[67] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, 2009.

[68] Anish Saxena, Saurav Mathur, and Moinuddin Qureshi. Rubix: Reducing the overhead of secure rowhammer mitigations via randomized line-to-row mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 1014–1028, New York, NY, USA, 2024. Association for Computing Machinery.

[69] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15(71):2, 2015.

[70] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, Santa Clara, CA, August 2019. USENIX Association.

[71] Benchmark with Slow-down. https://openbenchmarking.org/result/2408257-NE-TEST1322675.

[72] Standard Performance Evaluation Corporation. Spec cpu2017 benchmark suite, 2017.

[73] Nikko Ström. Scalable distributed dnn training using commodity gpu cloud computing. In *Interspeech 2015*, 2015.

[74] Xiaodan Serina Tan, Pavel Golikov, Nandita Vijaykumar, and Gennady Pekhimenko. Gpupool: A holistic approach to fine-grained gpu sharing in the cloud. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 317–332, 2022.

[75] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, 2018.

[76] Guillaume Thomas-Collignon and Vishal Mehta. Optimizing CUDA Applications for NVIDIA A100 GPU. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf, 2020. S21819 GTC 2020.

[77] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[78] Victor van der Veen and Ben Gras. Dramaqueen: Revisiting side channels in dram. In *Third Workshop on DRAM Security (DRAMSec)*. DRAMSec, 2023.

[79] Jan Verschelde. Memory Coalescing Techniques. https://homepages.math.uic.edu/~jan/mcs572f16/mcs572notes/lec35.html, 2016.

[80] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 125–137. IEEE, 2020.

[81] Mochi Xue, Jiacheng Ma, Wentai Li, Kun Tian, Yaozu Dong, Jinyu Wu, Zhengwei Qi, Bingsheng He, and Haibing Guan. Scalable gpu virtualization with dynamic sharing of graphics memory space. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1823–1836, 2018.

[82] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Beyond the bridge: Contention-based covert and side channel attacks on multi-gpu interconnect. *arXiv preprint arXiv:2404.03877*, 2024.

[83] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 960–974, 2023.

[84] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+Compare: A Timer-Free GPU cache attack primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2101–2118, Philadelphia, PA, August 2024. USENIX Association.

[85] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud faas. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 133–149, 2024.

11 Ethics Considerations

All experiments in this paper were conducted on private testbed in the lab, ensuring that no harm was inflicted on other users, nor were they put at any risk. No users were affected by our experiments. We disclosed our findings and code to NVIDIA and DRAM manufacturers, including Samsung, Micron, and SK Hynix.

12 Compliance with Open Science Policy

We commit to sharing the code and data utilized and described in this paper once private disclosure with the vendors has been completed. We fully comply with the Open Science Policy and recognize the importance of demonstrating transparency, reproducibility, and accessibility in our results. We made the artifacts available at this Zenodo repository. The readers may refer to the README files for additional details on how to reproduce results.

Appendix

Figure 9 presents GDDR hashing functions on the RTX 4060 and RTX 3070 Ti. Figure 10 shows RFM timing leakages for various websites and videos. Table 5 defines the 12 statistical features used, and Table 6 lists all 20 benchmarks in our side-channel attack.

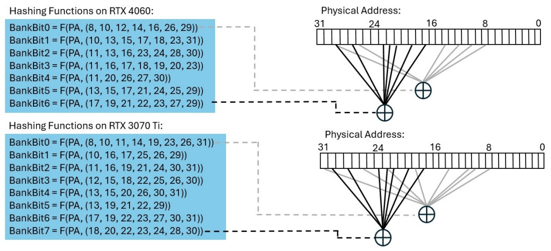

Hashing Functions on RTX 4060:
BankBit0 = F(PA, (8, 10, 12, 14, 16, 26, 29))
BankBit1 = F(PA, (10, 13, 15, 17, 18, 23, 31))
BankBit2 = F(PA, (11, 13, 16, 23, 24, 28, 30))
BankBit3 = F(PA, (11, 16, 17, 18, 19, 20, 23))
BankBit4 = F(PA, (11, 20, 26, 27, 30))
BankBit5 = F(PA, (13, 15, 17, 21, 24, 25, 29))
BankBit6 = F(PA, (17, 19, 21, 22, 23, 27, 29))
Physical Address:
31
24
16
8
0
Hashing Functions on RTX 3070 Ti:
BankBit0 = F(PA, (8, 10, 11, 14, 19, 23, 26, 31))
BankBit1 = F(PA, (10, 16, 17, 25, 26, 29))
BankBit2 = F(PA, (11, 16, 19, 21, 24, 30, 31))
BankBit3 = F(PA, (12, 15, 18, 22, 25, 26, 30))
BankBit4 = F(PA, (13, 15, 20, 26, 30, 31))
BankBit5 = F(PA, (13, 19, 21, 22, 29))
BankBit6 = F(PA, (17, 19, 22, 23, 27, 30, 31))
BankBit7 = F(PA, (18, 20, 22, 23, 24, 28, 30))
Physical Address:
31
24
16
8
0

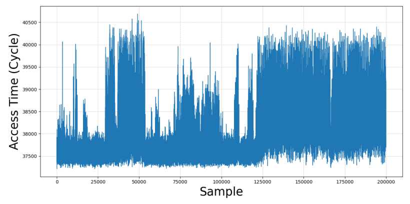
Figure 9: DRAM hashing functions for GDDR on the RTX 4060 and RTX 3070 Ti.

Table 5: Definitions of statistical features used in this work.

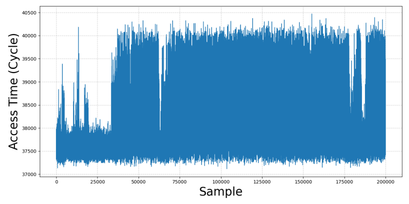
Features	Description
mean	Mean of all values.
max	Maximum of all values.
min	Minimum of all values.
median	Median of all values.
std	Standard deviation.
var	Variance.
range	Difference of maximum and minimum.
sum	Sum of all values.
count_am	Count of observations that exceed the mean.
percent_25	25th percentile value.
percent_75	75th percentile value.
iqr_val	Interquartile range.

Memory controller behaviour. We have seen that the DRAMA covert channel becomes very ineffective on tested NVIDIA GPUs. The success of this covert channel depends upon the row buffer being open until the other side of the communication channel accesses its address in the DRAM bank. If the row buffer is closed before the other side senses it, then he will not be able to measure the difference between open and closed row buffer; rather observe an empty row buffer. To analyze how much time is needed to catch the row buffer open, we carry out the following experiment, and Figure 11 shows the results of this analysis. The experiment conducts two memory accesses to the same memory bank and measures the amount by which a row buffer miss/conflict can be differentiated from all access times. Also, we increase the delay between these memory accesses. When the delay is zero, the accesses are back-to-back, meaning we do not try to put any delay. Up to about 40 clock cycles of delay, the difference caused by a row buffer conflict is at its maximum. However, as the delay increases, the conflict difference decreases, eventually hitting zero after 94 clock cycles. This indicates that after approximately 100 clock cycles, the row buffer is closed, and any subsequent access to the bank will hit an empty row buffer. Considering high global memory latency, getting a row conflict for the covert channel becomes difficult, therefore degrading the channel.

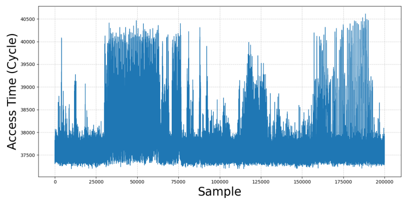
This mechanism is logical in GPUs since NVIDIA GPUs support coalescing memory access [79]. A warp can access a 128-byte (corresponding to a sector) aligned memory utilizing all 32 threads. In this case, all threads request data in a back-to-back fashion, hitting the row. Once the requests have been served and there is no other pending request hitting the same row, it is good to close the row buffer to avoid any *PRECHARGE* overhead.



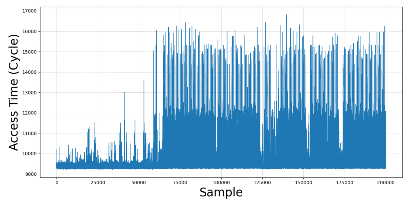
(a) linkedin.com



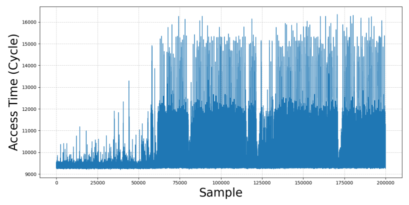
(b) facebook.com



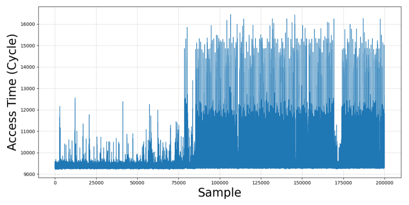
(c) taobao.com



(d) video_6



(e) video_9



(f) video_16

Figure 10: The RFM leakages from websites and videos exhibit distinct patterns. The y-axis represents the sum of all bank access times on the Jetson's LPDDR5, while the x-axis corresponds to the sample indices.

Table 6: CUDA Benchmarks evaluated in Section 5.

Benchmarks	**Description**
srad	Speckle Reducing Anisotropic Diffusion (SRAD) is a diffusion algorithm that removes image speckles.
radixSort	Radix sort algorithm sorts integers by processing digits from least to most significant.
scan	Scan is an optimized CUDA implementation of parallel prefix sum.
mergeSort	Merge sort algorithm recursively divides a list and merges the sorted parts.
transpose	Transpose is the CUDA implementation of matrix transpose.
needle	Needleman-Wunsch (Needle) is a global optimization method used in DNA sequence alignment.
simpleStreams	It utilizes CUDA streams to concurrently execute kernels while transferring data between the host and GPU device.
vectorAdd	VectorAdd computes the element-wise sum of two vectors.
simpleCUDAGraph	It shows the process of creating, instantiating, and launching CUDA Graphs utilizing the Graphs and Stream Capture APIs.
reduction	A parallel reduction algorithm that calculates the sum of a large array of values.
backprop	Backprop is a three-layer neural network used in face recognition.
gaussian	Gaussian Elimination solves systems of linear equations by transforming the matrix to row echelon form.
bf16TensorCoreGemm	It illustrates GEMM computation with the $\_\_nv\_bfloat16$ data type using the Warp Matrix Multiply and Accumulate (WMMA) API.
matrixMul	MatrixMul multiplies two matrices to produce a third matrix as the result.
graphMemoryFootprint	GraphMemoryFootprint shows how graph memory nodes reuse virtual addresses and physical memory.
dwt2d	2D Discrete Wavelet Transform (2D DWT) performs a 2-D discrete wavelet transform on an image.
graphMemoryNodes	It illustrates how memory is allocated and freed within CUDA graphs, utilizing the Graph and Stream Capture APIs.
conjugateGradient-CudaGraphs	It implements a conjugate gradient solver on the GPU, utilizing CUBLAS and CUSPARSE library functions.
histogram	Histogram illustrates an optimized implementation of 64-bin and 256-bin histograms.
convolutionSeparable	ConvolutionSeparable applies a separable convolution filter to a 2D signal using a Gaussian kernel.

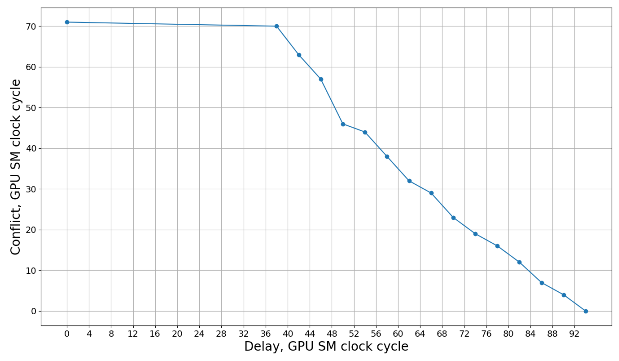


Figure 11: Memory controller timeout interval analysis on the RTX 3070 Ti. The y-axis represents the amount of conflict indicating a row buffer miss/conflict, while the x-axis shows the delay between two memory accesses.

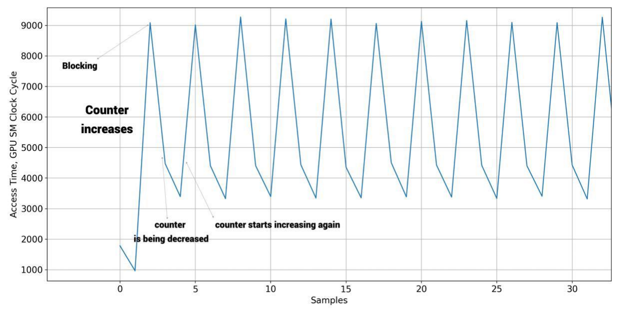


Figure 12: RFM Leakege in Jetson Orin AGX.