Trapped Handler: A Practical Performance Degradation Attack on NVIDIA vGPU via Preemption Manipulation

Cheng Gu
University of Rochester
Rochester, New York, USA
cgu11@ur.rochester.edu

Fatemeh Moradihaghighi
Clemson University
Clemson, South Carolina, USA
fmoradi@clemson.edu

Daniel Genkin
Georgia Tech
Atlanta, Georgia, USA
genkin@gatech.edu

Zhuo Li
Hydrox AI
San Jose, California, USA
zhuoli@hydrox.ai

Mengyuan Li
USC
Los Angeles, California, USA
mli49061@usc.edu

Zhenkai Zhang
Clemson University
Clemson, South Carolina, USA
zhenkai@clemson.edu

Yanan Guo
University of Rochester
Rochester, New York, USA
yguo51@cs.rochester.edu

Abstract

NVIDIA GPU virtualization technology (vGPU) enables multiple virtual machines (VMs) to share a single physical GPU and is widely adopted by major cloud providers. However, sharing a GPU introduces the risk of performance interference between co-located VMs. To mitigate this, NVIDIA provides three scheduling policies that govern how GPU time is allocated among VMs and provide Quality of Service (QoS) guarantees for each VM.

In this paper, we show that these guarantees can be fundamentally violated. Specifically, we reverse engineer the preemption mechanism on NVIDIA GPUs and uncover a severe vulnerability in it: a VM can directly modify the routine that handles preemption to extend the preemption duration and reduce the effective GPU time available to co-located VMs. Exploiting this vulnerability, we develop a performance degradation attack in which an attacker VM can significantly slow down a co-located victim VM. We show that in vGPU environments with T4 and A10 GPUs, our attack achieves degradations of up to 1894.9× and 429.3× under the default scheduling policy, and 41.1× and 6.2× under policies with stronger QoS guarantees. We also evaluate our attack on several public cloud platforms such as Alibaba Cloud and Azure. The results show that our attack is effective on all the tested cloud platforms.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

GPU, Performance Interference, Preemption



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846621

ACM Reference Format:
Cheng Gu, Fatemeh Moradihaghighi, Daniel Genkin, Zhuo Li, Mengyuan Li, Zhenkai Zhang, and Yanan Guo. 2026. Trapped Handler: A Practical Performance Degradation Attack on NVIDIA vGPU via Preemption Manipulation. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846621

1 Introduction

The demand for GPU-accelerated computing continues to grow, particularly in domains such as machine learning, data analytics, and scientific computing [16, 19, 49]. As a result, GPUs have become a critical component of today's computing infrastructure. Many cloud providers now offer GPU-powered virtual machines (VMs) to their customers [5, 6, 8, 45]. Traditionally, GPUs in cloud environments are only provided in a pass-through manner, where an entire physical GPU is assigned to a single VM. However, this approach can lead to poor utilization, as many workloads do not fully occupy the GPU. To address this, many cloud providers, such as Microsoft Azure [23], Vultr [46], Alibaba Cloud [3], and Amazon Web Services (AWS) [4], have adopted NVIDIA's GPU virtualization technology (vGPU), which allows multiple VMs to share a single physical GPU [36].

A critical concern introduced by this GPU sharing is performance interference between co-located VMs. Specifically, if one VM can significantly degrade the performance of another VM running on the same GPU, the consequences can be severe. First, many GPU applications are latency-sensitive; performance degradation can affect usability or even violate service-level agreements. Second, cloud providers commonly charge users based on usage time; therefore, performance degradation can directly translate to financial loss for the victim.

NVIDIA's proprietary vGPU scheduler plays a key role in controlling such performance interference: the scheduler determines how GPU time is allocated among co-located VMs, which directly affects the performance of each VM. Specifically, in most vGPU settings, VMs share the physical GPU through time slicing. The vGPU

scheduler selects a VM to run in each time slice, and the selection is determined by the scheduling policy in use: NVIDIA provides three scheduling policies—Best Effort, Equal Share, and Fixed Share [37]. These policies are designed to provide different levels of guarantees on the GPU time allocated to each VM. By doing so, they bound or even eliminate the impact one VM's activity can have on the GPU time allocated to other VMs, which is essential for practical vGPU deployments in the cloud.

However, vulnerabilities in the scheduler may allow one VM to occupy the GPU longer than permitted and enable performance degradation attacks on co-located VMs. Such scheduling-based attacks have been well studied on the CPU side [2, 40, 44, 52], yet much less is known about their GPU counterparts. Moreover, unlike CPU schedulers, which are implemented per hypervisor and vary across platforms, all cloud platforms that deploy vGPU rely on NVIDIA's proprietary vGPU scheduler for GPU scheduling. This means a vulnerability in this scheduler can affect every cloud platform with vGPU. Thus, given the critical role the vGPU scheduler plays, we ask the following questions:

Are the vGPU deployments in the cloud vulnerable to scheduling-based attacks? If so, what are the consequences of such attacks?

GPU preemption mechanism. To answer these questions, in this paper we investigate the security issues in vGPU scheduling. We begin by examining how NVIDIA GPUs implement preemption, since preemption is a key mechanism for enforcing vGPU scheduling policies: when a time slice ends and the scheduler selects a different VM for the next slice, it must preempt the currently running VM's execution to transfer control. Unfortunately, NVIDIA does not provide any documentation about the implementation of their preemption mechanism. Therefore, we conduct a series of experiments to reverse-engineer it. From the results, we find that in both native and virtualized environments, there exists a severe security vulnerability in the preemption mechanism on NVIDIA GPUs: despite being a privileged system operation, preemption can be directly manipulated by an unprivileged GPU user. Specifically, a user can modify the trap handler of its own GPU context—a routine that is invoked during preemption—to extend the duration of preemption far beyond normal. For example, on NVIDIA T4 GPUs, the typical duration of preemption is under 0.5 ms, but this vulnerability allows it to be increased to around 4000 ms.

Security implications. This vulnerability brings severe security concerns, especially in vGPU environments: in theory, one VM can extend the duration of preemption at the end of each time slice assigned to this VM and occupy the GPU much longer than normal. This can reduce the effective GPU time available to co-located VMs and degrade their performance. This degradation can be significant since a time slice is only a few milliseconds, while the extended preemption can last several thousand milliseconds.

We then verify whether these security implications hold in practice under each of the three scheduling policies in vGPU environments. We find that under the default Best Effort policy, one VM can directly exploit the vulnerability to occupy the GPU longer than allowed and slow down co-located VMs. In contrast, under Equal Share and Fixed Share—which are designed to provide strong performance isolation between VMs—directly exploiting the vulnerability is not effective. Specifically, the vGPU scheduler tracks the GPU time usage of each VM (including the preemption time) and applies compensation when it detects that a VM is consuming more than its allocated share. However, we identify an important limitation in this mechanism: the compensation is not applied when the trap handler generates a certain error message. Based on this, we develop an approach to bypass the compensation mechanism and then exploit the vulnerability in preemption to cause significant slowdowns on co-located VMs.

A scheduling-based performance degradation attack. Using the above techniques, we propose a scheduling-based attack targeting vGPU environments. In this attack, one VM (the attacker) deliberately manipulates the preemption process to occupy the GPU much longer than what the vGPU scheduler permits and degrade the performance of a co-located VM (the victim). We evaluate this attack with common GPU applications on T4 and A10 GPUs, which are among the most widely deployed GPUs for vGPU on public clouds. The results show that, under the default Best Effort policy, the attacker can slow down the victim by 302.5× to 1707.2× on average. Even under the Equal Share and Fixed Share policies—which are designed to provide strong performance guarantees for each VM—the slowdown remains 5.9× to 39.2× on average.

Evaluation on public clouds. To demonstrate the real-world impact of our attack, we evaluate it on public cloud platforms under controlled conditions (i.e., without other users on the same GPU). Specifically, we rent vGPU instances and examine our attack on Vultr, Alibaba Cloud, and Azure. We find that Vultr uses the default Best Effort policy, while both Alibaba Cloud and Azure use the Fixed Share policy (which provides the strongest performance guarantees for VMs among the three policies). More importantly, on all these tested platforms, our attack remains effective and allows one VM to significantly degrade the performance of another, with a magnitude between 4.0× and 436.4×.

Responsible disclosure. We have disclosed our findings to NVIDIA, Vultr, Alibaba, and Microsoft.

2 Background

In this section, we discuss background on NVIDIA GPUs that is relevant to our attack.

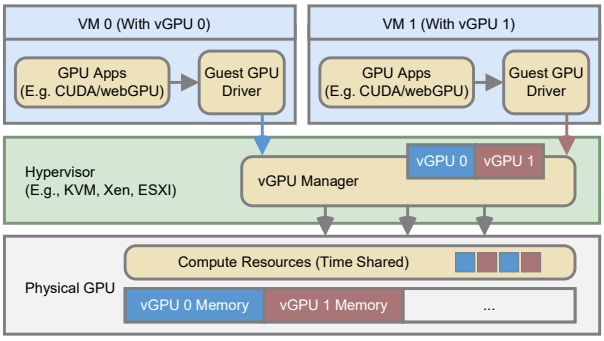


Figure 1: An overview of NVIDIA vGPU.

2.1 NVIDIA GPU Overview

GPU architecture. GPUs are designed to efficiently execute highly parallel workloads. The basic compute units in a GPU are called

streaming multiprocessors (SMs). Each SM contains multiple cores and executes a group of threads (a.k.a., a warp) in parallel. A GPU consists of many SMs. In addition, GPUs typically use on-board device memory to store data and program state. To reduce memory access latency and improve performance, modern GPUs employ a two-level cache hierarchy, which consists of a private L1 cache per SM and a shared L2 cache across all SMs.

GPU programming and execution model. GPUs were originally designed to accelerate graphics rendering and could only be programmed using graphics APIs, such as OpenGL [41]. However, modern GPUs have already evolved to support a wide range of non-graphics applications, such as deep learning and scientific computing. As a result, GPUs can now be programmed using general-purpose programming frameworks, such as CUDA [29].

In general-purpose GPU programming, computations to be executed on the GPU are written as *kernels*, which are functions annotated with special qualifiers. To execute a kernel, the CPU program calls the kernel function using a special syntax, which generates a kernel launch request sent to the GPU driver. The driver prepares the kernel for execution by setting up the necessary execution state on the GPU and then places a kernel launch command into the GPU command queue. The GPU fetches the command from this queue, schedules the kernel on its SMs, and then starts the execution.

GPU context and preemption. On modern GPUs, each program runs within a GPU context (which is similar to a CPU process) [30]. A context defines the program's execution environment on the GPU and stores its execution state, such as kernel state, register state, and allocated GPU resources.

Multiple contexts can exist simultaneously on a GPU, but they cannot run in parallel on separate SMs unless advanced features like Multi-Instance GPU (MIG) [32] are supported and enabled. By default, contexts time-share the GPU hardware. Prior research has shown that native GPU systems (without virtualization) employ a time-sliced scheduling policy for all the active contexts [18]. Specifically, each time slice is approximately 2 ms. If a kernel's execution exceeds this window, the GPU may preempt its context and yield execution to another active context.

NVIDIA SASS ISA. SASS is NVIDIA's native GPU instruction set. GPU programs written in CUDA or other high-level frameworks must be compiled into SASS instructions before they can be executed on the GPU. On Volta and later NVIDIA GPUs, each SASS instruction is 16 bytes long and encodes the opcode, operands, and control information. The SASS ISA is proprietary. NVIDIA provides only sparse documentation on it [28], which does not cover the full instruction set or its encoding. However, prior reverse-engineering studies have found that SASS instructions often include modifiers that determine the detailed behavior of the instruction [1, 14]. For example, `LD` may include modifiers that specify the access size, such as `LD.64` for a 64-bit load and `LD.128` for a 128-bit load.

2.2 NVIDIA Virtual GPU (vGPU)

NVIDIA vGPU is a GPU virtualization technology that allows multiple VMs to share a single physical GPU. It has been adopted by many cloud providers including Azure, AWS, Alibaba Cloud, Tencent Cloud, CloudFerro, and Vultr [3, 4, 9, 23, 43, 46]. Figure 1 shows an overview of the vGPU architecture. A vGPU manager running in the hypervisor is responsible for creating and managing vGPUs. Each vGPU is assigned a portion of the GPU physical memory, i.e., GPU memory is spatially partitioned across vGPUs. Each vGPU can be assigned to a VM. To use a vGPU, a VM must install and run a guest NVIDIA driver. This guest driver communicates with the vGPU manager through a virtualized interface provided by the hypervisor. Note that both the vGPU manager and the guest driver are proprietary and closed-source.

Although vGPU enables multiple VMs to share a GPU, in most vGPU settings, the execution of GPU workloads in different VMs is not carried out in parallel, but rather in a time-sharing manner.[1] This is similar to the native environment (cf. Section 2.1). Specifically, unlike GPU memory, GPU compute resources such as SMs are not spatially partitioned across vGPUs. Instead, vGPUs gain exclusive access to all the compute resources in turn through time slicing.

In each time slice, the vGPU scheduler (which is part of the vGPU manager) selects a vGPU to run. Overall, the scheduler selects vGPUs in a round-robin manner, but the exact behavior depends on the scheduling policy in use: NVIDIA vGPU provides three scheduling policies that determine how GPU time is divided among vGPUs [37]. The administrator can select the policy through the vGPU manager:

- **Best Effort (default):** The Best Effort scheduling policy assigns GPU time based on the actual demand of each vGPU: it skips a vGPU if it has no work to run. As a result, a vGPU may receive more GPU time when other vGPUs are idle or not fully utilizing the GPU. This policy aims to *maximize overall GPU utilization*, and it cannot guarantee fixed or fair allocation of GPU time among vGPUs.
- **Equal Share:** Under this policy, the GPU time is divided evenly among all active vGPUs, regardless of whether they have work to run. For example, if there are four active vGPUs, each of them receives approximately 1/4 of the GPU time. When the number of active vGPUs changes (i.e., when a VM starts or shuts down), the GPU time allocated to each vGPU adjusts accordingly. This policy provides stronger performance isolation among VMs compared to the Best Effort policy.
- **Fixed Share:** Under the Fixed Share policy, each vGPU is assigned a fixed portion of GPU time. This portion is determined by the vGPU profile and does not change when other vGPUs become active or inactive (i.e., when other VMs start or shut down). For example, on a T4 GPU, a T4-4Q vGPU always receives 1/4 of GPU time (and 1/4 of GPU memory), regardless of how many other vGPUs are active. Among the three policies, Fixed Share provides the most predictable per-VM performance and the strongest isolation among VMs.

2.3 GPU Virtual Memory and Page Protection

Modern NVIDIA GPUs use virtual memory. GPU programs access memory using virtual addresses and the GPU Memory Management Unit (GMMU) translates the virtual addresses to physical addresses using page tables: each GPU context has its own virtual address

[1] Some recent GPUs, such as the Blackwell Pro 6000, support both vGPU and MIG. On these GPUs, when MIG is enabled, VMs can execute their GPU workloads in parallel. However, most GPUs that support vGPU do not support MIG, and thus rely on time-sliced execution. A more detailed discussion can be found in Section 7.1.

space and a page table that stores the address translations. The page tables for all active contexts reside in GPU memory and are managed by the GPU driver.

Similar to CPU page tables, GPU page tables contain not only address translation information but also permission bits for each page. Prior work [14, 38, 39] has shown that on modern NVIDIA GPUs, the page tables include a read-only bit for each page. However, this bit is not set in practice, even for code pages. As a result, all the pages in a GPU context's virtual address space are writable, including the code pages. In addition, NVIDIA GPU page tables include a privileged bit for each page. However, this bit is also never set in practice, meaning all pages in a context's address space can be accessed by unprivileged code [34].

3 Motivation and Overview

A critical concern in multi-tenant environments such as vGPU is performance interference between co-located VMs. This is especially problematic in public clouds, where providers typically charge users based on execution time.

NVIDIA vGPU scheduling policies govern how GPU time is allocated among co-located VMs, and in doing so, implicitly bound performance interference, providing Quality of Service (QoS) guarantees for VMs. In this work, we investigate whether these guarantees can be violated, i.e., whether one VM can cause performance degradation on co-located VMs beyond what the scheduling policies permit. In particular, we focus on uncovering and exploiting the vulnerabilities in the preemption mechanism. Preemption plays a critical role in vGPU scheduling: at the end of a time slice, the scheduler relies on preemption to switch GPU execution from one VM to another. A vulnerability in this process could allow a malicious VM to extend the preemption duration, occupying the GPU beyond its allocated time slice and starving co-located VMs.

4 Demystifying GPU Preemption

In this section, we examine how preemption works on NVIDIA GPUs through a series of experiments. Although our ultimate goal is to understand how preemption can be exploited to manipulate scheduling in vGPU environments, we study its behavior in a native (non-virtualized) environment in this section to simplify the analysis. In the next section, we show how our findings extend to vGPU scenarios.

Unless otherwise specified, all experiments in this section are conducted on a Linux system equipped with an NVIDIA H100 GPU, driver version 560.35, and CUDA toolkit 12.6.

4.1 The `EXIT` Instruction

We begin the analysis by studying the behavior of the `EXIT` instruction in the NVIDIA SASS ISA. This instruction is used to signal the completion of a GPU kernel's execution. We inspect the SASS code of a large set of GPU programs, including both pre-compiled libraries (e.g., cuDNN [31]) and open-source applications (e.g., Polybench [13]). The results show that `EXIT` always appears at the end of every kernel. In addition, this instruction always appears in its plain form (`EXIT`), never with a modifier (i.e., `EXIT.[MODIFIER]`).

However, as mentioned in Section 2.1, many SASS instructions support modifiers (e.g., `LD.128` specifies a 128-bit load). In fact, the `EXIT` instruction also supports modifiers. Specifically, certain strings in `nvdisasm` contain some description of SASS instructions [1]; by analyzing this description, we find that the `EXIT` instruction supports modifiers belonging to two independent groups:

- **Group 1:** { `KEEPREFCOUNT` | `PREEMPTED` }
- **Group 2:** { `NO_ATEXIT` }

At most one modifier from each group can appear in an `EXIT` instruction. As a result, there exist multiple variants of the `EXIT` instruction, such as `EXIT.KEEPREFCOUNT`, `EXIT.NO_ATEXIT`, and `EXIT.PREEMPTED.NO_ATEXIT`.

Although we do not see these variants in compiled SASS code, they might be used by the GPU driver or firmware at runtime to implement system-level operations. Motivated by this, we then inspect GPU memory during the execution of a GPU program to determine whether any of these variants appear at runtime. Specifically, we dump the entire GPU memory while a CUDA kernel is running, using the tool from prior work [50]. We find that most `EXIT` instructions in GPU memory are in their plain form without any modifiers. However, there is one exception: an `EXIT.PREEMPTED.NO_ATEXIT` instruction appears in memory during the kernel's execution.

In addition, we find that this `EXIT.PREEMPTED.NO_ATEXIT` instruction is located at the end of an instruction sequence consisting of 111 instructions in total. The full instruction sequence is provided in Appendix C. Based on several contextual clues, such as the instruction names and the special registers accessed, we believe this sequence implements *a trap handler* that handles certain system-level events (e.g., preemption). More specifically, this handler appears to consist of four phases, which we describe below:

- **Phase 1: State preservation.** The trap handler begins with a `BPT.DRAIN` instruction, which likely drains outstanding operations to ensure a consistent architectural state before the rest of the handler runs. It then saves a subset of registers and other execution state to local memory for use during the handler's execution.
- **Phase 2: Save location calculation.** In this phase, the handler dynamically calculates a unique memory address for each active warp based on the SM index and the warp index within the SM. This address is used as the base address for subsequent memory operations (e.g., storing warp state).
- **Phase 3: Subroutine dispatch.** In this phase, the handler checks specific bits in a status register (likely to determine the cause of the trap). Depending on which bits are set, it calls one or more specialized subroutines. We find that one of these subroutines saves the entire context, including all general-purpose registers (i.e., R0–R255), to memory.
- **Phase 4: Trap completion.** In this final phase, the handler restores the state saved in Phase 1 and then finishes with an `EXIT.PREEMPTED.NO_ATEXIT` instruction. We believe this instruction signals that the trap handler has completed execution and that the current context can be removed from the SMs.

4.2 Trap Handler Invocation Upon Preemption

Based on the analysis in Section 4.1, our hypothesis is that the trap handler ending with the `EXIT.PREEMPTED.NO_ATEXIT` instruction is invoked every time a context is preempted. As described in Section 2.1, when multiple contexts are active on a GPU, they share the

GPU in a time-sliced manner. At the end of a time slice, the GPU needs to preempt the running context to switch to another context. Therefore, to verify our hypothesis, we conduct the following experiment:

- **Step 1:** We first ensure the GPU is idle, and then launch a kernel that runs an infinite loop in one context (CTX_A). While the kernel is running, we use Memory-Mapped I/O (MMIO) to modify the first instruction of the trap handler in GPU memory to an invalid instruction (e.g., an instruction encoding with all bits set to 1).
- **Step 2:** We check whether this modification causes an error (e.g., `illegal instruction error`) in the running context (CTX_A).
- **Step 3:** If no error occurs in Step 2, we start another context (CTX_B), which forces the GPU to preempt CTX_A. We then check again whether the modified trap handler causes an error.

We find that when only one context (CTX_A) is running, modifying the trap handler does not cause any errors. However, after we launch the second context (CTX_B), an error occurs in CTX_A. This indicates that the modified trap handler is executed when CTX_A is preempted, which supports our hypothesis.

Takeaway 1. During the execution of a GPU program, a trap handler that ends with an `EXIT.PREEMPTED.NO_ATEXIT` instruction is placed into GPU memory. This trap handler is invoked when the running context is preempted.

4.3 Security Implications of the Trap Handler

To understand the security implications of the trap handler, we first examine whether it is shared across GPU contexts or private to each context. Specifically, we launch different numbers of contexts on the same GPU and inspect the GPU memory to locate the handler code. Our results show that each context has its own trap handler, rather than all contexts sharing a single handler.

In addition, by analyzing the page tables stored in GPU memory, we find that for each context, the trap handler is mapped into the context's virtual address space. For CUDA programs, when CUDA Address Space Layout Randomization (ASLR) is disabled, the trap handler's virtual address is fixed across runs of the same program and across different programs. In contrast, when CUDA ASLR is enabled, the trap handler's virtual address changes from run to run; however, it is always located in the same (2 MB) page as the user-provided code.

This observation raises significant security concerns: on NVIDIA GPUs, all pages in a context's virtual address space—including code pages—are writable, even by unprivileged code (cf. Section 2.3). This means that an unprivileged GPU program can directly modify the trap handler of its context through normal memory accesses. To verify this, we launch a kernel that writes `0xdeadbeef` to the trap handler address. We then use the memory dumping tool to examine the memory contents, and we find that the trap handler has been successfully modified without triggering any error.

Takeaway 2. Each GPU context has its own trap handler, which is mapped into its virtual address space. This handler can be modified through regular memory write operations from unprivileged GPU code.

Listing 1: The code structure of P_A.

```
__global__ void K_A1() {
    uint64_t trap_addr = get_trap_handler_addr();
    uint64_t exit_offset = find_exit_offset();
    write_inst(trap_addr + exit_offset, BRA_SELF_BYTECODE);
    write_inst(trap_addr + exit_offset + 0x10, EXIT_BYTECODE);
}

__global__ void K_A2() {
    uint64_t counter = 0;
    while (counter < ITER_COUNT) {
        counter+=1;}
}

int main() {
    launch(K_A1);
    cudaDeviceSynchronize();
    while (1) {
        start = get_time();
        launch(K_A2);
        cudaDeviceSynchronize();
        end = get_time();
        printf("Kernel runtime: %f ms\n", end - start);}
}
```

Takeaway 2 leads to a natural question: *when multiple users share a GPU, can one (malicious) user modify the trap handler to completely prevent its context from being preempted, and thus monopolize the GPU?* To answer this question, we design an experiment with two GPU programs (P_A and P_B) running in two different contexts.

The structure of P_A. As shown in Listing 1, P_A contains two kernels, K_{A1} and K_{A2}. P_A first launches K_{A1} to modify the trap handler by inserting an infinite loop in it. Specifically, it inserts a direct branch instruction (`BRA`) right before the `EXIT.PREEMPTED.NO_ATEXIT` instruction. The branch target is the branch instruction itself: whenever the trap handler is invoked and the execution reaches this branch instruction, it repeatedly jumps to itself. As a result, the trap handler (and the preemption process) never completes.

After K_{A1} finishes, P_A repeatedly launches K_{A2} which executes a counting loop. We configure the number of iterations in this loop to ensure that its execution time exceeds a GPU time slice, so that it gets preempted when another context is scheduled. In addition, after each execution of K_{A2}, P_A measures and prints its execution time to track its execution status during the experiment.

The structure of P_B. P_B contains only one kernel K_B, which is the same as K_{A2}. Similar to P_A, P_B repeatedly launches this kernel and prints its execution time in order to monitor the execution status during the experiment.

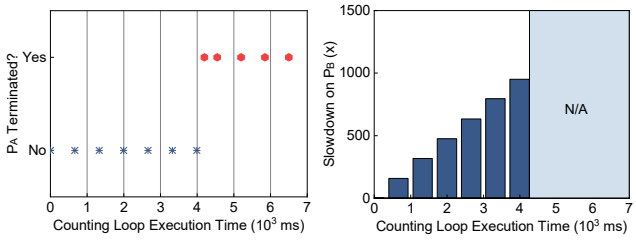
In the experiment, we first start P_A. After it finishes K_{A1} (i.e., finishes modifying the trap handler) and begins repeatedly launching K_{A2}, we start P_B on the same GPU.

Expected behavior. Under normal conditions, once we start P_B, the GPU scheduler should time-slice between P_A and P_B, i.e., the two programs repeatedly get preempted and resumed. But with the modified trap handler, when P_A is preempted, the trap handler spins, and the preemption process can never complete. As a result, P_B should stop making progress (e.g., stop printing the execution time) because it cannot be scheduled.

Observed behavior. However, when conducting the experiment, we observe behavior that is very different from the expected one. Specifically, once P_B starts running and P_A gets preempted, both P_A and P_B stop making progress for approximately 4000 ms. This is likely because during this 4000-ms period, P_A remains in the trap handler's infinite loop and P_B is therefore not scheduled. But after

this period, the context of P_A is corrupted and terminated[2] due to an error, and then P_B continues to make progress. The error message reported by the driver is `unspecified launch error`.

Based on this observation, we suspect the GPU system (driver or firmware) implements a timeout mechanism: if a context that is being preempted does not complete the preemption process within a certain time period (approximately 4000 ms), the system terminates this context.



(a) Whether P_A is terminated. (b) The performance degradation on P_B.

Figure 2: The results of the preemption timeout experiment.

Preemption timeout experiment. To further verify this hypothesis, we rerun the above experiment. However, this time, instead of letting K_{A1} insert an infinite loop in the trap handler, we let it insert a counting loop with a finite number of iterations (same as the counting loop in K_{A2}). Then every time P_A is preempted and the trap handler is invoked, it must finish running this loop before reaching the `EXIT.PREEMPTED.NO_ATEXIT` instruction and completing the preemption process. We vary the number of iterations in this loop to control its execution time (and therefore the total execution time of the trap handler).

Results. As shown in Figure 2(a), we find that when the execution time of the inserted loop is less than approximately 4000 ms, P_A never catches an error and gets terminated. However, once the loop execution time exceeds this threshold, P_A gets terminated with `unspecified launch error` about 4000 ms after P_B starts. This confirms our hypothesis that the GPU system enforces a preemption timeout of approximately 4000 ms.

Slowdown on P_B. From the printed kernel execution time in P_B, we observe that when P_A is not terminated (i.e., the loop in the trap handler is shorter than 4000 ms), the modified trap handler in P_A causes significant slowdown on P_B. This slowdown increases as the loop execution time increases. When the loop execution time reaches approximately 4000 ms (the upper bound), the kernel execution time of P_B becomes 951.4× compared to the case where P_A does not modify the trap handler, as shown in Figure 2(b).

This is because, with the loop inserted into the trap handler, every time P_A is scheduled, it spends extra time in the trap handler before yielding the GPU, as shown in Figure 3. Specifically, each time slice on NVIDIA GPUs is approximately 2 ms (cf. Section 2.1), and a regular preemption (without modification) takes around 0.2 ms. Then, in normal cases without trap handler modification, P_B gets to run for 2 ms in every 4.4 ms. Now with the modified trap handler in P_A, P_B only gets to run for 2 ms in every $(2 + t + 2 + 0.2)$ ms, where

[2]Note that strictly speaking, the context is not automatically terminated; instead, it enters a corrupted and unrecoverable state, and the only way to recover is to terminate the context and start a new one. For simplicity, we use the term "terminated" to describe this outcome throughout the paper.

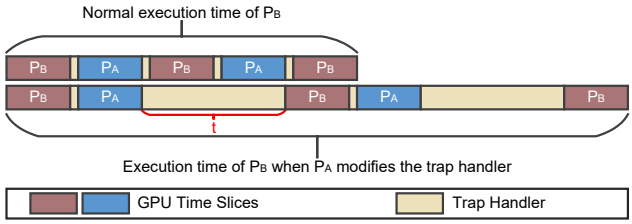


Figure 3: The time-sliced GPU execution before and after modifying the trap handler of P_A.

t is the execution time of the modified trap handler. When $t \approx 4000$ ms, this results in a slowdown of 910.0× in theory (measured as 951.4× in practice).

> **Takeaway 3.** NVIDIA uses a timeout-based preemption mechanism: if a preempted context does not complete the preemption process within a fixed time window (approximately 4000 ms), the GPU system terminates that context. This mechanism, together with Takeaway 2, can be exploited by one GPU user to slow down a co-located user.

Note that NVIDIA has patented a software-assisted instruction-level preemption mechanism [35]. This patent describes that when the instruction-level preemption is initiated, a "trap handler" containing a "preemption-save kernel" is invoked to save the context. This high-level description is consistent with our reverse-engineering findings in Section 4.1. However, the patent does not provide the detailed properties of the trap handler that we uncover, which lead to the security vulnerabilities.

4.4 A Preemption Manipulation Mechanism

Based on the analysis in Section 4.3, we propose a mechanism that a GPU user can exploit to manipulate the preemption process and significantly degrade the performance of users on the same GPU. Specifically, the user only needs to run a single program that first inserts a loop into the program's trap handler, and then continuously generates GPU workload (e.g., by running an infinite loop) to trigger preemption. Every preemption of this program will take much longer than normal to complete, and therefore this mechanism significantly reduces the effective GPU time available to other users and degrades their performance.

5 Manipulating vGPU Scheduling

Similar to native environments, vGPU also uses time slicing to share the GPU among VMs (cf. Section 2.2). Therefore, in this section, we investigate how the preemption manipulation mechanism proposed in Section 4 extends to vGPU environments—specifically, how one VM can obtain more GPU time than its allocated share and degrade the performance of co-located VMs.

Experimental platform. Unless otherwise specified, all the experiments in this section are conducted on a KVM-based virtualized system. Both the host and guest VMs run Ubuntu 24.04.4 LTS with Linux kernel version 6.8. For vGPU software (vGPU manager and guest driver), we use version 19.2, which was released in October 2025. The H100 GPU used in Section 4 does not support vGPU;

therefore, here we switch to three vGPU-supported GPUs, T4, A10, and L4. All three GPUs are used on one or more cloud platforms. Table 1 provides detailed specifications for them.

To simplify the discussion, in Section 5.1 to Section 5.4 we only discuss the findings on the T4 GPU. In Section 5.5, we discuss how our findings generalize to the A10 and L4 GPUs.

Setup and terminology. In all the experiments in this section, we assign one vGPU to each VM. Therefore, we use the terms "VM" and "vGPU" interchangeably when explaining the findings. This is also the most common setup in public clouds.

Table 1: An overview of the GPUs used in Section 5.

GPU	NVIDIA T4	NVIDIA A10	NVIDIA L4
Architecture	Turing	Ampere	Ada Lovelace
Memory Size	16GB	24GB	24GB
Virtualization	Software-mediated	Hardware-assisted	Hardware-assisted
Public Cloud Support	✓	✓	✓

5.1 Trap handler in vGPU

We first verify that the trap handler is still present in GPU memory when vGPU is used. Specifically, we modify the memory dumping tool to support vGPU and inspect GPU memory from the host while workloads run inside VMs. The results show that, similar to native environments, each context (in each VM) has its own trap handler. With this confirmed, the rest of Section 5 examines how our preemption manipulation mechanism behaves in vGPU environments. As mentioned in Section 2.2, unlike native environments that use a fixed scheduling policy, vGPU provides three policies: Best Effort, Equal Share, and Fixed Share. Thus, in the next few subsections, we study our mechanism under each of these policies. Since these policies have not been thoroughly characterized in any prior work, we also conduct experiments to verify that their behavior matches the documented specifications from NVIDIA.

5.2 Best Effort Policy

Best Effort is the default scheduling policy in vGPU. According to NVIDIA, under this policy, VMs are overall scheduled in a round-robin manner; however, a VM is skipped if it has no work to run (cf. Section 2.2). Thus, the GPU performance of a VM should depend on 1) the number of VMs sharing the GPU and 2) how heavily each VM uses the GPU. To verify this, we measure the execution time of a GPU program[3] in a VM under four different scenarios:

(1) No GPU workloads are running in other VMs; the measured execution time serves as the baseline.
(2) One other VM has GPU work for 50% of the time. For example, it launches a kernel that runs for 40 ms, then sleeps for 40 ms, and then launches the kernel again.
(3) One other VM has GPU work for 100% of the time. For example, it repeatedly launches kernels without sleeping.
(4) Two other VMs have GPU work for 100% of the time.

The results show that compared to the first scenario, the execution time in the second, third, and fourth scenarios increases by ≈50%, ≈100%, and ≈200%, respectively,[4] which confirms the documented behavior of the Best Effort scheduling policy.

Preemption timeout experiment (in vGPU). Then, to understand whether the preemption manipulation mechanism in Section 4 also works in vGPU under the Best Effort scheduling policy, we adapt the preemption timeout experiment from Section 4.3 to a vGPU setting. Specifically, we use two VMs (each with a vGPU), $\texttt{VM}_\texttt{A}$ and $\texttt{VM}_\texttt{B}$, and run programs $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{B}$ in $\texttt{VM}_\texttt{A}$ and $\texttt{VM}_\texttt{B}$, respectively. $\texttt{P}_\texttt{A}$ and $\texttt{P}_\texttt{B}$ are the same as those used in Section 4.3: $\texttt{P}_\texttt{A}$ first launches $\texttt{K}_{\texttt{A1}}$ to insert a counting loop into the trap handler, and then repeatedly launches $\texttt{K}_{\texttt{A2}}$ and measures its execution time. $\texttt{P}_\texttt{B}$ repeatedly launches $\texttt{K}_\texttt{B}$, which is identical to $\texttt{K}_{\texttt{A2}}$, and measures its execution time. We repeat this experiment with different numbers of iterations in the counting loop inserted into the trap handler, which correspond to different loop execution times.

Results. The results show that our preemption manipulation mechanism also works in vGPU under the Best Effort policy. Specifically, $\texttt{K}_{\texttt{A1}}$ can insert a loop into the trap handler with an execution time of up to 4000 ms without causing $\texttt{P}_\texttt{A}$ to be terminated; when the inserted loop exceeds this duration, it triggers the `unspecified launch error` and the context is terminated. This behavior is the same as what we observe in native environments.

In addition, the modified trap handler causes $\texttt{VM}_\texttt{A}$ to occupy the GPU much longer than normal and thus significantly degrades the performance of $\texttt{P}_\texttt{B}$ in $\texttt{VM}_\texttt{B}$. Specifically, the execution time recorded in $\texttt{P}_\texttt{B}$ reaches up to 1894.9× compared to the case where $\texttt{P}_\texttt{A}$ does not modify the trap handler. In other words, $\texttt{VM}_\texttt{A}$ can degrade the performance of $\texttt{VM}_\texttt{B}$ by up to 1894.9× beyond what is possible under normal operation of the Best Effort policy (which is when $\texttt{VM}_\texttt{A}$ fully utilizes the GPU without modifying the trap handler).

Note that 1894.9× is much larger than the maximum performance impact observed in native environments. This is because vGPU under the Best Effort policy uses a smaller time slice compared to native environments, which increases the frequency of preemption.

> **Takeaway 4.** The preemption manipulation mechanism remains effective in vGPU under the Best Effort policy. One VM can exploit it to slow down co-located VMs by up to 1894.9× beyond what is possible under normal operation of this policy.

5.3 Equal Share Policy

According to NVIDIA, the Equal Share policy divides GPU time *equally* among all active VMs. As a result, the performance of a GPU workload in one VM depends only on the number of active VMs, but not on how heavily the other VMs use the GPU. We empirically verify this by measuring the execution time of a GPU program in one VM while varying two factors: 1) the number of other active VMs and 2) the GPU utilization of those VMs.

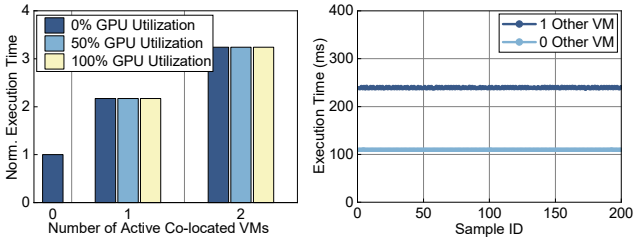
As shown in Figure 4(a), the *average* execution time scales approximately linearly with the number of co-located active VMs, regardless of how heavily those VMs use the GPU. This confirms that the policy ensures equal sharing of GPU time among all active VMs. In addition, Figure 4(b) shows the *raw* execution time samples

[3] We use a simple GPU program which executes a counting loop. We ensure the program's execution time exceeds one time slice in order to measure how the performance is affected by the scheduling policy.

[4] The actual increases are slightly above 50%, 100%, and 200% due to context switch overhead.

collected with zero and one co-located VM, each at varying GPU utilization levels. In both cases, the samples are highly stable, which further supports the documented behavior of Equal Share.

Preemption manipulation mechanism. To understand whether our preemption manipulation mechanism still works under this policy, we use the same preemption timeout experiment as the one used in Section 5.2: we start two VMs, VM_A and VM_B. P_A (with a modified trap handler) runs in VM_A, and P_B runs in VM_B.



(a) (b)

Figure 4: The empirical verification results of Equal Share's behavior: (a) the average execution time under different numbers of co-located VMs at different GPU utilization levels, normalized to the case with zero co-located VMs; (b) the raw execution time traces with zero and one co-located VM.

Results. We find that, similar to the case under the Best Effort policy, P_A can insert a loop into the trap handler with a duration of up to 4000 ms without causing its context to be terminated. However, this loop in the trap handler does not degrade the average performance of P_B.

Figure 5 shows the execution time trace of P_B when the duration of the loop inserted in P_A's trap handler is near 4000 ms. Before P_A starts (i.e., VM_A is active but idle), the execution time of P_B is stable at around 240 ms, which is consistent with the execution time shown in Figure 4(b). After P_A starts, periodic spikes appear in the execution time of P_B. These spikes are slightly above 4000 ms, which are likely caused by the delay in preempting P_A. However, aside from these spikes, most execution time samples after P_A starts are around only 110 ms, which is about the execution time of P_B when VM_B is the only active VM in the system (cf. Figure 4(b)). We find that this is because after each spike, VM_A is not scheduled for a period of time. As a result, the *average* execution time of P_B before and after P_A starts is roughly the same. In other words, modifying the trap handler in P_A cannot effectively degrade the performance of P_B on average.

Based on this observation, we believe that under the Equal Share policy, the vGPU scheduler tracks the cumulative GPU time used by each VM. Although the modified trap handler allows VM_A to occupy the GPU longer than normal, the vGPU scheduler is able to detect this and compensates for other VMs by not scheduling VM_A for a subsequent period. As a result, our original preemption manipulation mechanism cannot be directly used to reduce the average GPU time available to co-located VMs.

A limitation of the compensation mechanism. However, we find that this compensation mechanism has an important limitation: the vGPU manager does not compensate for the extra GPU time consumed by VM_A if P_A's context is terminated with an `unspecified launch error`, e.g., due to preemption timeout. Note that when

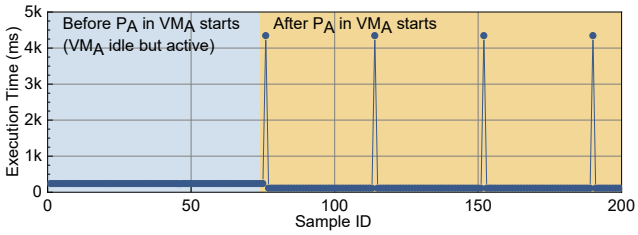


Figure 5: The execution time trace recorded in P_B when the preemption manipulation mechanism is applied in P_A.

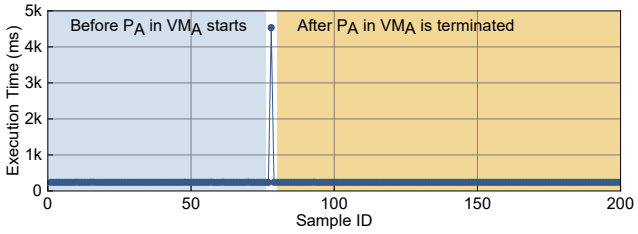


Figure 6: The execution time trace recorded in P_B when P_A triggers preemption timeout.

the context is terminated by the user (e.g., via `SIGINT`) or encounters other errors (e.g., `illegal instruction error`), the vGPU scheduler still applies compensation.

Specifically, Figure 6 shows the execution time trace of P_B when the loop inserted in the trap handler of P_A exceeds 4000 ms. When P_A starts running, P_B is unscheduled for approximately 4000 ms due to the delay in preempting P_A. This results in a large spike in the recorded execution time, similar to those observed in Figure 5. After this, P_A is terminated with an `unspecified launch error` (caused by preemption timeout). Then, unlike Figure 5, the execution time of P_B returns to the same as the baseline (i.e., before P_A started), rather than dropping below it. This indicates that the lost GPU time for VM_B is not compensated by the scheduler.

Adapted preemption manipulation mechanism. Leveraging the above observation, we design an adapted preemption manipulation mechanism that VM_A can use to occupy the GPU longer than normal and degrade the performance of VM_B. The key idea is to repeatedly launch a program that deliberately triggers preemption timeout (and gets terminated) every time it runs. Specifically, VM_A needs to run a program that first inserts a loop *longer than* 4000 ms into the trap handler, and then executes some GPU code to (eventually) trigger preemption. When preempted, the program remains in the trap handler for around 4000 ms and then catches an `unspecified launch error` (due to preemption timeout) and gets terminated. Each time after it is terminated, VM_A relaunches the program: each run of this program injects a 4000-ms delay to co-located VMs that is not compensated by the scheduler; repeatedly launching the program sustains this effect.

Figure 7 shows the execution time trace of P_B when we apply this mechanism in VM_A: the execution time of P_B fluctuates between 4533 ms and 17416 ms, and the average execution time is 9873 ms, which is 41.1× compared to the baseline without applying this mechanism (about 240 ms).

Note that this mechanism achieves a smaller degradation magnitude than the original preemption manipulation mechanism (for

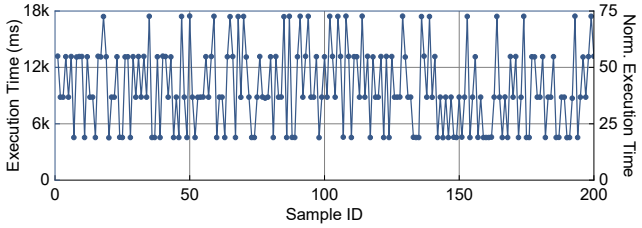


Figure 7: The execution time trace recorded in P_B when the adapted preemption mechanism is applied in P_A.

Best Effort) due to the high overhead of starting and terminating a GPU context: starting and terminating a context takes around 100 ms which is much longer than a time slice (≈2 ms). As a result, this mechanism cannot delay *every* preemption on VM_A. In Figure 7, we have already applied pipelining techniques by pre-launching multiple contexts to address this issue; however, the overhead remains significant compared to the time slice duration.

Takeaway 5. Under the Equal Share policy, one VM can exploit the adapted preemption manipulation mechanism to bypass compensation and degrade the performance of co-located VMs by 41.1×. This fundamentally violates the QoS guarantee that this policy is designed to provide.

5.4 Fixed Share Policy

The Fixed Share policy allocates a fixed portion of GPU time to each VM. The exact portion is determined by the vGPU type, and is unrelated to how many other VMs are active or how heavily they use the GPU. For example, on a T4 GPU, a VM with a T4-4Q vGPU (which has 1/4 of the GPU memory) receives approximately 1/4 of the GPU cycles. We have empirically verified this behavior on our system: the performance of a GPU program in one VM remains unchanged when other VMs start, stop, or vary their GPU utilization.

Preemption manipulation mechanism. We repeat all experiments that are used to evaluate the two preemption manipulation mechanisms under Equal Share, now under Fixed Share, and find that the results are very similar. Specifically, the original preemption manipulation mechanism is not effective, as the scheduler tracks cumulative GPU time used by each VM and compensates when one VM uses more than its allocated share. However, the adapted preemption manipulation mechanism remains effective: by deliberately triggering the preemption timeout, one VM can bypass this compensation and consistently degrade the performance of co-located VMs.

The exact degradation magnitude depends on the vGPU type. This is because, as discussed above, different types of vGPUs receive different fractions of GPU time and thus have different baseline performance. Overall, the performance degradation remains at a similar order of magnitude as under Equal Share. For example, with T4-8Q, the co-located VMs experience an average slowdown of 41.6×, and with T4-4Q, the slowdown is approximately 36.3×, both compared to the case where the manipulation mechanism is not applied. We provide a more detailed analysis and evaluation on how the vGPU type affects the degradation magnitude in Section 6.3.

Takeaway 6. The adapted preemption manipulation mechanism remains effective under the Fixed Share policy, despite its strong isolation properties. The mechanism produces a similar degree of performance degradation as under Equal Share.

5.5 Results on Other GPUs

We also conduct all the analyses from Section 5.1 to Section 5.4 on NVIDIA A10 and L4 GPUs (cf. Table 1), using the same system configuration as before. We find that our conclusions regarding the preemption manipulation mechanisms generalize to both GPUs. Specifically, under Best Effort, one VM can use the original preemption manipulation mechanism to degrade the performance of co-located VMs. Under Equal Share and Fixed Share, the adapted mechanism must be used instead.

However, the performance degradation magnitudes on these two GPUs differ from those on T4: the preemption timeout threshold on A10 and L4 is the same, which is much smaller than the one on T4. This directly limits how long the modified trap handler can occupy the GPU per preemption. As a result, on these two GPUs, one VM can degrade the performance of a co-located VM by up to 429.3× under Best Effort, and 6.2× under Equal Share, compared to the case where the trap handler is not modified. As before, the performance impact under Fixed Share depends on the vGPU type but remains in the same order of magnitude as under Equal Share. Note that although the magnitudes of the performance degradation on A10 and L4 are smaller than those on T4, they still represent fundamental violations of the scheduling policies.

We believe this difference between T4 and A10/L4 is caused by the underlying GPU virtualization mechanism. NVIDIA implements vGPU differently depending on the GPU generation. Prior to the Ampere generation (e.g., T4), vGPU relies on software-mediated virtualization using the mediated device (mdev) framework. Starting with Ampere (e.g., A10 and L4), vGPU uses hardware-assisted virtualization via Single Root I/O Virtualization (SR-IOV). These fundamentally different virtualization approaches likely affect how GPU resources are scheduled and managed, which leads NVIDIA to adopt different timeout thresholds even under the same vGPU software version. Notably, in native environments, the preemption timeout threshold on both A10 and L4 remains the same as the one on T4, which suggests that the difference is specific to vGPU.

6 A Performance Degradation Attack on vGPU

Based on the analysis in Section 5, we develop a cross-VM performance degradation attack targeting vGPU. In this section, we first describe the threat model, then present the attack, and finally evaluate it with real-world GPU applications in both local and public cloud environments.

6.1 Threat Model

We assume a setting where the attacker and the victim run in two separate VMs. Each VM is assigned a vGPU, and the two vGPUs are backed by the same physical NVIDIA GPU. The attacker's goal is to manipulate the preemption process and cause significant performance degradation on the victim's GPU workload.

Table 2: GPU workloads used in the evaluation.

Category	Workloads
Blender	Classroom, Barcelona Pavilion, Repeat Zone-Flower, Chocolate Donut, Nishita Sky
CNN Inference	ResNet18, ResNet50, DenseNet121, VGG16, AlexNet
Transformer Inference	BART-Base, T5-Base, T5-Large, Qwen3-0.6B, DistilGPT2

We assume that the attacker can run arbitrary GPU programs within its VM. In addition, the attacker knows the model of the underlying physical GPU (e.g., T4 or A10). This information can be obtained by simply querying the guest vGPU driver within the attacker's VM, and is also typically disclosed by cloud providers when provisioning VM instances.

6.2 Attack Overview

The key idea behind the attack is to extend the preemption duration using the preemption manipulation mechanisms from Section 5, thus reducing the effective GPU time available to the victim. Specifically, the attack consists of three steps, which we explain below.

Step 1: Preparing the loop. The attacker first prepares the machine code of a counting loop that can execute on the target GPU. The execution time of the loop should be either slightly below the preemption timeout threshold or above it, depending on which mechanism is used in Step 3.

Step 2: Locating the trap handler. Next, the attacker identifies the virtual address of the trap handler. As explained in Section 4, the trap handler resides in the same virtual page as the user-provided code. Therefore, the attacker can first obtain the value of a function pointer in its program and then search within the corresponding page to locate the trap handler.

Step 3: Launching the attack program. Finally, the attacker launches a program that modifies the trap handler with the loop from Step 1 and triggers delayed preemption. The exact strategy depends on the scheduling policy:

- **Best Effort.** The attacker uses the original preemption manipulation mechanism: it launches a long-running GPU program that first inserts a loop—with an execution time slightly below the preemption timeout threshold—into the trap handler, and then executes an infinite loop (or any GPU workload) to continuously trigger delayed preemption.
- **Equal Share and Fixed Share.** The attacker uses the adapted preemption manipulation mechanism: it *repeatedly* launches a short-lived GPU program that first inserts a loop—with an execution time above the preemption timeout threshold—into the trap handler, and then executes an infinite loop (or any GPU workload) to trigger delayed preemption. Each run ends with the program being terminated due to preemption timeout.

To select the attack strategy in Step 3, the attacker needs to first identify the vGPU scheduling policy in use, which can be done by profiling context switch latency following the approach in prior work [17]. Alternatively, the attacker can always use the adapted mechanism designed for Equal Share and Fixed Share, which also works under Best Effort (with a smaller magnitude).

Table 3: Major public cloud providers offering vGPU instances based on NVIDIA T4 and/or A10 GPUs.

GPU	Azure	Alibaba Cloud	Tencent Cloud
NVIDIA T4	✗	✓	✓
NVIDIA A10	✓	✓	✓

6.3 Local Evaluation

In this section, we evaluate the performance impact of the attack on GPU workloads in local vGPU environments. Specifically, we use workloads from Blender [7], a widely used 3D rendering application, as well as representative convolutional neural network (CNN) and large language model (LLM) inference workloads. The full list is shown in Table 2.

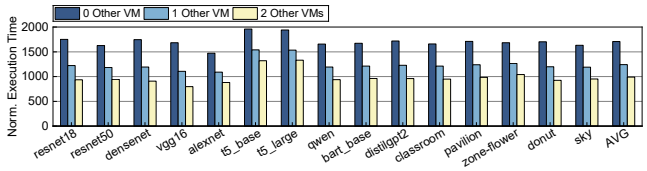
We use two vGPU-supported GPUs, T4 and A10, in our evaluation, since they use different virtualization frameworks and therefore produce very different attack results (cf. Section 5.5). In addition, T4 and A10 are among the most widely deployed GPUs for vGPU services in public clouds. For example, Table 3 lists major cloud providers that offer vGPU instances based on these two GPUs. The system setup of the evaluation is the same as the one used in Section 5.

Under Best Effort, the *baseline* performance of the victim is affected by the number of additional active VMs (beyond the attacker and the victim) and their GPU utilization, which in turn affect the degradation magnitude of the attack. Specifically, more additional VMs and higher GPU utilization in those VMs result in a proportionally larger baseline execution time for the victim; however, the victim's execution time under the attack is dominated by the inserted preemption delay and remains largely unchanged regardless of these two factors. Therefore, the degradation magnitude of the attack decreases with more additional VMs and higher GPU utilization in those VMs. Similarly, under Equal Share, the number of additional active VMs (but not their GPU utilization) affects the attack magnitude.

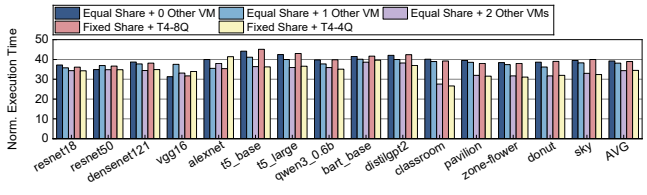
Thus, for these two policies, we evaluate the attack with different numbers of additional VMs. For Best Effort, we configure the additional VMs with 100% GPU utilization to show the lower bound of the attack's impact. Specifically, on the T4 GPU, we use T4-4Q instances where each VM receives 1/4 of the GPU resources, allowing at most two additional VMs alongside the attacker and the victim. On the A10 GPU, we use A10-8Q instances where each VM receives 1/3 of the GPU resources, allowing at most one additional VM. Note that T4-4Q and A10-8Q are among the most commonly used instance types in public clouds.

In addition, under Fixed Share, the vGPU type (rather than the number of active VMs) affects the baseline performance of the victim, and therefore the magnitude of the attack. Thus, we evaluate the attack with different vGPU types. Specifically, on the T4 GPU we use T4-4Q and T4-8Q, and on the A10 GPU we use A10-8Q and A10-12Q. Again, these are all among the most commonly used vGPU types in public cloud environments.

Results. Figure 8(a) and Figure 9(a) show the execution time of workloads in the victim VM under the attack with the Best Effort policy. All the results are normalized to the baseline without the attack. On the T4 GPU, the attack causes an average performance

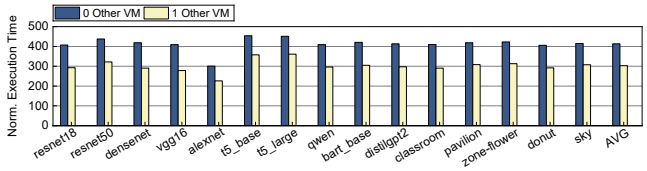


(a) Best Effort

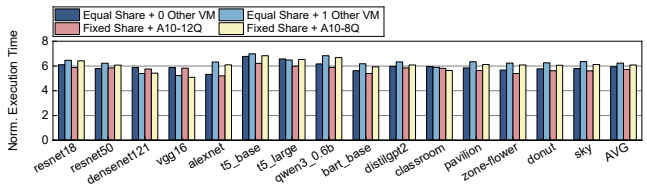


(b) Equal Share and Fixed Share

Figure 8: The normalized execution time of GPU applications under the performance degradation attack on the T4 GPU.



(a) Best Effort



(b) Equal Share and Fixed Share

Figure 9: The normalized execution time of GPU applications under the performance degradation attack on the A10 GPU.

degradation of 1707.2×, 1240.1×, and 987.9× with 0, 1, and 2 additional VMs respectively. On the A10 GPU, the average degradation is 412.9× and 302.5× with 0 and 1 additional VM respectively. As explained above, the performance degradation decreases with more additional VMs.

Figure 8(b) and Figure 9(b) show the normalized execution time of the victim VM under the attack with the Equal Share and Fixed Share policies. On the T4 GPU, under Equal Share, the average degradation is 39.2×, 38.1×, and 34.4× with 0, 1, and 2 additional VMs respectively. On the A10 GPU, the average degradation is 5.9× and 6.2× with 0 and 1 additional VMs respectively. Unlike under Best Effort, the degradation does not decrease significantly with more active VMs; on A10 it even slightly increases. This is because the adapted preemption manipulation mechanism cannot inject a delay into every preemption of the attacker due to the high overhead of launching/terminating a context, as explained in Section 5.3. With more co-located VMs, the attacker is preempted less frequently, giving more time for the context to be relaunched before the next preemption opportunity, which increases the chance of successfully injecting a delay. As a result, the two effects partially offset each other: more VMs raise the baseline but also increase the chance of injecting delays, which keeps the degradation magnitude relatively stable.

Likewise, under Fixed Share, the performance degradation magnitude remains similar across the tested vGPU types: the average degradation is 38.9× and 34.5× with T4-8Q and T4-4Q respectively, and 5.7× and 6.1× with A10-12Q and A10-8Q respectively.

The observed slowdown slightly varies across workloads. We believe there are several factors that affect the end-to-end slowdown caused by the attacker, and the observed slowdown is an overall reflection of all of them. One critical factor is the ratio of GPU-active time to total execution time of the victim workload: our attack only affects the GPU-active portion of execution. Workloads that spend a larger fraction of their time on the CPU side (e.g., on kernel launches, synchronization, and memory copy operations) may experience a smaller overall slowdown, since the attack does not affect CPU-side execution. This ratio also affects how the slowdown changes with the number of co-located VMs: when more VMs share the GPU, the victim waits longer between its time slices, but can still perform CPU-side work during the wait. In addition, there is a tail effect: after the final time slice of the victim workload completes, no subsequent delay is injected. For workloads with shorter end-to-end execution time, the execution spans fewer time slices, so this undelayed final slice has a proportionally larger impact on the overall slowdown. For example, AlexNet has the shortest execution time among the evaluated workloads, which makes the tail effect's impact on the overall slowdown one of the largest.

6.4 Evaluation on Public Cloud Platforms

We evaluate our attack on public cloud platforms to demonstrate its real-world impact. Due to ethical considerations, we only evaluate our attack when we are able to occupy all the vGPU instances on a physical GPU to ensure that no other user is sharing the GPU with us. For example, if a physical GPU supports four vGPU instances (four VMs), we only run the evaluation if we are able to occupy all four. This is necessary because evaluating the attack on a GPU shared with other users would directly impact their performance and lead to financial loss.

As explained in Section 2.2, many cloud providers now offer vGPU instances, such as Alibaba Cloud, Azure, Vultr, AWS, Tencent Cloud, and CloudFerro. We successfully achieve this condition—occupying all the vGPU instances on a physical GPU—in some regions of Alibaba Cloud, Azure, and Vultr, and then evaluate on these three platforms. Note that similar to Section 6.3, we use the workloads in Table 2 to evaluate our attack.

Evaluation on Alibaba Cloud. On Alibaba Cloud, we evaluate our attack with both T4 and A10 GPUs. For T4, we rent the `vgn6i-m4-vws.xlarge` instances; each of them is equipped with a T4-4Q vGPU (1/4 T4). For A10, we rent `sgn7i-vws-m8s.4xlarge` instances; each of them is equipped with an A10-8Q vGPU (1/3 A10). We find that Alibaba Cloud uses the Fixed Share policy on both T4 and A10 GPUs. On T4 GPUs, our attack causes an average performance degradation of 111.3×, which is much higher than the degradation measured in our local experiments. We find that this is because Alibaba Cloud uses an older version of the vGPU software that handles preemption very differently from more recent

versions; the details are provided in Appendix D. In addition, on A10 GPUs, our attack causes an average degradation of 6.4×, which is consistent with our local results.

Evaluation on Azure. On Azure, we evaluate our attack on A10 GPUs.[5] We rent `NV12ads_A10_v5` instances; each of these instances is equipped with an A10-8Q vGPU (1/3 A10). We find that Azure also uses the Fixed Share policy. Our attack causes an average degradation of 4.0×, which is lower than the degradation measured in our local environments. To investigate this discrepancy, we locally set up the same vGPU software version that we used on Azure (vGPU 18.6) and re-run the experiments; the results show that the degradation under this version is similar to that in our earlier local experiments (≈6.1×). Therefore, we believe the difference is not caused by the vGPU software version but rather some other factors specific to Azure (e.g., their specialized hypervisor).

Evaluation on Vultr. Vultr supports vGPU with A40 and A16 GPUs. We rent the `vcg-a40-6c-30g-12vram` instances; each of these instances is equipped with an A40-4Q vGPU (1/4 A40). We find that unlike Alibaba Cloud and Azure, Vultr uses the default Best Effort policy. As a result, our attack achieves a much higher performance degradation than on the other two platforms: on average 436.4×, 317.3×, and 256.7×, compared to the baseline with 0, 1, or 2 other VMs running at 100% GPU utilization, respectively. These results are also consistent with those measured in the local environments.

7 Discussion

7.1 Applicability

In this section, we discuss the applicability of our performance degradation attack across different execution environments on NVIDIA GPUs. As discussed throughout this paper, vGPU is a multi-tenancy technique focusing on virtualizing the GPU hardware. Another multi-tenancy technique is MIG [32], which spatially partitions the physical GPU into hardware-isolated instances. These two techniques are complementary (with different purposes). Most GPUs support at most one of the two techniques; only a few GPUs support both.

Native environment. In native GPU environments without virtualization/MIG, GPU contexts share the physical GPU through time slicing by default (cf. Section 2.1). Our attack is effective in this environment: a malicious user can modify its trap handler to extend its preemption duration and occupy the GPU far beyond a normal time slice.

vGPU without MIG. vGPU is supported on a wide range of NVIDIA GPUs, spanning from Turing and Ampere to Ada Lovelace and Blackwell. The full list of supported GPUs can be found in [33]. Among all the vGPU-capable GPUs, only a few GPUs also support MIG. On all other vGPU-capable GPUs, MIG is not supported, and VMs must time-share the physical GPU under one of the three vGPU scheduling policies. The same applies to those MIG-capable GPUs when MIG is not enabled. Our attack, as demonstrated in Section 5, is effective under all three policies.

[5] Azure currently only supports renting partial GPUs for A10; for other NVIDIA GPUs, users must rent at least one entire GPU.

vGPU with MIG. On GPUs that support both vGPU and MIG, the physical GPU can be divided into several MIG instances when MIG is enabled. Then with vGPU, each VM can use a fraction of a MIG instance. VMs within the same MIG instance time-share the GPU hardware, while VMs in different instances run in parallel and are hardware-isolated. We expect our attack to remain effective between VMs that run in the same MIG instance, since they still rely on the preemption mechanism to enforce time slicing. However, the attack may not apply across VMs in different MIG instances, as they do not time-share the hardware.

MIG-only (without vGPU). Some compute-focused GPUs, such as A30, A100, and H100, support MIG but do not support vGPU. On these GPUs, when different users run in separate MIG instances without time-sharing, our attack does not apply.

7.2 Countermeasures

Attack detection. We propose a method to detect our attack for cloud providers that use the Equal Share or Fixed Share policy. Specifically, the hypervisor can obtain the total GPU time consumed by each VM in each time slice, including time spent in preemption, through the vGPU manager (using the `nvidia-smi vgpu -sl` command). When the attack is launched, the GPU time consumed by the attacker VM is significantly higher than normal, which can serve as a detection signal. Note that the vGPU manager does not provide per-VM timing information under the Best Effort policy; therefore, this detection method is not applicable to cloud platforms using Best Effort (such as Vultr). However, we strongly recommend all cloud providers to use Equal Share or Fixed Share, since Best Effort provides the weakest QoS guarantee for users and our attack results in significantly higher performance degradation under Best Effort than the other two policies.

Preventing trap handler modification. A more fundamental defense mechanism is to prevent users from modifying the trap handler entirely. Since NVIDIA page tables have a read-only bit for each page, and the trap handler page appears to hold only code and no data, the vGPU manager should set this bit for the trap handler page, preventing any modification from user programs. However, unlike the native NVIDIA drivers which are now open-source, the vGPU managers (and guest drivers) are still closed-source; therefore, implementing this defense mechanism requires changes from NVIDIA.

7.3 Generality Across vGPU Software Versions

We test our attack with different recent versions of vGPU software, including 18.5 to 18.6 and 19.0 to 19.4. The results show that our attack remains effective with all the tested versions, and the degradation magnitudes are similar to those shown in Section 6.3. In addition, since some cloud platforms use older versions of vGPU software, we also test the versions used or recommended by Alibaba Cloud, Azure, Vultr, AWS, Tencent Cloud, and CloudFerro. For platforms that use/recommend multiple versions, we test the latest one. The results show that our attack remains effective with all these versions. With most versions, the degradation magnitudes are similar to those shown in Section 6.3. The only exception is vGPU 13.11 used by Alibaba Cloud, which shows a higher magnitude of degradation on T4 GPUs (as discussed in Section 6.4).

Table 4: Comparison with prior work exploiting time slicing on GPUs.

Attack	Attack Goal	Validated in vGPU	Manipulate Scheduling
Cache Attack [12, 51]	Confidentiality	✗	✗
TLB Attack [17, 26]	Confidentiality	✔	✗
Timeslice-Sandwich [18]	Confidentiality	✗	✗
Leaky DNN [48]	Confidentiality	✗	✔
This Work	Performance	✔	✔

7.4 Related Work

GPU memory management vulnerabilities. Several prior studies have analyzed the vulnerabilities in GPU memory management. First, Park *et al.* showed that NVIDIA GPUs allow users to modify code pages [38]. Based on this property, they developed the *Mind Control* attack, which corrupts the code of deep learning systems to significantly degrade the prediction accuracy. Guo *et al.* extended this line of work to more recent NVIDIA GPUs and demonstrated that, in addition to code pages being writable, data pages are executable [14]. They also analyzed the stack layout on NVIDIA GPUs and demonstrated the feasibility of return-oriented programming (ROP) attacks. More recently, Roels *et al.* further analyzed the ROP attacks and showed that the available gadget set is Turing-complete. They also identified several weaknesses in existing mitigation mechanisms for GPU memory corruption, such as CUDA ASLR [39].

Prior work exploiting time slicing on GPUs. A few recent studies have analyzed security attacks exploiting GPU time slicing, as shown in Table 4. These studies mainly focus on passively monitoring side effects that span across GPU time slices, in order to infer private information (i.e., side-channel attacks). In contrast, our work focuses on actively manipulating the preemption mechanism to degrade the victim's performance—a different attack goal that exploits a different vulnerability.

First, Zhang *et al.* developed side-channel attacks that exploit the GPU L2 cache [51]. Giner *et al.* demonstrated that GPU cache side-channel attacks can be launched from web browsers via the WebGPU API [12]. These attacks leak private information from a victim running on the same GPU. Neither of these specific attack primitives has been empirically validated in vGPU environments, although they should remain effective. Nayak *et al.* developed a covert channel exploiting the GPU translation lookaside buffer (TLB) [26]. Jin *et al.* further analyzed this problem and developed a TLB side-channel attack, which has been validated in vGPU environments [17].

In addition, Kim *et al.* analyzed GPU scheduling in native environments and demonstrated a side-channel attack that exploits the scheduling delay to leak private information from a co-located GPU program. This attack has not been demonstrated in vGPU environments, but it may remain effective under Best Effort. Note that none of the prior attacks discussed so far can manipulate the GPU scheduler as our attack does.

Moreover, Wei *et al.* developed an attack that uses CUPTI (i.e., NVIDIA's performance counters) to leak the hyper-parameters of a DNN model from a training workload [48]. However, CUPTI is not available to VMs in vGPU environments (empirically verified locally and on Azure), which makes this attack impractical on virtualized cloud platforms. In addition, this prior work observed that the attacker can reduce the victim's time-slice length by changing the attacker's GPU kernel parameters, which can be used to slow down the victim and improve the side-channel attack granularity. However, we found that this method does not work in vGPU.

Other GPU security problems. Beyond the above, researchers have also studied side-channel attacks targeting other GPU features/components, including the CPU–GPU interconnect [22, 42], hardware partitioning [50], Multi-Process Service (MPS) [22, 25–27], and software interfaces [25]. There are also prior studies on other GPU security issues such as memory safety vulnerabilities [10, 11, 14, 24] and rowhammer attacks [15, 20, 21, 47].

CPU scheduling attacks. Several studies have investigated attacks that manipulate the CPU scheduler, e.g., in Linux [44] and Xen [2, 40].

8 Conclusion

In this paper, we reverse-engineered the implementation of the preemption mechanism on NVIDIA GPUs and uncovered a severe security vulnerability. Leveraging this vulnerability, we developed a preemption manipulation mechanism that can extend the duration of preemption far beyond normal. We studied this mechanism in vGPU environments and showed that it, or its adapted version, can be used by one VM to significantly reduce the effective GPU time available to co-located VMs, violating the QoS guarantees the vGPU scheduling policies are designed to provide. Based on this, we proposed a performance degradation attack targeting vGPU environments and demonstrated that it is effective on both local machines and public cloud platforms.

9 Acknowledgment

We sincerely thank the anonymous CCS 2026 reviewers for their constructive feedback. This work was partially supported by the U.S. National Science Foundation (NSF) under Grants #2530649 and #2443671. This work was also partially supported by the Advanced Research Projects Agency for Health (ARPA-H) under Agreement No. 140D042590046, the Alfred P. Sloan Research Fellowship, a grant from Coefficient Giving, and gifts from Qualcomm and Zama.

References

[1] 0xD0GF00D. 2026. DocumentSASS. Available at https://github.com/0xD0GF00D/DocumentSASS.
[2] Suaad Alarifi and Stephen D Wolthusen. 2013. Robust coordination of cloud-internal denial of service attacks. In *2013 International Conference on Cloud and Green Computing*.
[3] Alibaba. 2026. Elastic GPU Service instance families (gn, vgn, and sgn series). Available at https://www.alibabacloud.com/help/en/ecs/user-guide/gpu-accelerated-compute-optimized-and-vgpu-accelerated-instance-families-1.
[4] Amazon. 2026. Amazon EC2 G6 Instances. Available at https://aws.amazon.com/ec2/instance-types/g6/.
[5] AWS. 2026. Recommended GPU Instances. Available at https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html.
[6] Azure. 2026. Virtual Machine series. Available at https://azure.microsoft.com/en-us/pricing/details/virtual-machines/series/.
[7] Blender Online Community. 2026. Blender. Available at https://www.blender.org.
[8] CloudFerro. 2026. GPU Services. Available at https://cloudferro.com/cloud/public-cloud/compute/gpu/.

[9] CloudFerro. 2026. Virtual Machines (VM) with GPU. Available at https://cloudferro.com/pricing/pricing-tables/waw4-1-cloud/virtual-machines-vm-with-gpu/.
[10] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient Buffer Overflow Detection on GPU. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021).
[11] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A Dynamic GPU Memory Overflow Detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*.
[12] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. 2024. Generic and automated drive-by GPU cache attacks from the browser. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*.
[13] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*.
[14] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security)*.
[15] Yichang Hu, Noah Brown, Yuhang Chen, Joshua Bakita, Tianlong Chen, Daniel Genkin, and Andrew Kwong. 2026. GDDRHammer: Greatly Disturbing DRAM Rows—Cross-Component Rowhammer Attacks from Modern GPUs. In *47th IEEE Symposium on Security and Privacy (S&P)*.
[16] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[17] Hongyue Jin, Yanan Guo, and Zhenkai Zhang. 2026. Exploiting TLBs in Virtualized GPUs for Cross-VM Side-Channel Attacks. In *2026 Network and Distributed System Security Symposium (NDSS)*.
[18] Hodong Kim, Gyeongsup Lim, Seunghee Shin, Youngjoo Shin, and Junbeom Hur. 2026. TIMESLICE-SANDWICH: A GPU Side-Channel Attack Exploiting Time-Sliced Scheduling. In *35th USENIX Security Symposium (USENIX Security)*.
[19] Tao Liao, Yongjie Zhang, Peter M Kekenes-Huskey, Yuhui Cheng, Anushka Michailova, Andrew D McCulloch, Michael Holst, and J Andrew McCammon. 2013. Multi-core CPU or GPU-accelerated multiscale modeling for biomolecular complexes. *Computational and Mathematical Biophysics* 1, 2013 (2013), 164–179.
[20] Chris S Lin, Joyce Qu, and Gururaj Saileshwar. 2025. GPUHammer: Rowhammer attacks on GPU memories are practical. In *34th USENIX Security Symposium (USENIX Security)*.
[21] Chris S Lin, Yuqin Yan, Guozhen Ding, Joyce Qu, Joseph Zhu, David Lie, and Gururaj Saileshwar. 2026. GPUBreach: Privilege escalation attacks on GPUs using Rowhammer. In *47th IEEE Symposium on Security and Privacy (S&P)*.
[22] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. 2024. Veiled Pathways: Investigating covert and side channels within GPU uncore. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[23] Microsoft. 2026. NVadsA10_v5 sizes series. Available at https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nvadsa10v5-series?tabs=sizebasic.
[24] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12 (2016), 113–120.
[25] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security (CCS)*.
[26] Ajay Nayak, Pratheek B, Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis) managed: A novel TLB-based covert channel on GPUs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*.
[27] Ravan Nazaraliyev, Yicheng Zhang, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. 2025. Not so Refreshing: Attacking GPUs using RFM rowhammer mitigation. In *34th USENIX Security Symposium (USENIX Security)*.
[28] NVIDIA. 2026. CUDA Binary Utilities. Available at https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.
[29] NVIDIA. 2026. CUDA C++ Programming Guide. Available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[30] NVIDIA. 2026. NVIDIA AI Enterprise Deployment Guide. Available at https://docs.nvidia.com/ai-enterprise/deployment/vmware/latest/advance-gpu.html#terminology.
[31] NVIDIA. 2026. NVIDIA cuDNN. Available at https://developer.nvidia.com/cudnn.
[32] NVIDIA. 2026. NVIDIA Multi-Instance GPU. Available at https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.
[33] NVIDIA. 2026. NVIDIA® Virtual GPU Software Supported GPUs. Available at https://docs.nvidia.com/vgpu/gpus-supported-by-vgpu.html.
[34] NVIDIA. 2026. open-gpu-kernel-modules. Available at https://github.com/NVIDIA/open-gpu-kernel-modules/blob/main/kernel-open/nvidia-uvm/hwref/hopper/gh100/dev_mmu.h.
[35] NVIDIA. 2026. Software-Assisted Instruction Level, Execution Preemption. Available at https://patentimages.storage.googleapis.com/56/18/16/5cda49b4381cf3/US9652282.pdf.
[36] NVIDIA. 2026. Virtual GPU Software User Guide. Available at https://docs.nvidia.com/vgpu/latest/grid-vgpu-user-guide/index.html.
[37] NVIDIA. 2026. Virtual GPU Software User Guide. Available at https://docs.nvidia.com/knowledge-base/latest/vgpu-features.html.
[38] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* 102 (2021), 102115.
[39] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. Cuda, woulda, shoulda: Returning exploits in a sass-y world. In *Proceedings of the 18th European Workshop on Systems Security*.
[40] Hong Rong, Ming Xian, Huimei Wang, and Jiangyong Shi. 2013. Time-Stealer: A Stealthy Threat for Virtualization Scheduler and Its Countermeasures. In *International Conference on Information, Communications and Signal Processing*.
[41] Dave Shreiner and The Khronos OpenGL ARB Working Group. 2009. *OpenGL Programming Guide: The official guide to learning OpenGL, Versions 3.0 and 3.1* (7th ed.). Addison-Wesley Professional.
[42] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. 2021. Invisible Probe: Timing attacks with PCIe congestion side-channel. In *2021 IEEE Symposium on Security and Privacy (SP)*.
[43] Tencent. 2026. Cloud GPU Service. Available at https://www.tencentcloud.com/document/product/560/75945.
[44] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Secretly Monopolizing the CPU Without Superuser Privileges. In *USENIX Security Symposium*.
[45] VULTR. 2026. High-Performance Cloud GPU Solutions. Available at https://www.vultr.com/products/cloud-gpu/.
[46] Vultr. 2026. How to Manage vGPU on Vultr Cloud GPU Instances. Available at https://docs.vultr.com/how-to-manage-vgpu-on-vultr-cloud-gpu-instances.
[47] Junpeng Wan, Yanan Guo, Zhi Zhang, Zhuo Li, Dave Jing Tian, and Zhenkai Zhang. 2026. Geforge: Hammering GDDR memory to forge GPU page tables for fun and profit. In *47th IEEE Symposium on Security and Privacy (S&P)*.
[48] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. 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)*.
[49] Anuradha Welivita, Indika Perera, Dulani Meedeniya, Anuradha Wickramarachchi, and Vijini Mallawaarachchi. 2018. Managing complex workflows in bioinformatics: An interactive toolkit with GPU acceleration. *IEEE transactions on nanobioscience* 17, 3 (2018), 199–208.
[50] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. 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)*.
[51] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive. In *33rd USENIX Security Symposium*.
[52] Fangfei Zhou, Manish Goel, Peter Desnoyers, and Ravi Sundaram. 2011. Scheduler vulnerabilities and attacks in cloud computing. *arXiv preprint arXiv:1103.0759* (2011).

A Open Science

All the artifacts supporting this paper are available at https://github.com/SecureArch/trapped_handler.

B Ethical Considerations

In this paper, we identified a security vulnerability in the implementation of the preemption mechanism on NVIDIA GPUs and developed a performance degradation attack targeting vGPU environments. This attack allows one VM to cause significant performance degradation on co-located VMs. Although exposing the vulnerability identified in this paper poses risks for public cloud environments, it also raises awareness of security issues in vGPU, encourages vendors to strengthen their designs, and enables the research community to develop more robust defenses. As we have already provided mitigation methods and disclosed the vulnerability, we believe the benefits outweigh the risks.

To demonstrate the real-world impact of our attack, we evaluated it on public cloud platforms. However, all evaluation was conducted

under controlled conditions. Specifically, we only conducted attack experiments when we were able to rent all vGPU instances on a single physical GPU. This prevented the platform from placing other users' VMs on the same GPU. We believe it is very unlikely that live migration occurred (for one or more of our rented vGPU instances) during our experiments because live migration is used only for maintenance, upgrades, or load balancing across physical hosts—it is not a routine operation. In addition, we observed consistent slowdown in the victim instance, which indicates maintained co-location during the experiments.

C Trap Handler

Listing 2: The trap handler code.

```
Address      Instruction
7ffde3751100  BPT.DRAIN;
7ffde3751110  BMOV.32 MACTIVE, 0xffffffff ;
7ffde3751120  STL.128 [0xffffd0], R4 ;
7ffde3751130  STL.128 [0xffffe0], R8 ;
7ffde3751140  STL.128 [0xfffc0], R0 ;
7ffde3751150  MOV R4, R12 ;
7ffde3751160  MOV R5, R13 ;
7ffde3751170  P2R R6, PR, RZ, 0xff ;
7ffde3751180  BMOV.32 R7, TRAP_RETURN_MASK ;
7ffde3751190  STL.128 [0xfffff0], R4 ;
7ffde37511a0  RPCMOV.32 R8, Rpc.LO ;
7ffde37511b0  RPCMOV.32 R9, Rpc.HI ;
7ffde37511c0  BMOV.32 R10, OPT_STACK ;
7ffde37511d0  BMOV.32.CLEAR R11, B0 ;
7ffde37511e0  STL.128 [0xffffb0], R8 ;
7ffde37511f0  BMOV.32 R0, TRAP_RETURN_PC.LO ;
7ffde3751200  BMOV.32 R1, TRAP_RETURN_PC.HI ;
7ffde3751210  STL.64 [0xffffa8], R0 ;
7ffde3751220  MOV R1, 0xffffa0 ;
7ffde3751230  S2R R2, SR_GLOBALERRORSTATUS ;
7ffde3751240  LOP3.LUT R2, R2, 0x4, RZ, 0xc0, !PT ;
7ffde3751250  ISETP.EQ.AND P1, PT, R2, 0x4, PT ;
7ffde3751260  @!P1 BRA.U 0x7ffde3751360 ;
7ffde3751270  MOV R2, 0xde000000 ;
7ffde3751280  MOV R3, 0x7ffd ;
7ffde3751290  S2R R4, SR_VIRTUALSMID ;
7ffde37512a0  MOV R5, 0x85620 ;
7ffde37512b0  IMAD.WIDE.U32 R2, R4, R5, R2 ;
7ffde37512c0  IADD3 R2, P3, R2, 0x79010, RZ ;
7ffde37512d0  IADD3.X R3, R3, RZ, RZ, P3, !PT ;
7ffde37512e0  S2R R4, SR_VIRTID ;
7ffde37512f0  SHF.R.U32.HI R4, RZ, 0x8, R4 ;
7ffde3751300  SGXT.U32 R4, R4, 0x7 ;
7ffde3751310  MOV R5, 0x2d0 ;
7ffde3751320  IMAD.WIDE.U32 R2, R4, R5, R2 ;
7ffde3751330  LD.E.64 R4, [R2+0x188] ;
7ffde3751340  CCTLL.IVALL ;
7ffde3751350  SETLMEMBASE R4;
7ffde3751360  R2P PR, RZ ;
7ffde3751370  IADD3 R1, R1, -0x10, RZ ;
7ffde3751380  MOV R2, RZ ;
7ffde3751390  LEPC R12 ;
7ffde37513a0  CALL.ABS 0x7ffde3752100 ;
7ffde37513b0  MOV R3, RZ ;
7ffde37513c0  STL.64 [R1], R2 ;
7ffde37513d0  LOP3.LUT P0, RZ, R2, 0x1, RZ, 0xc0, !PT ;
7ffde37513e0  @!P0 BRA.U 0x7ffde3751420 ;
7ffde37513f0  LEPC R12 ;
7ffde3751400  CALL.ABS 0x7ffde3753300 ;
7ffde3751410  LDL R2, [R1] ;
7ffde3751420  LOP3.LUT P0, RZ, R2, 0x2, RZ, 0xc0, !PT ;
7ffde3751430  @!P0 BRA.U 0x7ffde3751490 ;
7ffde3751440  LEPC R12 ;
7ffde3751450  CALL.ABS 0x7ffde3753e00 ;
7ffde3751460  MOV R3, R2 ;
7ffde3751470  STL [R1+0x4], R3 ;
7ffde3751480  LDL R2, [R1] ;
7ffde3751490  LOP3.LUT P0, RZ, R2, 0x4, RZ, 0xc0, !PT ;
7ffde37514a0  @!P0 BRA.U 0x7ffde37514e0 ;
7ffde37514b0  LEPC R12 ;
7ffde37514c0  CALL.ABS 0x7ffde3754800 ;
7ffde37514d0  LDL R2, [R1] ;
7ffde37514e0  LOP3.LUT P0, RZ, R2, 0x8, RZ, 0xc0, !PT ;
7ffde37514f0  @!P0 BRA.U 0x7ffde3751540 ;
7ffde3751500  LEPC R12 ;
7ffde3751510  CALL.ABS 0x7ffde3755300 ;
7ffde3751520  MOV R2, RZ ;
7ffde3751530  BRA.U 0x7ffde3751390 ;
7ffde3751540  LOP3.LUT P0, RZ, R2, 0x10, RZ, 0xc0, !PT ;
7ffde3751550  @!P0 BRA.U 0x7ffde37515a0 ;
7ffde3751560  LEPC R12 ;
7ffde3751570  CALL.ABS 0x7ffde3756600 ;
7ffde3751580  LDL R2, [R1] ;
7ffde3751590  BRA.U 0x7ffde37517d0 ;
7ffde37515a0  LOP3.LUT P0, RZ, R2, 0x20, RZ, 0xc0, !PT ;
7ffde37515b0  @!P0 BRA.U 0x7ffde3751600 ;
7ffde37515c0  LEPC R12 ;
7ffde37515d0  CALL.ABS 0x7ffde3757100 ;
7ffde37515e0  MOV R2, RZ ;
7ffde37515f0  BRA.U 0x7ffde3751390 ;
7ffde3751600  LOP3.LUT P0, RZ, R2, 0x40, RZ, 0xc0, !PT ;
7ffde3751610  @!P0 BRA.U 0x7ffde3751670 ;
7ffde3751620  LEPC R12 ;
7ffde3751630  CALL.ABS 0x7ffde3757b00 ;
7ffde3751640  MOV R3, R2 ;
7ffde3751650  STL [R1+0x4], R3 ;
7ffde3751660  LDL R2, [R1] ;
7ffde3751670  LOP3.LUT P0, RZ, R2, 0x80, RZ, 0xc0, !PT ;
7ffde3751680  @!P0 BRA.U 0x7ffde37516c0 ;
7ffde3751690  LDL R3, [R1+0x4] ;
7ffde37516a0  LEPC R12 ;
7ffde37516b0  CALL.ABS 0x7ffde3758600 ;
7ffde37516c0  LDL.64 R4, [0xffffa8] ;
7ffde37516d0  BMOV.32 TRAP_RETURN_PC.LO, R4 ;
7ffde37516e0  BMOV.32 TRAP_RETURN_PC.HI, R5 ;
7ffde37516f0  LDL.128 R8, [0xffffb0] ;
7ffde3751700  RPCMOV.32 Rpc.LO, R8 ;
7ffde3751710  RPCMOV.32 Rpc.HI, R9 ;
7ffde3751720  BMOV.32 OPT_STACK, R10 ;
7ffde3751730  BMOV.32 B0, R11 ;
7ffde3751740  LDL.128 R0, [0xfffff0] ;
7ffde3751750  MOV R12, R0 ;
7ffde3751760  MOV R13, R1 ;
7ffde3751770  R2P PR, R2 ;
7ffde3751780  BMOV.32 TRAP_RETURN_MASK, R3 ;
7ffde3751790  LDL.128 R8, [0xffffe0] ;
7ffde37517a0  LDL.128 R4, [0xffffd0] ;
7ffde37517b0  LDL.128 R0, [0xfffc0] ;
7ffde37517c0  RTT ;
7ffde37517d0  EXIT.PREEMPTED.NO_ATEXIT ;
7ffde37517e0  BRA 0x7ffde37517e0 ;
```

D Evaluation on Alibaba T4 GPUs

Alibaba Cloud uses vGPU version 13.11, which was released in 2024. We find that for T4 GPUs, this version of vGPU handles preemption in a very different way from more recent vGPU versions (e.g., version 19.2 used in our local experiments). Specifically, with the Fixed Share policy, by modifying the trap handler, the attacker can keep the execution stalled in the trap handler for up to five minutes when it is preempted. After five minutes, the scheduler forces a context switch and applies compensation by not scheduling the attacker for a subsequent time period.

We further find that this compensation mechanism also has a limitation (different from the one in recent vGPU versions) and thus can be bypassed: if the attacker voluntarily terminates its program before the 5-minute threshold, compensation is applied but only lasts approximately 12 seconds. After this period, the attacker can relaunch the same program and repeat the preemption delay process. This means the victim can only use the GPU (without the attacker) for 12 seconds out of every 5-minute-12-second cycle. We find that this allows the attacker to achieve an average slowdown of 157.7× on short-running applications and 18.5× on long-running applications, reaching an average of 111.3× across all applications in Table 2. As discussed in Section 7.3, recent versions of vGPU do not have this problem.