Behind Bars: A Side-Channel Attack on NVIDIA MIG Cache Partitioning Using Memory Barriers

Cheng Gu
University of Rochester

Reese Levine
UC Santa Cruz

Zhenkai Zhang
Clemson University

Tyler Sorensen
Microsoft and UC Santa Cruz

Yanan Guo
University of Rochester

Abstract

NVIDIA Multi-Instance GPU (MIG) is a feature designed to enable isolation and secure multi-tenancy on large data center GPUs. MIG partitions a single GPU into multiple instances, each with dedicated hardware resources such as L2 cache slices. MIG is also documented to form the foundation of NVIDIA's confidential computing stack by providing hardware-isolated trusted execution environments. However, the security claims of MIG deserve closer investigation, especially given the complexity of the GPU memory system and its many (sparsely documented) memory instructions.

In this work, we empirically examine the behavior of GPU L2 cache with MIG enabled. We find that despite the partitioning design, cross-instance L2 cache interference still occurs. Specifically, memory barriers (membars) generated in one MIG instance have side effects that propagate across L2 partitions and affect the timing of certain load operations in other instances. We also find that these membars can be triggered by specific GPU activities, such as kernel launches. Building on these observations, we develop a new timing-based side-channel attack in which an attacker in one MIG instance can infer the kernel launch patterns of a victim in another instance. We show that this attack compromises the confidentiality of widely used GPU applications, such as large language model inference, because kernel launch patterns in these applications are correlated with sensitive information.

1 Introduction

The demand for GPU-accelerated computing continues to grow, driven by advances in AI, data science, and high-performance computing [1–3]. As a result, GPU data centers are now a critical part of our computational infrastructure. Due to the high cost of GPUs, both in terms of acquisition and operation, providers must ensure high utilization for their GPUs. A common method for improving hardware utilization is multi-tenancy, where multiple users share the same physical resources. Multi-tenancy is particularly effective when individual workloads cannot fully utilize the capabilities of the hardware.

However, allowing multiple users to share a GPU introduces performance and security concerns. For example, on NVIDIA GPUs, the L2 cache is shared across all the compute units. When workloads from different users execute in parallel, contention in this shared cache can degrade performance and bring opportunities for cache timing side-channel attacks (a.k.a., cache attacks) [4–28].

To address these issues, NVIDIA introduced the Multi-Instance GPU (MIG) feature starting with its Ampere-generation data center GPUs (e.g., the A100) [29]. MIG enables a single GPU to be partitioned into isolated instances, each with its own dedicated compute and memory resources. In particular, the L2 cache is partitioned between instances. With the release of its next architecture, Hopper (e.g., the H100), NVIDIA introduced the second generation of MIG, intended to support its confidential computing framework [30]. This new MIG version still partitions the L2 cache and is therefore documented to provide a trusted execution environment with *cache side channels mitigated* [31, Figure 6].

Given the increasingly critical role MIG plays, it is important to fully understand the extent of MIG's hardware isolation and its actual security guarantees—especially for the second-generation MIG. Although several prior studies have examined the security of MIG [32,33], they focused only on the first-generation MIG and/or on hardware components that MIG does not guarantee to protect (e.g., the PCIe bus). In contrast, this paper focuses on the second-generation MIG and on the security of the partitioned and supposedly protected L2 cache. Specifically, we aim to investigate whether the L2 cache partitioning design in MIG is able to prevent all cache attacks on NVIDIA GPUs.

We conduct a series of experiments on NVIDIA Hopper GPUs with MIG enabled. Our results show that although MIG physically partitions the L2 cache between instances, cross-instance L2 cache interference still occurs. We find that this interference is caused by memory barriers. Specifically, when certain memory barrier requests are issued in one MIG

instance, the performance profiler (provided by NVIDIA) running in another instance observes additional L2 cache requests, namely, L2-level memory barrier requests (or membars for short).

Triggering membars. On GPUs, membars are used to enforce ordering of memory operations [34]. NVIDIA GPUs provide a `MEMBAR` instruction; GPU programs can directly execute this instruction to trigger membars. In addition, our investigation reveals that certain program activities that invoke GPU driver operations also trigger membars. These activities fall into three categories: 1) launching CUDA kernels, 2) calling certain CUDA memory management APIs, and 3) creating or destroying CUDA contexts (i.e., processes).

Detecting membars across MIG instances. The above observation raises security concerns: the attacker in one MIG instance can detect membars issued by the victim in another instance simply by profiling the attacker's workload. However, the profiler typically requires privileged access and is entirely disabled in security-sensitive environments [31]. This makes the profiling-based detection method less practical for the attacker. To overcome this limitation, we investigate whether the cross-instance effects of membars can be detected through alternative means—specifically, timing. Since membars are used to guarantee memory ordering, it may affect the execution time of memory instructions. Therefore, we design a set of experiments to measure the execution time of different memory instructions when membars are issued in another MIG instance. Our results show that a specific annotated load instruction, `LD.STRONG.GPU`, consistently runs slower in the presence of membars. This means that, instead of relying on the profiler, the attacker could detect membars issued in a separate MIG instance by measuring the latency of `LD.STRONG.GPU`.

Covert channels. Based on the above findings, we develop a new timing-based covert channel, Membar+Load, which operates across MIG instances. The sender in one instance transmits a bit by either repeatedly issuing membars (to send a "1") or remaining idle (to send a "0"). To issue membars, the sender can either directly execute the `MEMBAR` instruction or invoke one of the membar-generating activities (e.g., launching kernels). The receiver in another instance receives the bit by timing the execution of the `LD.STRONG.GPU` instruction: longer execution time indicates a bit "1" and shorter execution time indicates a bit "0".

Side-channel attacks. We then show that Membar+Load can also be exploited as a side-channel attack. As explained above, membars can be triggered either by directly executing the `MEMBAR` instruction or through one of the three membar-generating activities. In practice, we find that common GPU applications, such as machine learning applications, rarely execute the `MEMBAR` instruction directly. In addition, among the three types of membar-generating activities, GPU applications do not often call the CUDA memory management APIs or create/destroy CUDA contexts; however, they very frequently launch CUDA kernels. Moreover, the kernel launch patterns often reflect sensitive application information. Therefore, using Membar+Load, the attacker can detect the kernel launches in the victim workload and infer private information correlated with the kernel launch patterns.

Building on this insight, we demonstrate side-channel attacks on two GPU applications. In the first attack, we target large language model (LLM) inference. We show that different LLMs exhibit different kernel launch patterns. Therefore, the attacker can use Membar+Load to fingerprint the LLM in use. Moreover, given a specific LLM, the two primary phases of the inference process—prefill and decode—also exhibit distinct kernel launch patterns. By separating these phases, the attacker can estimate their durations and infer the number of input and output tokens, which correlate with these durations. This information can further reveal characteristics of the inference, such as the topic of the task. In the second attack, we target a different domain: graph processing. We show that the kernel launch pattern of a graph processing workload varies depending on the input graph. Therefore, the attacker can fingerprint the graph being processed from a set of candidates by detecting the kernel launches using Membar+Load. ***To the best of our knowledge, Membar+Load is the first cache attack method that works across MIG instances.***

Responsible disclosure. We disclosed our findings to NVIDIA on June 30, 2025. NVIDIA acknowledged our report and requested a three-month embargo on July 25, and lifted the embargo on October 13. In addition, AMD MI300X GPUs feature a design called Core Partitioned X-celerator (CPX), which is similar to NVIDIA's MIG. Although we were unable to test CPX due to lack of access to AMD GPUs, we disclosed our findings to AMD as well.

2 Background

In this section, we provide background information on GPUs and cache attacks. We focus on NVIDIA GPUs, since the goal of this work is to develop cache attacks that bypass the cache partitioning design on these GPUs.

2.1 GPU Architecture and Programming

GPU architecture. GPUs are designed to handle highly parallel workloads efficiently. The basic compute units in a GPU are called streaming multiprocessors (SMs). Each SM contains an array of simple cores and can execute a group of 32 threads—known as a warp—in an SIMT (single-instruction, multiple-thread) fashion. A GPU includes many SMs that operate in parallel to support massive thread-level parallelism. For example, NVIDIA H100, one of the latest server-grade GPUs, contains 132 SMs.

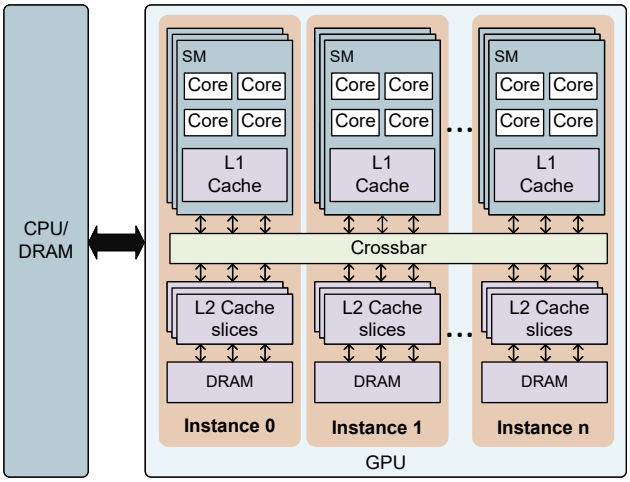


Figure 1: Overview of NVIDIA GPU architecture with MIG.

GPUs usually use their own on-board memory to store data and program state. This memory typically uses special DRAM technologies that are designed to achieve high bandwidth, such as HBM. GPU memory is often referred to as device memory; it has its own management mechanism and operates independently from the main memory (on the CPU side). When needed, data can be transferred between CPU and GPU memory over the PCIe bus.

The latency of accessing GPU memory is very high—much higher than that of main memory. To reduce the impact of this significant latency and improve performance, modern GPUs typically use a two-level cache hierarchy. Specifically, each SM has its own private data and instruction caches, and all the SMs share the L2 cache (i.e., the last-level cache). According to prior studies [35, 36], both L1 and L2 caches are set-associative; in addition, the L2 cache is non-inclusive of the L1 and is organized into slices. As on CPUs, accessing the L2 cache on GPUs is much faster than accessing the GPU memory but slower than accessing the L1 cache.

GPU programming & kernel execution. GPUs were originally designed to accelerate graphics rendering and could only be programmed using graphics-specific APIs, such as OpenGL [37] and DirectX [38]. However, modern GPUs have evolved to handle a wide range of non-graphics tasks, such as deep learning and scientific computing, and they can now be programmed through general-purpose GPU programming interfaces, such as CUDA [39] (for NVIDIA GPUs).

In CUDA, GPU tasks are written as *kernels*, which are special functions launched from the CPU but executed on the GPU. To launch a kernel, the CPU program calls the kernel function using a special syntax that specifies the number of thread blocks and the number of threads per block. The CUDA runtime forwards this request to the GPU driver, which then sets up the execution of the thread blocks on the GPU. In a typical CUDA program, before launching a kernel, it first allocates memory on the GPU (e.g., using `cudaMalloc`) and copies the required data from main memory to device memory (e.g., using `cudaMemcpy`). After executing one or more kernels, the program copies the results back to main memory, and releases the allocated device memory (e.g., using `cudaFree`).

GPU context & GPU sharing. A GPU context is conceptually similar to a CPU process: every CUDA program runs within a context on the GPU, and the context provides isolation between different programs [40]. A context is typically created implicitly when the program first interacts with GPU—for example, by calling CUDA APIs like `cudaMalloc`. Alternatively, it can be created explicitly using low-level driver calls. On NVIDIA GPUs, multiple contexts time-share the GPU by default: a hardware scheduler inside the GPU multiplexes their execution by allocating each context a time slice. However, when advanced features such as Multi-Instance GPU (discussed later) are enabled, contexts can run in parallel on the GPU.

GPU Instruction Set Architecture (ISA). NVIDIA defines two layers of ISAs for its GPUs. PTX is a virtual ISA that provides a stable interface across different GPU generations. SASS is the native ISA, which is specific to each GPU architecture. When compiling CUDA code, the code is first translated into PTX, which is then further compiled into SASS instructions that execute directly on the hardware. Unlike PTX, which is open and well-documented, SASS is a closed, proprietary ISA.

2.2 NVIDIA Multi-Instance GPU

Multi-Instance GPU (MIG) is a feature introduced by NVIDIA with the A100 GPUs in 2020. It is designed to improve GPU resource utilization and isolation in multi-tenant environments. MIG allows a single GPU to be partitioned into multiple independent instances,[1] and workloads can run in parallel in different instances. As shown in Figure 1, each instance has its own compute resources (SMs). In addition, each instance is allocated a dedicated set of memory-system resources, including on-chip crossbar ports, L2 cache slices, memory controllers, and DRAM channels. This hardware partitioning is intended to prevent interference between workloads in different instances and therefore ensures that they can run in parallel with both predictable performance and security.

In 2022, with the release of H100 GPUs, NVIDIA introduced the second-generation MIG [30]. Beyond hardware partitioning, this second-generation design includes additional hardware features to support NVIDIA's confidential computing (CC) technology. NVIDIA CC ensures that each confidential virtual machine is provided with a *hardware-isolated* trusted execution environment (TEE). Without MIG, a TEE must occupy an entire GPU (or multiple GPUs), which limits

[1] Depending on the GPU, there can be up to four or seven instances.

Table 1: Platform details.

CPU	2× Intel Xeon Gold 6526Y
GPU	1× NVIDIA H100 PCIe
GPU L2 cache	50 MB, 16-way associative
GPU memory	80 GB
Max number of MIG instances	7
Operating system	Ubuntu 22.04.4
GPU driver version	560.35.03
CUDA version	12.6
GPU DVFS	Enabled

the GPU resource utilization. However, with MIG enabled, a TEE can be assigned to just a single MIG instance instead of the whole GPU, significantly improving efficiency. With these hardware-isolated TEEs, NVIDIA claims that cache attacks have been mitigated in their CC solution [31, Figure 6].

Note that the driver support for NVIDIA CC is still in its early stages. The drivers for single-GPU CC and multi-GPU CC were released in February 2024 and February 2025, respectively [41]. As of this writing, NVIDIA has not yet released the driver for MIG+CC, and therefore we cannot examine MIG with CC enabled. However, according to the white paper for NVIDIA CC [31], MIG+CC is one of the official CC configurations.

2.3 GPU Cache Attacks

Cache attacks have been extensively studied on CPUs. Most cache attacks are based on cache evictions. For example, in Prime+Probe [7], the attacker first fills specific cache sets with its own cache lines. Then, the attacker detects the victim's accesses in these sets by observing evictions of the attacker's cache lines. More recently, researchers have shown that eviction-based cache attacks such as Prime+Probe are also feasible on GPUs [35, 36]. However, these attacks have only been demonstrated without MIG; with MIG enabled, the attacker and victim no longer share any cache sets if they are running in different instances.

3 Goal of Behind Bars

As explained in Section 2.2, with MIG enabled, each instance on the GPU is assigned a separate L2 partition, i.e., a distinct group of L2 cache slices. Based on this, NVIDIA claims that workloads running in different MIG instances *do not interfere* with each other at the L2 cache level [29, 30], and thus cache attacks are mitigated [31]. The goal of this work is to examine the extent of this mitigation—specifically, whether cross-instance L2 cache interference can still occur under the partitioning design and lead to security concerns. This investigation is critical since MIG has become a key feature in modern GPUs:

First, MIG plays an important role in multi-tenant GPU environments, since it introduces hardware-enforced isolation that provides both performance and security benefits for GPU applications. Second, the significance of MIG is further amplified in NVIDIA CC: the second-generation MIG is specifically designed to provide hardware-isolated TEEs to support NVIDIA CC (although the driver for MIG+CC has not yet been released, cf. Section 2.2).

4 Characterization of MIG L2 Cache Isolation

In this section, we conduct experiments to examine the L2 partitioning design in MIG on NVIDIA GPUs. The physical L2 cache partitioning design in MIG makes cross-instance L2 cache evictions theoretically infeasible, and we have empirically verified this in Appendix A. However, other forms of cross-instance interference may still occur. For example, the execution of certain memory instructions may have side effects that span across L2 partitions and influence memory instructions issued in other instances. Therefore, we focus on identifying and analyzing such interference.

In this section, we present only the experiments performed on the system specified in Table 1. This system features an NVIDIA H100 GPU, which is the first GPU to support the second-generation MIG design. However, we have also tested all our findings on other systems (with different GPUs and drivers), and the results are discussed in Section 7.

4.1 Cross-Instance Cache Interference

We start our investigation using NVIDIA Nsight Compute [42], which is the official CUDA profiler from NVIDIA. This profiler collects detailed performance data *for each individual CUDA kernel execution* using hardware performance counters. This data provides rich information about the kernel's L2 cache activities—such as the overall L2 hit/miss rates and the total number of L2 cache requests—which may reveal signs of potential cross-instance interference in the L2 cache.

Experiments. To detect cross-instance L2 cache interference, we profile the L2 cache behavior of a simple GPU program (the detector) in one MIG instance, while a workload with various GPU activities (the stressor) is running in another instance. We then compare the profiling results to those collected without the stressor.

Specifically, we create two MIG instances, $\texttt{Ins}_0$ for the detector and $\texttt{Ins}_1$ for the stressor. The detector launches a simple single-threaded CUDA kernel that consists of a counter loop. The code for this kernel is shown in Listing 1: it increments a counter by one in each loop iteration. We profile this detector kernel and collect the metrics related to L2 cache requests. The detector kernel itself should generate only a minimal number of L2 requests (since it is only accessing two

```
1 __global__ void detector(uint64_t NUM_ROUND){
2     uint64_t cnt = 0;
3     while (cnt++ < NUM_ROUND){}
4 }
```

Listing 1: The detector kernel.

variables); if the profiler reports otherwise, it may indicate interference from the stressor.

For the stressor, we test both user-level and driver-level GPU activities that can potentially cause L2 cache interference across instances. First, for user-level activities, the stressor consists of a long-running CUDA kernel that repeatedly issues data accesses using the `LD`, `ST`, or `ATOM` (for atomic) SASS instructions. These accesses are designed to miss in the local L1 cache and are therefore served by the L2 cache (we cover both L2 hits and misses).

Second, for driver-level activities, we test a range of common program behaviors that invoke driver routines and may implicitly trigger cache requests on the GPU. These behaviors cover all typical interactions with the GPU driver in real-world applications, including launching CUDA kernels, managing CUDA memory (via APIs such as `cudaMalloc`), and controlling CUDA execution flow (e.g., events and contexts, cf. Section 2.1).

Results. We find that none of the tested user-level activities—via the `LD`, `ST`, and `ATOM` instructions—have any observable impact on the profiling results. However, several driver-level activities substantially affect the profiling results, in particular, the value reported by the metric `lts__t_requests`. This metric reports the total number of L2 requests[2] generated by a CUDA kernel during its execution. We find that compared to when no stressor is running, three types of driver-level stressor activities cause a significant increase in the number of L2 requests recorded by the profiler for the detector kernel:

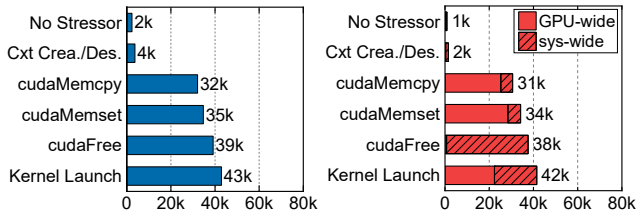
1) Launching CUDA kernels.
2) Calling `cudaFree`, `cudaMemcpy`, or `cudaMemset` APIs.[3]
3) Creating or destroying CUDA contexts. Note that creating and destroying contexts both affect the profiling results to a similar extent, so we discuss them together.

The specific number of L2 requests reported by the profiler under each of the above stressor activities is shown in Figure 2(a). Note that in our experiments, the kernel launch stressor uses a simple single-threaded kernel, while the `cudaFree`/`cudaMemset`/`cudaMemcpy` stressors all use a small 16B buffer.

One natural question here is where these additional L2 requests come from. To answer this, we further examine

[2]The prefix `lts` is used for all the metrics at the L2 cache level.

[3]Variants of these APIs (e.g., `cudaMemcpyAsync`) have the same effect.



(a) Total L2 requests. (b) L2 memory barrier requests.

Figure 2: The profiling results for the detector kernel under multiple stressors: (a) the total number of L2 requests; (b) the number of L2 memory barrier requests. For (b), we also show the breakdown: GPU-wide vs. system-wide memory barrier requests.

the categories of these L2 requests: in addition to reporting the total number of L2 requests across all types, the profiler also reports the number of requests for each individual type (e.g., `lts__t_requests_op_load` for L2 load requests and `lts__t_requests_op_store` for L2 store requests). We find that almost all these additional L2 requests caused by the stressor fall into one specific type: L2-level memory barrier requests (with the metric `lts__t_requests_op_membar`), as shown in Figure 2(b).[4]

The profiler also provides a breakdown of these memory barrier requests based on their scope: *GPU-wide* vs. *system-wide*. We find that different stressor activities trigger memory barrier requests of different scopes, as shown in Figure 2(b). For example, launching kernels issues both GPU-wide and system-wide memory barrier requests, but calling `cudaFree` triggers only system-wide memory barrier requests.

Note that the scope of a memory barrier request determines the cache level it affects (e.g., L1 vs. L2, explained further in Section 4.2). However, the profiler only reports memory barrier requests that take effect at the L2 level, namely the GPU-wide and system-wide memory barrier requests. For clarity and brevity, we refer to memory barrier requests simply as memory barriers or membars throughout the rest of this paper. In addition, all mentions of membars assume the L2 level unless stated otherwise.

What are membars? On modern GPUs, membars are special instructions that enforce ordering between memory operations issued before and after the membar. They are also called fences on some architectures. The precise semantics of membars have been explored in prior studies [34, 43]. NVIDIA GPUs feature a `MEMBAR` instruction in their SASS ISA; it supports several suffixes, including a scope qualifier (i.e., `MEMBAR.SCOPE`). This instruction ensures that all memory operations issued before the membar (by the current thread) complete within the scope defined by the qualifier before any

[4]For the stressor that creates and destroys CUDA contexts, we also observe a slight additional increase in load/store requests that miss in the L2 cache. The details are explained in Appendix A.

subsequent memory operations are executed.

According to NVIDIA [44], the scope qualifier of the MEMBAR instruction can take the values CTA (for thread block), GPU, or SYS (for system). However, using NVIDIA's cuobjdump utility from the CUDA toolkit, we discover two additional scope values: SM and VC. Empirically, we find that SM behaves similarly to CTA, and VC behaves similarly to GPU (Appendix B). Therefore, we omit these two scopes from the rest of the discussion.

Why do the activities in another instance affect the number of membars observed by the profiler? The profiling results in Figure 2 are very surprising: the profiler is configured to monitor only the detector kernel; however, the number of the reported membars varies significantly depending on the stressor activities in a separate MIG instance. With the hardware isolation promised by MIG, such cross-instance effects should not occur.

We hypothesize that the GPU driver operations triggered by these stressor activities (such as kernel launches) intrinsically issue the MEMBAR instructions on the GPU, and these instructions have effects that bypass MIG boundaries. Specifically, these instructions trigger certain predefined L2-level operations in the L2 partitions of all MIG instances on the GPU. The hardware performance counters used by the profiler capture these operations, and the profiler incorrectly attributes them to the kernel that is being profiled at the time (in our case, the detector kernel). We provide concrete experimental evidence supporting this hypothesis in Section 4.2.

> Takeaway 1: Launching CUDA kernels, calling certain CUDA memory management APIs, and creating/destroying CUDA contexts in one MIG instance can cause the profiler in another instance to observe additional membars.

4.2 Cross-Instance Membars

In this section, we conduct experiments to further analyze why certain GPU activities in one MIG instance cause the profiler in another instance to report additional membars. As explained earlier, we suspect this occurs because these activities lead to the execution of the MEMBAR instructions. To test this hypothesis, we examine whether a CUDA kernel that explicitly executes these instructions produces the same effects on the profiling results in another MIG instance.

Experiments. Similar to the experiments in Section 4.1, we run the detector (with the profiler) in $\texttt{Ins}_0$ and the stressor in $\texttt{Ins}_1$. The detector is the same as the one used in Section 4.1: it runs a CUDA kernel consisting of a counter loop. However, the stressor is different: the stressor now directly executes a CUDA kernel that repeatedly issues the MEMBAR instruction with one of the three scopes (CTA, GPU, or SYS). Listing 2 shows an example of the stressor code. In addition, we vary the number of threads used in the stressor to study how the density of the MEMBAR instructions affects the profiling results; using more threads likely causes more MEMBAR instructions to be issued in parallel.

```
1 __global__ void stressor(){
2   while (1){ asm volatile("membar.sys;":::"memory"); }
3 }
```

Listing 2: The stressor code with the MEMBAR.SYS loop. Note that the PTX-level instruction in the code will be compiled into the SASS-level MEMBAR.SYS instruction.

Table 2: The profiling results for the detector kernel; each reported value is the average of 1K runs.

Stressor	Threads	GPU-wide membars	Sys-wide membars	Total membars
None	N/A	< 1K	< 1K	< 1K
MEMBAR.CTA	128	< 1K	< 1K	< 1K
	256	< 1K	< 1K	< 1K
	512	< 1K	< 1K	< 1K
	1024	< 1K	< 1K	< 1K
MEMBAR.SYS	128	< 1K	32K	32K
	256	< 1K	61K	61K
	512	< 1K	103K	103K
	1024	< 1K	104K	104K
MEMBAR.GPU	128	56K	< 1K	56K
	256	111K	< 1K	112K
	512	194K	< 1K	194K
	1024	194K	< 1K	194K

The profiling results for the detector are shown in Table 2; we have two important observations from this table:

Observation 1. First, compared to the baseline with no stressor, a stressor executing MEMBAR.GPU or MEMBAR.SYS causes an increase in the total number of membars reported by the profiler. Specifically, the stressor with MEMBAR.GPU increases the number of GPU-wide membars, while the stressor with MEMBAR.SYS increases the number of system-wide membars. In contrast, a stressor executing MEMBAR.CTA shows no cross-instance impact on the profiling results, as this narrower-scope membar likely does not trigger global L2 operations and is thus not captured by the profiler in a separate instance.

Based on this observation, we believe it is very likely that the three types of GPU activities discussed in Section 4.1—namely, launching CUDA kernels, calling certain CUDA memory management APIs (cudaFree/cudaMemcpy/cudaMemset), and creating/destroying CUDA contexts—intrinsically issue the MEMBAR.GPU or MEMBAR.SYS instructions. These instructions then affect the profiling results in other MIG instances. In the rest of this paper, we refer to these three types of GPU activities as *membar-generating activities*.

Observation 2. From Table 2, for a stressor that executes `MEMBAR.GPU` or `MEMBAR.SYS`, increasing the number of threads in the stressor initially causes the profiler to report more membars. This is because using more threads could lead to a higher density of issued membars. However, this effect eventually reaches a limit: once the number of threads exceeds 512, further increasing the number of threads does not significantly raise the number of membars reported by the profiler, which is likely due to the limited hardware resources on the GPU (to process membars).

An immediate question here is whether changing the parameters of membar-generating activities changes the membar density. For example, does increasing the number of threads in the launched kernels or increasing the buffer size in `cudaMemset` calls causes more membars to be generated? To answer this question, we run a stressor in one MIG instance that executes one of these membar-generating activities with varying parameters; at the same time, we profile a detector kernel (the same as the one used earlier) in another instance. We find that modifying these parameters, such as the number of threads in the launched kernels, does not cause any changes in the number of profiled membars (as long as the activities are issued at the same frequency, explained next).

However, the frequency of these activities affects the number of profiled membars. For example, for the stressor that repeatedly launches kernels, inserting some waiting time between consecutive launches reduces the number of membars reported by the profiler. Figure 3 shows the number of profiled membars with different waiting times added between consecutive kernel launches: a longer waiting time results in a lower membar density and thus fewer profiled membars. We observe the same trend for other membar-generating activities as well. Note that in the experiments of Section 4.1, no waiting time is added between consecutive iterations of activities such as kernel launches; each launch occurs immediately after the previous one finishes.

> Takeaway 2: Membar-generating activities trigger the `MEMBAR.GPU` or `MEMBAR.SYS` instructions, which affect the profiling results on membars in other MIG instances. In addition, a higher membar density could lead to a greater effect.

4.3 Membar Detection

The findings in the previous sections have security implications: an attacker in one MIG instance can potentially detect the membars issued by a victim in another instance by monitoring the cross-instance effects of membars. In earlier sections, we relied on the profiler for this monitoring. However, the profiler requires privileged access and is *typically disabled* in security-sensitive scenarios such as confidential computing environments. Therefore, in this section, we explore alternative methods to detect the cross-instance membar effects. Specifically, we focus on leveraging timing information, i.e., we study whether membars issued in one instance affect the latency of operations in another instance.

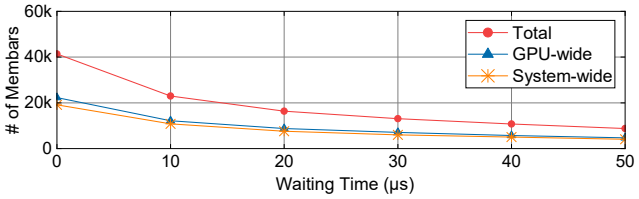


Figure 3: The profiling results on membars with varying waiting times inserted after each kernel launch in the kernel-launch stressor.

4.3.1 Impact on Load/Store Timing

As explained in Section 4.1, the purpose of membars is to enforce ordering of memory operations. Therefore, the membars issued in one MIG instance may affect the latency of load and store operations in another instance.

To test this hypothesis, we again run an experiment with two MIG instances. The detector in `Ins0` and the stressor in `Ins1` each launches a CUDA kernel (with a varying number of threads, explained next). Each thread in the stressor kernel repeatedly issues the `MEMBAR.GPU` or `MEMBAR.SYS` instructions. Each thread in the detector kernel repeatedly measures the execution time of the `LD` or `ST` instructions.[5] Note that when the detector runs multiple threads, it collects multiple execution-time samples in each measurement round, and we report only the average across all threads for each round.

To gain a comprehensive understanding of the timing impact introduced by membars, we repeat the experiments with different numbers of threads in both the stressor and the detector. In addition, we flush the L1 and/or L2 caches after each round of measurement in the detector, in order to study the impact of membars on the execution time of the `LD/ST` instructions that access data from different levels of the memory hierarchy (e.g., L2 cache vs. device memory).

Results. We find that 1) the effect the stressor has on the latency of the detector's `LD/ST` instructions, and 2) the magnitude of that effect, both depend on the configurations of the detector and the stressor, which we discuss in detail below.

Detector configuration. We observe that, compared to the baseline without a stressor, having a stressor increases the execution time of the detector's `LD/ST` instructions ***only if*** the detector is running a large number of threads in parallel, i.e., issuing a large number of load/store operations in parallel. Specifically, when the detector runs about 1K threads, there is an observable increase in the execution time of its `LD/ST`

[5] The instructions in different threads target different addresses.

Table 3: The increase in the latency of executing LD/ST when different stressors (with different membar loops and different numbers of threads) are running in a separate instance; each value is the average of 1K iterations.

Stressor		Detector (12K Threads)	
Instruction	# of Threads	LD Latency ↑	ST Latency ↑
MEMBAR.SYS	128	0.4%	< 0.1%
	256	1.0%	< 0.1%
	512	1.7%	< 0.1%
	1024	1.7%	< 0.1%
MEMBAR.GPU	128	0.8%	< 0.1%
	256	1.8%	< 0.1%
	512	3.2%	< 0.1%
	1024	3.2%	< 0.1%

instructions (with the stressor). Raising the number of threads in the detector from 1K to 12K results in a slightly larger timing increase. However, beyond this point,[6] further raising the number of threads leads to a small reduction in the timing increase. In addition, the timing increase is larger when the detector's LD/ST instructions hit in the L2 cache than when they miss. Therefore, in the following paragraphs, we focus on discussing the results where the detector runs 12K threads and all of its LD/ST instructions hit in the L2 cache.

Stressor configuration. Table 3 shows the increase in the execution time of the detector's LD/ST instructions when there is a stressor; each result is averaged over 1K iterations. We fix the number of threads in the detector to 12K (as explained above), and vary only the number of threads in the stressor. The results show that the stressor (across all tested configurations) causes an observable increase in the average execution time of the detector's LD instructions. In addition, raising the number of threads in the stressor initially leads to greater timing increases due to a higher density of membars. However, the impact eventually reaches a peak—further raising the number of threads does not result in greater timing increases. This trend matches the one shown in Table 2. Note that the stressor also increases the average execution time of the detector's ST instructions, as shown in Table 3, but this effect is much less observable than the effect for the LD instructions.

> Takeaway 3: Membars issued in one MIG instance increase the average execution time of the LD instructions in another instance when there are a large number of LD instructions issued in parallel. A higher membar density can lead to a greater timing impact.

Now, to understand how the membar-generating activities affect the latency of executing the LD/ST instructions, we

[6]This value depends on the available hardware resources in the MIG instance. We use a 1g,10g MIG instance for the detector in our experiments.

Table 4: The increase in the latency of executing LD/ST when different stressors (with different GPU activities) are running in a separate instance; each value is the average of 1K iterations.

Stressor	LD Latency ↑	ST Latency ↑
Launching kernels	0.6%	< 0.1%
Calling cudaFree	0.6%	< 0.1%
Calling cudaMemcpy	0.5%	< 0.1%
Calling cudaMemset	0.5%	< 0.1%
Crea./destr. CUDA contexts	< 0.1%	< 0.1%

re-run the experiments in Table 3 with the same detector setup; but we change the stressor from directly executing the MEMBAR.GPU/MEMBAR.SYS instructions to implicitly issuing membars through one of the membar-generating activities.

We first run the experiments with the membar-generating activities issued in a tight loop, with no waiting time inserted between consecutive iterations (cf. Section 4.2). The results, as shown in Table 4, indicate that these activities noticeably increase the average latency of executing the LD instructions in the detector, but the increases here are slightly smaller than those in Table 3.

We then run the experiments while varying the frequency of these activities by inserting waiting time in the stressor. The results show that changing the frequency of the activities alters the magnitude of the timing effect. For example, Figure 4 shows the latency increase for the detector's LD instructions when the stressor repeatedly launches kernels with different waiting times between consecutive launches. A longer waiting time results in a smaller latency increase, which is consistent with the observation in Figure 3.

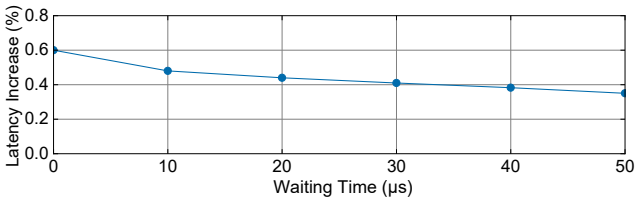
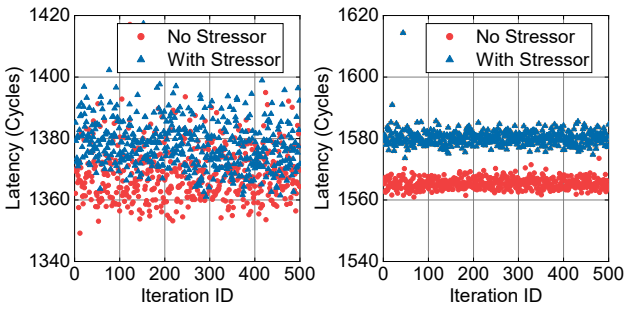


Figure 4: The increase in the latency of executing LD with varying waiting times added after each kernel launch in the kernel launch stressor.

4.3.2 Detecting Membars Using Timing Information

Although Table 3 and Table 4 show that the ***average*** latency of the LD instructions increases when membars are issued by the stressor, we find that the individual latency measurements are not very stable. Figure 5(a) shows the raw latency samples of the LD instructions from 500 iterations, both with and without the stressor. While the average values in the two cases



(a) LD latency. (b) LD.STRONG.GPU latency.

Figure 5: The latency of (a) LD and (b) LD.STRONG.GPU, measured over 500 iterations with and without the kernel-launch stressor; values are reported in GPU cycles.

Table 5: The increase in the latency of executing LD.STRONG.GPU/ST.STRONG.GPU when different stressors (with different GPU activities) are running in a separate instance; each value is the average of 1K iterations.

Stressor	LD Latency ↑	ST Latency ↑
Launching kernels	0.7%	< 0.1%
Calling cudaFree	0.7%	< 0.1%
Calling cudaMemcpy	0.5%	< 0.1%
Calling cudaMemset	0.5%	< 0.1%
Crea./destr. CUDA contexts	< 0.1%	< 0.1%

are clearly different, the individual measurements partially overlap. This overlap significantly limits the attacker's ability to reliably use this timing signal to detect membars issued in another instance.

How to reliably detect GPU activities in a different instance? By examining a large amount of NVIDIA library assembly code, we find that, in addition to the regular LD and ST instructions, NVIDIA also provides the LD.STRONG.GPU and ST.STRONG.GPU instructions. We could not find any official documentation describing the exact semantics of these instructions. However, we believe these instructions are designed to enforce stronger ordering guarantees, based on several observations from our experiments. For example, we find that PTX-level memory operations that are annotated with ordering qualifiers are all compiled to the LD.STRONG.GPU and ST.STRONG.GPU instructions. Other observations are provided in Appendix C.

To understand if the latency of executing these instructions is affected by membars, we conduct the same experiments as in Table 4, but use LD.STRONG.GPU and ST.STRONG.GPU instead of LD and ST. The results are shown in Table 5. We find that similar to the case with LD, membars in one instance cause an observable increase in the latency of executing LD.STRONG.GPU in another instance.

More importantly, we find that the latency of executing LD.STRONG.GPU is much more stable than that of LD. As shown in Figure 5(b), for LD.STRONG.GPU, there is a clear separation between the latency samples collected with and without the stressor. Therefore, by measuring the execution time of LD.STRONG.GPU, an attacker can more reliably detect membars in another MIG instance and construct powerful covert channels and side-channel attacks. We demonstrate these later in Section 5 and Section 6.

> Takeaway 4: Compared to LD, the execution time of LD.STRONG.GPU can be used to detect the membars issued in a separate MIG instance with significantly higher precision.

4.3.3 Impact on Timing of Other Instructions

We have also tested the impact of membars on the execution time of other memory-related instructions, such as MEMBAR and ATOM. We find that, similar to the case with LD, the average execution time of these instructions also slightly increases when a stressor is present, but the increases in individual measurements are not stable and therefore cannot be used to reliably detect membars.

4.4 Summary

To summarize, in this section we demonstrated that membars, which are a specific type of L2 cache requests, have effects across MIG instances on NVIDIA GPUs. Specifically, we observed the following properties of membars:

1) A CUDA program can issue membars either by directly executing the MEMBAR.GPU or MEMBAR.SYS instructions or by having one of those membar-generating activities, namely launching CUDA kernels, calling cudaFree/cudaMemset/cudaMemcpy, and creating/destroying CUDA contexts.

2) Membars issued in one MIG instance can be observed in another instance, either through the profiler or by measuring the execution time of LD.STRONG.GPU.

These observations are important because traditional eviction-based cache attacks such as Prime+Probe are no longer practical when MIG is enabled, as experimentally verified in Appendix A. However, these membar-based effects can enable a new class of cache attacks. In the next two sections, we show how our observations on membars can be exploited to build powerful cross-instance covert channels and side-channel attacks.

5 Covert Channel

The reverse-engineering results from Section 4 can be used to build a timing-based cross-instance covert channel. In this section, we first introduce the threat model, and then we explain the details of the covert channel. This covert channel also serves as a primitive for the side-channel attacks discussed later in Section 6.

5.1 Threat Model

We assume that the two key participants in the covert channel, the sender and the receiver, are unprivileged users running on the same system equipped with an NVIDIA GPU that has MIG enabled. The sender and the receiver are assigned to different MIG instances and can launch CUDA kernels only in their respective instances. We also assume that both the sender and the receiver can access processor timing information, such as the `CLOCK` register on the GPU.[7] In addition, the sender and the receiver agree on predefined channel protocols, such as the synchronization protocol. Importantly, the sender and the receiver do not need to share any data or CPU hardware resources; for example, they can run on different cores of a CPU, or even on different CPUs in a multi-CPU system.

5.2 Membar+Load

As explained in Section 4.3, the membars issued in one MIG instance affect the latency of the `LD.STRONG.GPU` instructions in another instance. Based on this timing effect, we develop a new covert channel named Membar+Load; the detailed channel protocol and its performance are described below.

Channel protocol. In Membar+Load, the sender sends one bit of information in each iteration during the transmission period. To send a bit "1", it repeatedly issues membars in its MIG instance. To send a bit "0", it does not issue any membars. The receiver receives the bit by issuing a large number of the `LD.STRONG.GPU` instructions in parallel and measuring their execution time: if it takes longer to execute these instructions, the receiver receives a "1"; otherwise the receiver receives a "0".

Channel implementation. There are multiple ways to implement Membar+Load, particularly in how the sender issues membars: the sender can issue membars either by directly executing `MEMBAR.GPU`/`MEMBAR.SYS`, or by triggering one of the membar-generating activities. In our implementation, we use the latter approach. Specifically, the sender triggers membars by launching CUDA kernels. We choose this option because it has a high potential to be exploited as a side-channel attack—many GPU applications frequently launch CUDA kernels (further explained in Section 6).

[7] Note that `CLOCK` is a user-level register, similar to `TSC` on CPUs.

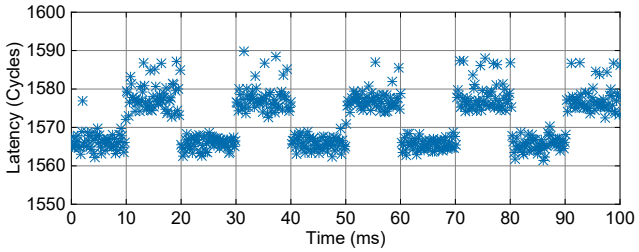


Figure 6: An example raw timing trace collected by the receiver in Membar+Load, when the sender is sending "0101010101"; the transmission interval is 10ms.

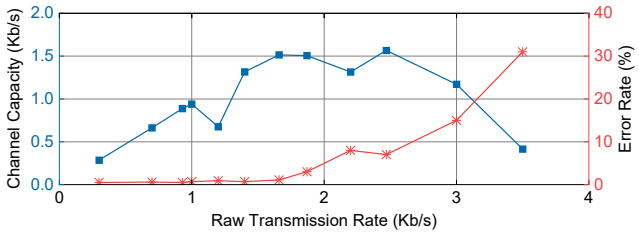


Figure 7: The channel capacities and error rates of Membar+Load.

Evaluation. We implemented a prototype of Membar+Load and tested it on the H100 GPU system specified in Table 1. Figure 6 shows an example of sending "0101010101" through this channel. To evaluate this channel, we use the metric *channel capacity* (as in prior work [45,46]) to quantify the channel throughput. It is calculated by multiplying the raw transmission rate by $(1-H(e))$, where e is the bit error rate and H is the binary entropy function. Figure 7 shows the channel capacities and bit error rates under different raw transmission rates (i.e., different transmission intervals). The channel capacity peaks at 1.57 Kb/s.

6 Side-Channel Attacks

In this section, we demonstrate that Membar+Load can be exploited for side-channel attacks across MIG instances. Specifically, by monitoring the latency of `LD.STRONG.GPU`, the attacker can detect membars issued by the victim in another instance. As explained earlier, these membars are issued either because the victim directly executes `MEMBAR.GPU`/`MEMBAR.SYS`, or because the victim triggers membar-generating activities. We find that, first of all, most GPU applications rarely use these instructions directly (for example, see a survey of kernels in prior work [47]). Second, among the three types of membar-generating activities, launching kernels occurs much more frequently than calling `cudaFree`/`cudaMemset`/`cudaMemcpy` and creating/destroying CUDA contexts. For example, Table 6 shows how many times each type of membar-generating activity is trig-

Table 6: The number of each type of membar-generating activities triggered during GPT-Neo 1.3B inference; the results are collected using NVIDIA Nsight Systems [49].

Launching kernels	Calling `cudaFree`	Calling `cudaMemcpy`	Calling `cudaMemset`	Crea./destr. contexts
19K	1	905	72	2

Table 7: The LLM fingerprinting evaluation results: macro-averaged F1 (%), Precision (%), and Recall (%).

Models	F1 $\mu(\sigma)$	Precision $\mu(\sigma)$	Recall $\mu(\sigma)$
All models	94.28 (3.12)	94.42 (3.43)	94.31 (4.79)
Encoder-Decoder	93.71 (2.38)	93.25 (2.85)	94.29 (3.75)
Decoder-Only	94.58 (3.42)	95.04 (3.55)	94.32 (5.27)

gered during a single inference pass of GPT-Neo 1.3B [48], a widely used LLM.

Given this, the attacker can use Membar+Load to detect kernel launches in the victim and leak sensitive information correlated with kernel launch patterns. We demonstrate this attack on LLM inference and a few other applications. In this section, we focus on the attack against LLM inference; the attacks on other applications can be found in Appendix D and Appendix E.

LLM inference typically consists of a sequence of kernel launches, with each kernel carrying out a specific operation such as matrix multiplication or normalization. Because these operations differ in complexity, their execution times vary. As a result, the intervals between consecutive kernel launches are not uniform, leading to variations in the kernel launch patterns. Using the receiver mechanism of Membar+Load described in Section 5, the attacker can capture a timing trace (similar to Figure 6) which reflects the kernel launch patterns during LLM inference. We find that this trace can be used to achieve two objectives: 1) fingerprinting the LLM in use, and 2) leaking the input/output sizes (which can be used to infer the topic of the inference task [50]). In the rest of this section, we first introduce the threat model, and then describe the details of these two attacks. Note that all the experiments to evaluate these two attacks are conducted on the same H100 GPU system used in previous sections (cf. Table 1).

6.1 Threat Model

We assume a similar threat model as in Section 5. Specifically, the system of interest is equipped with an NVIDIA GPU with MIG enabled. The attacker and the victim are assigned to different MIG instances. The victim performs LLM inference. The attacker's goal is to leak information about the victim, such as fingerprinting the model being used. We do not assume any shared data between the attacker and the victim, nor any privileged access to the system. In addition, we assume the victim uses a fixed batch size, e.g., one, since MIG is most likely adopted when inference is performed for personal use.

6.2 LLM Fingerprinting Attack

The goal of this attack is to fingerprint the LLM used by the victim. LLM fingerprinting has significant security and privacy implications. First, different models are tuned for different tasks; identifying the model in use can reveal the inference purpose. Second, each LLM may have its own set of vulnerabilities or limitations. Fingerprinting the model allows the attacker to launch more targeted follow-on attacks, such as crafting adversarial inputs. Third, in model-routing systems that dynamically select the LLM to use based on user inputs [51], identifying the LLM being used can leak sensitive information about the victim's input.

Attack method. Different LLMs have distinct architectures and algorithms, which result in different CUDA kernel launch patterns during inference. By monitoring these patterns, the attacker can identify the model used by the victim.

The attack consists of two phases. In the offline data collection and training phase, the attacker uses Membar+Load to collect timing traces during the inference with each LLM from a list of popular LLMs; then, the attacker trains a classifier using these traces. In the online attack phase, the attacker records the timing trace during the victim's inference and feeds it to the trained classifier to determine which LLM the victim is using.

Evaluation. We evaluate this attack method on a list of 20 LLMs. These LLMs are all from Hugging Face [52], and they cover both encoder-decoder models (for translation/summarization) and decoder-only models (for text generation). The full list is provided in Appendix F. For each LLM, we collect 300 traces. Each trace is 3 seconds long, which is sufficient for all models to complete one inference pass in our setup. We sample every 0.2 ms, which produces 15,000 data points per trace. To reduce the length of the traces sent to the classifier, we downsample each trace by a factor of 5 to reduce it to 3,000 samples. Note that when collecting the traces, we run each of the LLMs with varying prompts. We do this because the prompt can affect the collected timing trace. For example, the prompt content affects the output length, which directly influences the number of iterations in the decode stage of the inference (further explained in Section 6.3).[8] This in turn affects how many times the one-iteration decode pattern appears in the collected trace. By varying the prompts, we can capture these effects in the evaluation.

For the classifier, we use a one-dimensional CNN (1D-

[8]The maximum number of output tokens is set in the configuration.

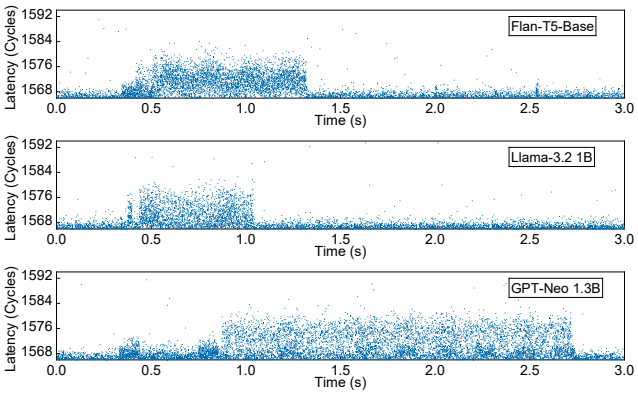


Figure 8: The raw timing traces corresponding to three widely used LLMs.

CNN), similar to prior work [46]. We use the CrossEntropyLoss function to handle the multi-class classification task and use the AdamW optimizer to adapt the learning rate. To evaluate the fingerprinting performance, we conduct 5-fold cross-validation and the results are shown in Table 7. On average, our attack achieves a classification accuracy (F1 score) of 94.28%. The accuracy for decoder-only models is slightly higher than that for encoder-decoder models. Figure 8 shows example traces collected for Llama-3.2 1B [53], Flan-T5-Base [54], and GPT-Neo 1.3B [48]. Moreover, Appendix F provides additional traces with different prompts.

6.3 LLM Input/Output Profiling Attack

LLM inference typically consists of two stages, *prefill* and *decode*. In the prefill stage, the LLM processes the entire input sequence in one forward pass to build the initial key-value (KV) cache. In the decode stage, the LLM generates output tokens one at a time, using the cached context from the prefill stage. Each decode step produces one output token, and this process repeats until the desired number of tokens is generated or a stopping condition is met. Compared to the decode stage, the prefill stage is more computationally intensive, since it processes the entire input at once. As a result, the kernels launched during the prefill stage typically have longer execution times, resulting in lower launch frequencies. This distinction makes these two stages clearly distinguishable in the timing trace captured by the attacker.

Figure 9 shows two traces captured for the GPT-Neo 1.3B model. The upper trace corresponds to a full inference pass that produces multiple output tokens. The lower trace captures only the prefill stage; this is achieved by setting the maximum output token count to 1, which effectively skips the decode stage. In the prefill stage, kernel launches are more spread out and produce a weaker timing signal. In contrast, in the decode stage, the launches are denser and result in a stronger timing impact. This distinction allows the attacker to identify

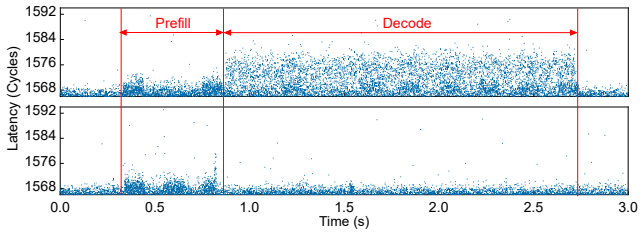


Figure 9: The raw timing traces collected for 1) both the prefill and decode stages and 2) only the prefill stage, during the inference with GPT-Neo 1.3B.

the prefill and decode stages from a captured trace and use this information to infer high-level properties of the input and output (e.g., the length), as we explain below.

First, the duration of the decode stage reflects the number of output tokens. Since the decode stage generates output tokens one at a time, longer outputs result in longer decode stages. This allows the attacker to infer the number of output tokens, which may provide clues about the output content.

We demonstrate the attack on GPT-Neo 1.3B, but it applies to other LLMs as well. We run inference with different output lengths, and our results show that different output lengths lead to clearly different decode stage durations. For example, Figure 10 shows the distribution of decode stage durations for outputs of 30, 31, and 32 tokens. To evaluate this attack, we collect traces with output lengths ranging from 30 to 100 tokens, which reflect typical chatbot responses. For each output length, we compute the kernel density estimate (KDE) of the decode stage duration and set decision boundaries at the intersections of adjacent KDE curves. Our results show that the output lengths can be distinguished at single-token granularity with an accuracy of 97.4%. Note that one token corresponds to 0.75 English words on average [55].

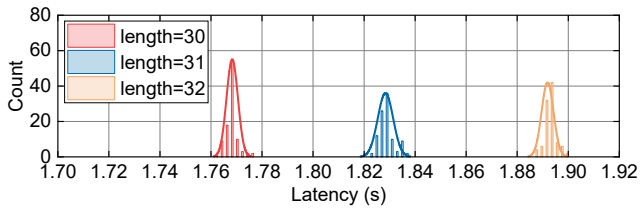


Figure 10: The distribution of decode stage duration for GPT-Neo 1.3B, with an output length of 30, 31, and 32, respectively.

Similarly, the duration of the prefill stage can reveal the input prompt length: although in the prefill stage the entire input is processed at once rather than token by token, a longer input results in a higher computational cost. The evaluation for GPT-Neo 1.3B is provided in Appendix G. Note that for certain LLMs, such as Llama-3.2 1B, inferring the input length is very difficult because its prefill stage is extremely short. However, this attack method remains effective for other

LLMs.

Using the numbers of input and output tokens together, the attacker can infer additional information about the LLM inference. For example, prior work [50] shows that this information can reveal the topic of the task: summarization tasks typically have long inputs and short outputs, while context-creation tasks are the opposite.

7 Discussion

7.1 Results with Different GPUs and Drivers

In addition to the system specified in Table 1 with an H100 GPU, we have also tested our findings on other MIG-capable GPUs, including another Hopper GPU (H200) as well as two Ampere GPUs (A30 and A100).

For the H200 GPU which also supports the second-generation MIG, we observe results consistent with those on the H100. All the takeaways from Section 4 hold on the H200, and Membar+Load works effectively, as both a covert channel and side-channel attack. In contrast, the Ampere GPUs, which feature the first-generation MIG, exhibit slightly different behavior. On these GPUs, issuing `MEMBAR.GPU` and `MEMBAR.SYS` in one instance still affects the latency of `LD.STRONG.GPU` in another. However, kernel launches do not trigger membars; only calling the CUDA memory management APIs and creating/destroying CUDA contexts trigger membars. Therefore, while our covert channel described earlier still functions, the side-channel attacks no longer work. Nonetheless, if an application frequently calls the CUDA memory management APIs (e.g., `cudaMemcpy` due to memory oversubscription [32]), it may create opportunities for side-channel attacks.

Moreover, we tested our experiments with multiple NVIDIA drivers, ranging from version 560.35.03 (released in August 2024) to 570.172.08 (released in July 2025). We also tested multiple CUDA versions, from CUDA 12.0 to CUDA 12.8. The results show that neither the driver nor the CUDA version affects our findings on both Hopper and Ampere GPUs.

7.2 Countermeasures

One possible defense against membar-based attacks is to obfuscate the timing signal by introducing additional membars. As mentioned in Section 2.2, MIG supports up to seven (or four) instances on a GPU. If the victim can acquire more than one instance, it could run a secondary workload in one instance that randomly issues `MEMBAR.GPU`/`MEMBAR.SYS`, while running the main application in another. Note that alternatively, the victim could insert `MEMBAR.GPU`/`MEMBAR.SYS` directly into its application code. However, this approach may incur high performance overhead: while membars cause only subtle timing effects on load/store operations in other instances, they significantly degrade the performance of load/store operations within the same instance.

Another potential defense is to merge the application code into fewer, longer-running kernels (which has been proposed for performance reasons [56, 57]). Our attacks rely on detecting frequent kernel launches to leak information; reducing the kernel launch frequency makes it harder for the attacker to extract useful information. However, applying this strategy requires substantial engineering effort—especially when the original code heavily relies on closed-source libraries such as `cuDNN`, which inherently use short kernels.

7.3 Limitations

A major limitation of the side-channel attacks described in Section 6, particularly the input/output length attack, is the batch size. Specifically, to accurately leak input and output lengths, we assume a batch size of one. When the batch size is larger than one, leaking input and output length for each prompt becomes much more challenging; instead, only coarse-grained information can be leaked. Taking output length as an example, each prompt in a batch may finish decoding at a different time, producing multiple distinct "drop-offs" in the timing trace. Therefore, an attacker can observe the set of output lengths, but it cannot reliably map each length to a specific prompt. This limitation is common in LLM side-channel attacks, e.g., MoEcho [58].

However, MIG is more likely to be used for lightly loaded inference scenarios, where a batch size of one is common. In contrast, larger batch sizes are typically used in large-scale deployments serving many users, which require more hardware resources and are therefore less likely to use MIG.

Table 8: The maximum channel capacity of Membar+Load with one or two co-running workloads that launch kernels at different frequencies.

Kernel launch frequency	1 KHz	2 KHz	3 KHz
One co-running workloads	1.30 kbps	1.05 kbps	0.97 kbps
Two co-running workloads	0.93 kbps	0.87 kbps	0.77 kbps

Noise tolerance. In the experiments in Section 5 and Section 6, we assume that the attacker and victim (or sender and receiver) are the only workloads running on the GPU. In practice, however, additional co-running workloads may be present, and their activities can introduce noise into Membar+Load. Specifically, membars issued by any user can perturb the attacker's timing measurements, and the attacker cannot distinguish whether a high-latency sample originates from the victim or from other workloads. Higher membar-generation rates therefore introduce more noise.

We evaluate this effect by adding co-running workloads

that launch kernels at different frequencies. While the channel capacity is reduced, Membar+Load remains functional under these conditions. Detailed results are shown in Table 8. Note that enabling seven MIG instances leaves each instance with very limited memory; in practice, deploying fewer but larger MIG instances is more realistic.

7.4 Related Work

Over the past decade, there have been many studies on GPU side channels. A large portion of them target the cache hierarchy. For example, Naghibijouybari *et al.* built covert channels using contention in the L1 and L2 caches [59]. Dutta *et al.* showed that when an attacker on one GPU can access the victim's device memory on another GPU, it can mount cross-GPU L2 cache attacks [36]. More recently, Zhang *et al.* proposed a timer-free cache attack that leverages the `CCTL.RML2` instruction (a.k.a., `DISCARD`) [35]. Beyond caches, researchers have also exploited other shared resources for covert and side channels, including GPU functional units [59], memory components [60, 61], software interfaces such as performance counters and APIs [62], and even scheduling policies [63].

However, all these attacks rely on hardware or software resources being shared between the attacker and victim—resources that MIG partitions across instances. As a result, they are no longer effective when the attacker and victim run in different MIG instances.

Attacks across GPU MIG instances. There are only a few GPU attacks that claim to work across MIG instances. First, prior work [33] showed that although MIG partitions the cache and memory system, it does not partition the Translation Lookaside Buffer (TLB) on Ampere GPUs. By creating TLB conflicts, an attacker can fingerprint the CNN models used by a victim. However, as noted in their paper, this attack is effective only during the model-loading phase. Moreover, according to NVIDIA [31], the TLB side channel has been mitigated on Hopper GPUs.

In addition, several studies have shown that an attacker can build side-channel attacks by creating and detecting contention on the PCIe bus between the CPU and GPU [32, 64]. Because the PCIe bus is shared across all MIG instances, this attack method remains possible even when MIG is enabled. However, compared to Membar+Load, this method is more susceptible to noise from other PCIe devices. Additionally, these prior studies assume either 1) the victim's DNN model does not fit entirely in GPU memory and thus the model parameters must be frequently swapped between CPU and GPU memory, or 2) the victim is training a model—rather than running inference—which involves substantial data movement over PCIe. If the victim is performing DNN inference and the entire model fits in GPU memory, the PCIe usage is minimal and this contention-based attack becomes challenging, while Membar+Load still functions.

8 Conclusion

In this work, we presented the first cache attack that operates across MIG instances on NVIDIA GPUs. We showed that although MIG partitions the L2 cache, certain L2 operations—specifically membars—can still affect the execution time of load operations in another instance. Based on this observation, we developed Membar+Load, a timing-based covert channel that functions across MIG instances. We further demonstrated that Membar+Load can also work as a side-channel attack: the attacker in one instance can detect the membar-generating activities from the victim in another instance and leak sensitive information related to those activities.

9 Acknowledgment

We thank the anonymous USENIX Security 2026 reviewers for their constructive feedback. This work was supported in part by the U.S. National Science Foundation under Grants #2530649 and #2443671.

Ethical Considerations

The ethical implications of this work were carefully considered throughout the research process. Our study exposes a new cache side-channel vulnerability in NVIDIA MIG. We believe our findings are critical for advancing GPU security research; however, they also pose potential risks if misused. Below, we outline the primary stakeholders, the impacts of our research, the steps we have taken to mitigate the risks brought by our research, and why we have decided to publish the paper.

Stakeholders. The primary stakeholders are GPU vendors, GPU users, and security researchers.

- **GPU vendors.** GPU vendors, especially NVIDIA, are directly affected because the identified vulnerability reveals design limitations in current hardware isolation mechanism on NVIDIA GPUs. We have disclosed the discovered vulnerability to GPU vendors (as explained below). This paper may cause vendors to reexamine their partitioning designs, allocate engineering resources to develop mitigations, and update guidance provided to customers.
- **GPU users.** GPU users may be impacted because the discovered vulnerability can leak sensitive information in multi-tenant environments. For example, the attack can reveal the LLM model being used or the sizes of inference inputs and outputs. These outcomes affect user privacy and may influence workload deployment decisions. At the same time, users stand to gain from increased transparency about the limitations of current GPU isolation mechanism.
- **Security researchers.** Security researchers benefit from a clearer understanding of GPU microarchitectural risks.

This work highlights gaps in current GPU partitioning techniques and can guide the development of more robust defense mechanisms.

Impacts. GPUs are increasingly used to accelerate critical workloads such as AI, and cloud providers often rely on hardware partitioning techniques like NVIDIA MIG to increase GPU utilization while protecting user security and privacy. The positive impact (beneficence) of our work is that it identifies a security flaw in the design of MIG, which enables vendors and operators to strengthen isolation mechanisms and improve protections for GPU users. It also raises awareness of GPU security and may encourage further research in this area. At the same time, the vulnerability introduces potential harms. An adversary could exploit the side channel to infer sensitive workload characteristics, such as the model used during LLM inference or the sizes of input and output data, creating tangible privacy risks. These risks are worsened when users trust that MIG fully prevents cross-tenant leakage and do not deploy additional defenses against side-channel attacks.

Mitigations. We have taken multiple steps in mitigating the discovered vulnerability. First, to reduce the risk of the discovered vulnerability being used by an adversary, we have provided practical mitigation strategies in Section 7 that users can adopt to harden their workloads against the discovered vulnerability. These measures provide an additional layer of defense while waiting for official vendor patches. In addition, we disclosed our findings to NVIDIA on June 30, 2025. NVIDIA requested a three-month embargo on July 25, 2025 and lifted it on October 13, 2025. We also disclosed our findings to AMD since AMD has CPX on their MI300X GPU, which is similar to NVIDIA's MIG. The disclosure made GPU vendors aware of the discovered vulnerability so that they can investigate internally and evaluate potential mitigation approaches. NVIDIA did not provide us with details about any mitigation plans for Hopper GPUs. However, they briefly mentioned that newer Blackwell GPUs have included a stronger MIG isolation mechanism, which reduces the signal-to-noise ratio of our attack and can therefore help mitigate it. AMD investigated our attack internally on their GPUs and informed us that they plan to release a security brief advising their customers to continue following established best practices for defending against side-channel attacks.

Importantly, all the experiments in this study were conducted in a controlled environment on machines owned solely by the authors. No public clouds or third-party systems were targeted, ensuring that no external users were harmed by our work.

Decision to publish the paper. After considering both the benefits and the potential harms of this work, we believe that publishing the paper is justified. The paper offers long-term security gains, such as raising awareness of GPU side-channel risks, encouraging follow-up research, and enabling vendors to improve their designs. Although the findings introduce risks of misuse, we provide practical defense mechanisms that GPU users and operators can adopt to reduce the risks in practice. Overall, we believe the benefits outweigh the risks.

Open Science

All the artifacts supporting this paper are permanently available at https://doi.org/10.5281/zenodo.17861917. These artifacts include 1) the source code and compilation scripts for characterizing the cross-instance impact of membars, 2) the source code and compilation scripts for Membar+Load, the cross-instance covert channel, and 3) the source code, compilation scripts, and testing scripts for the side-channel attack primitive.

References

[1] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[2] T. Liao, Y. Zhang, P. M. Kekenes-Huskey, Y. Cheng, A. Michailova, A. D. McCulloch, M. Holst, and J. A. McCammon, "Multi-core CPU or GPU-accelerated multiscale modeling for biomolecular complexes," *Computational and Mathematical Biophysics*, vol. 1, no. 2013, pp. 164–179, 2013.

[3] A. Welivita, I. Perera, D. Meedeniya, A. Wickramarachchi, and V. Mallawaarachchi, "Managing complex workflows in bioinformatics: An interactive toolkit with GPU acceleration," *IEEE transactions on nanobioscience*, vol. 17, no. 3, pp. 199–208, 2018.

[4] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security)*, 2014.

[5] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[6] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, "A new prime and probe cache side-channel attack for cloud computing," in *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015.

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE symposium on security and privacy (S&P)*, 2015.

[8] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[9] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security)*, 2020.

[10] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security)*, 2015.

[12] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[13] C. Percival, "Cache missing for fun and profit," 2005.

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*, 2006.

[15] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The sandbox: Practical cache attacks in JavaScript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[16] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[17] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *ACM SIGSAC conference on Computer and communications security (CCS)*, 2009.

[18] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *21st USENIX Security Symposium (USENIX Security)*, 2012.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM SIGSAC conference on Computer and communications security (CCS)*, 2012.

[20] ——, "Cross-tenant side-channel attacks in PaaS clouds," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[21] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[22] O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-cache," in *ACM Workshop on Computer Security Architecture*, 2007.

[23] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *Cryptographers' Track at the RSA Conference*, 2008.

[24] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+ Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[25] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and Reload–a cache attack on the BLISS Lattice-based signature scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[26] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *28th USENIX Security Symposium (USENIX Security)*, 2019.

[27] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *26th USENIX Security Symposium (USENIX Security)*, 2017.

[28] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with Flush+Reload," *Black-Hat USA*, 2016.

[29] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," available at https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[30] ——, "NVIDIA H100 Tensor Core GPU Architecture," available at https://resources.nvidia.com/en-us-data-center-overview/gtc22-whitepaper-hopper.

[31] ——, "Confidential Compute on NVIDIA Hopper H100," available at https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf.

[32] Y. Miao, Y. Zhang, D. Wu, D. Zhang, G. Tan, R. Zhang, and M. T. Kandemir, "Veiled Pathways: Investigating covert and side channels within GPU uncore," in *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[33] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "TunneLs for Bootlegging: Fully reverse-engineering GPU TLBs for challenging isolation guarantees of NVIDIA MIG," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[34] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[35] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, "Invalidate+Compare: A timer-free GPU cache attack primitive," in *33rd USENIX Security Symposium (USENIX Security)*, 2024.

[36] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Spy in the GPU-Box: Covert and side channel attacks on multi-GPU systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[37] D. Shreiner and T. K. O. A. W. Group, *OpenGL Programming Guide: The official guide to learning OpenGL, Versions 3.0 and 3.1*, 7th ed. Addison-Wesley Professional, 2009.

[38] F. Luna, *Introduction to 3D game programming with DirectX 12*. Mercury Learning and Information, 2016.

[39] NVIDIA, "CUDA C++ Programming Guide," available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[40] ——, "NVIDIA AI Enterprise Deployment Guide," available at https://docs.nvidia.com/ai-enterprise/deployment/vmware/latest/advance-gpu.html#terminology.

[41] ——, "Deployment Guide for SecureAI," available at https://docs.nvidia.com/cc-deployment-guide-tdx.pdf.

[42] ——, "NVIDIA Nsight Compute," available at https://developer.nvidia.com/nsight-compute.

[43] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A formal analysis of the NVIDIA PTX memory consistency model," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[44] NVIDIA, "Parallel Thread Execution ISA Version 8.8," available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[45] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the Ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *30th USENIX Security Symposium (USENIX Security)*, 2021.

[46] D. R. Dipta and B. Gulmezoglu, "DF-SCA: Dynamic frequency side channel attacks are practical."

[47] T. Sorensen and A. F. Donaldson, "Exposing errors related to weak memory in GPU applications," in *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[48] "GPT-Neo 1.3B," available at https://huggingface.co/EleutherAI/gpt-neo-1.3B.

[49] NVIDIA, "NVIDIA Nsight Systems," available at https://developer.nvidia.com/nsight-systems.

[50] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[51] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay, "Routerbench: A benchmark for multi-LLM routing system," *arXiv preprint arXiv:2403.12031*, 2024.

[52] "Hugging Face," available at https://huggingface.co/.

[53] "Llama-3.2-1B," available at https://huggingface.co/meta-llama/Llama-3.2-1B.

[54] "FLAN-T5 base," available at https://huggingface.co/google/flan-t5-base.

[55] OpenAI, "Key concepts to understand when working with the OpenAI API," available at https://platform.openai.com/docs/concepts/tokens.

[56] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–14.

[57] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for GPUs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[58] R. Ding, T. Xu, X. Shen, A. A. Ding, and Y. Fei, "Moecho: Exploiting side-channel attacks to compromise user privacy in mixture-of-experts llms," in *2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025.

[59] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on GPGPUs," in *Proceedings of the 50th annual IEEE/ACM international symposium on microarchitecture (MICRO)*, 2017, pp. 354–366.

[60] T. Sorensen and H. Khlaaf, "LeftoverLocals: Listening to LLM responses through leaked GPU local memory," *arXiv preprint arXiv:2401.16603*, 2024.

[61] R. Nazaraliyev, Y. Zhang, S. B. Dutta, A. Marquez, K. Barker, and N. Abu-Ghazaleh, "Not so Refreshing: Attacking GPUs using RFM rowhammer mitigation," in *34rd USENIX Security Symposium (USENIX Security)*, 2025.

[62] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU side channel attacks are practical," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security (CCS)*, 2018.

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

[64] M. Tan, J. Wan, Z. Zhou, and Z. Li, "Invisible Probe: Timing attacks with PCIe congestion side-channel," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[65] "Stanford Network Analysis Project," available at http://snap.stanford.edu/data/index.html.

[66] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[67] R. Kraus, "Traffic Camera Object Detection," available at https://www.kaggle.com/datasets/ryankraus/traffic-camera-object-detection?resource=download.

[68] "Google-switch-base-8," available at https://huggingface.co/google/switch-base-8.

Appendix

A Cross-Instance Cache Eviction

Since each MIG instance has its own L2 cache partition that cannot be accessed by other instances, workloads running in one instance should, in theory, never be able to evict data from the L2 partition of another instance. Here, we empirically verify whether this claim always holds: we check if any L2 set of one instance observes additional evictions when certain GPU activity is present in another instance.

Experiments. We use two MIG instances in our experiment, $\texttt{Ins}_0$ and $\texttt{Ins}_1$. In $\texttt{Ins}_0$ we run a stressor workload that triggers various types of user-level and driver-level GPU activities, which are exactly the same with the ones in Section 4.1. Specifically, user-level activities include directly issuing data accesses using the `LD`, `ST`, or `ATOM` SASS instructions. Driver-level activities include launching CUDA kernels, managing CUDA memory, and controlling CUDA execution flow (e.g., events and contexts). The details of these activities can be found in Section 4.1.

In $\texttt{Ins}_1$ we run a detector workload that examines whether any of these activities cause evictions in the detector's L2 cache partition. To achieve this, the detector first constructs an eviction set (i.e., a group of cache lines mapped to the same L2 set) for each cache set in the L2 partition of $\texttt{Ins}_1$. This is done using the method from prior work [35]. With these eviction sets, the detector then uses Invalidate+Compare [35], the GPU version of timer-free Prime+Probe, to detect evictions in its own L2 partition: it fills all the cache sets (within the L2 partition) using the cache lines in the eviction sets, waits for a period of time, and then checks how many of these cache lines in each L2 set have been evicted.

Results. We repeat the detector 1K times (with each stressor) and we find that, all the tested user-level activities do not trigger any cross-instance L2 cache evictions.

In addition, most of the driver-level GPU activities from the stressor also do not trigger any cross-instance L2 evictions. However, there is one exception: when the stressor repeatedly destroys CUDA contexts, all the cache lines in each L2 set of the detector's instance ($\texttt{Ins}_1$) are always evicted. Then, by varying the number of MIG instances enabled and adjusting the assignment of instances for the stressor and detector, we find that *destroying a CUDA context in any MIG instance triggers an L2 cache flush across the entire GPU, evicting data in all L2 partitions belonging to all MIG instances.* This also explains why, in Figure 2, the increase in the total number of L2 requests is greater than the increase in the number of membars for the context-creating/destroying stressor.

Is this a security concern? CUDA context creation and destruction do not occur frequently in modern GPU applications. They typically happen only at the beginning and end of an application, which makes this cross-instance eviction difficult

for an attacker to exploit.

B MEMBAR.SM and MEMBAR.VC

We find that, compared to the baseline with no stressor running, the stressor issuing `MEMBAR.SM` does not affect the number of membars observed by the profiler in the detector's instance, similar to the stressor issuing `MEMBAR.CTA`. In contrast, the stressor issuing `MEMBAR.VC` increases the number of GPU-wide membars in the profiling results, and the magnitude of this increase is the same as when the stressor issues `MEMBAR.GPU`.

C Observations on LD/ST.STRONG.GPU

We have two observations. First, when a GPU buffer is declared with `cuda::atomic`, all load and store operations to this buffer are compiled into the `LD.STRONG.GPU` and `ST.STRONG.GPU` instructions. Second, according to NVIDIA, the `__ldcg()` function can be used to load a cache line while bypassing the L1 cache. We find that using this function causes the corresponding load operation to be compiled into a `LD.STRONG.GPU` instruction.

D Attack on Graph Processing Workloads

For graph processing workloads, such as breadth-first search (BFS), the size of the input graph—specifically the number of nodes and edges—affects both the number of CUDA kernels launched and the execution time of each kernel. Different graphs often have different sizes and structures, which in turn lead to distinct kernel launch patterns for the same workload. By detecting the kernel launches during the execution of a graph processing workload using Membar+Load, an attacker can fingerprint the graph being analyzed by the workload from a set of known graph candidates.

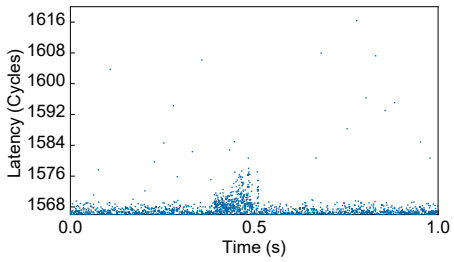
We demonstrate this fingerprinting attack using five social network graphs from Stanford Network Analysis Project (SNAP) [65]: `feather-lastfm-social`, `ego-Facebook`, `musae-twitch`, `gemsec-Deezer`, and `twitch-gamers`. Similar to the fingerprinting attack on LLMs, we build a classifier on timing traces collected during the execution of BFS with each graph as the input, and conduct 5-fold cross-validation. Our results show that the classifier achieves a macro F1 score of 98.4%, recall of 98.5%, and precision of 98.4%.

E Attack on Other AI Workloads

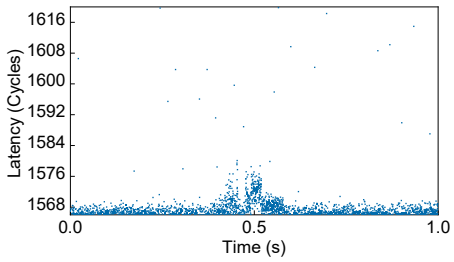
In Section 6, we discussed how Membar+Load can be used to leak information from LLMs. Here, we extend this discussion to other types of AI models. Specifically, for some modern AI models, certain input characteristics directly influence the kernel launch patterns during inference and can therefore be leaked using Membar+Load.

First, in object detection models such as Fast R-CNN [66], the number (and spatial distribution) of detected objects in an input image affect the amount of region-based computation performed in later stages, such as region proposal processing. Consequently, inputs with different object counts induce observable differences in the number, ordering, and timing of kernel launches. For example, Figure 11 shows timing traces captured using Membar+Load while running Fast R-CNN on inputs containing only five objects and more than 30 objects, respectively; the two traces exhibit clear differences.

As a result, Membar+Load can be used to infer the number of detected objects in input images, potentially exposing sensitive information, such as traffic conditions, when the inputs are collected from traffic cameras or dash cameras. To demonstrate this potential, we selected 50 images from the Traffic Camera Object Detection dataset [67] and divided them into two classes—light traffic and heavy traffic—based on the number of objects present. We then trained a binary classifier on the timing traces collected during Fast R-CNN inference. Our results show that the classifier achieves an F1 score of 83%.



(a) Trace with five objects.



(b) Trace with over 30 objects.

Figure 11: The traces captured using Membar+Load when running Mask R-CNN with different numbers of objects in the input.

Second, for mixture-of-experts (MoE) models, prior work [58] has shown that different input prompts can result in different token loads across experts, i.e., the number of tokens routed to each expert. This expert-level load information can, in turn, reveal characteristics of the input prompts. Prior work demonstrated this leakage using performance-counter-based

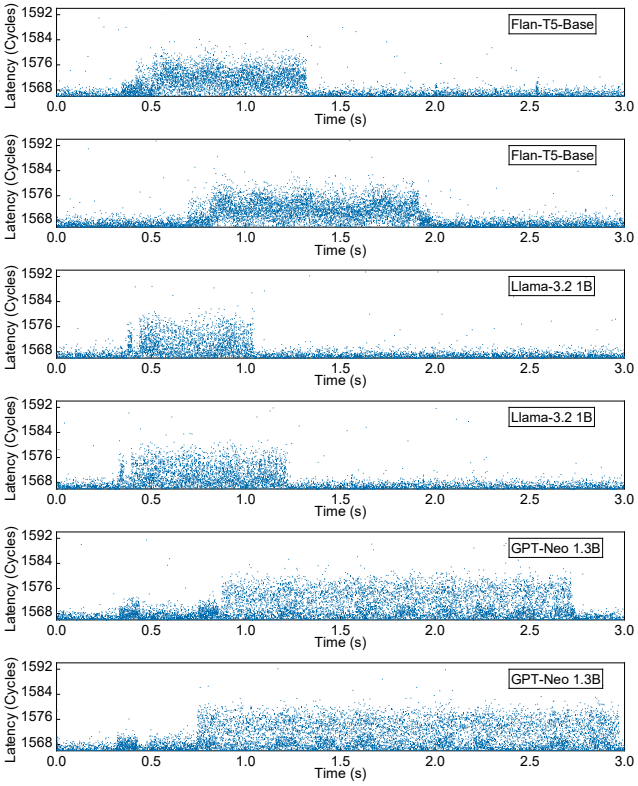


Figure 12: The raw timing traces corresponding to three widely used LLMs with different prompts.

attacks; the same leakage can potentially be exploited using Membar+Load. To demonstrate this potential, we selected 20 prompts with the same total number of tokens but that induce different token-routing decisions across experts, for a popular MoE model, switch-base-8 [68]. We then collected timing traces using Membar+Load while running the MoE model with each prompt and trained a classifier to fingerprint the prompts based on these traces. Our results show that the classifier achieves an F1 score of 99%.

F LLM Fingerprinting

Table 9 provides the full list of LLMs in our LLM fingerprinting attack. Figure 12 shows the raw latency traces collected by the attacker when different prompts are used.

G LLM Input Profiling Attack

Similar to the attack on the number of output tokens, the duration of the prefill stage can be used to estimate the number of input tokens. However, we find that this method yields lower accuracy for input tokens than for output tokens. This is because, unlike the decode stage—which generates tokens sequentially—the prefill stage processes the entire input sequence at once. While a longer input leads to a higher computational cost (more attention operations and more memory movement), the scaling is sublinear with respect to the number of input tokens. Consequently, variations in input length have a more limited effect on prefill timing. In our experiments with GPT-Neo 1.3B, we are able to distinguish input lengths only at a granularity of about 15 tokens ($\sim$11 words) with an accuracy of over 90%.

Table 9: The LLMs tested; for the LLMs that cannot fit in the GPU memory by default, we the use 4-bit or 8-bit quantized version.

1. google/flan-t5-large	11. facebook-bart-large
2. google/flan-t5-base	12. google/pegasus-xsum
3. meta-llama/Llama-3.2-1B	13. google/switch-base-8
4. EleutherAI/gpt-neo-1.3B	14. microsoft/phi-1_5
5. EleutherAI/gpt-neo-125M	15. Qwen/Qwen3-0.6B
6. EleutherAI/pythia-1B	16. titiuae/Falcon-E-3B-Instruct
7. EleutherAI/pythia-410m	17. tiiuae/falcon-rw-1b
8. distilgpt2	18. google/gemma-3-1b-it
9. Helsinki-NLP/opus-mt-en-de	19. facebook/opt-1.3B
10. facebook/bart-base	20. kurakurai/Luth-0.6B-Instruct