TIMESLICE-SANDWICH: A GPU Side-Channel Attack Exploiting Time-Sliced Scheduling

Hodong Kim
Korea University

Gyeongsup Lim
Korea University

Seunghee Shin
State University of New York at Binghamton

Youngjoo Shin
Korea University

Junbeom Hur
Korea University

Abstract

Modern GPUs support resource sharing among concurrent applications, introducing the risk of side-channel attacks. While prior research has explored GPU side channels that exploit shared GPU resources, the security implications of time-sliced scheduling, a standard feature for resource sharing in today's GPUs, remain largely unexplored concerning side-channel attacks. In this study, we analyze timing variations caused by concurrent execution under the GPU's time-sliced scheduling mechanism. We begin by identifying the upper bound of a time slice and then leverage this bound to estimate the duration of a concurrent program's time slice, ultimately enabling us to infer the program's overall GPU utilization patterns.

Building on this finding, we introduce TIMESLICE-SANDWICH, a novel GPU side-channel attack that leverages variations in time-slice duration to infer and distinguish victim execution patterns. Unlike prior GPU side-channel attacks, TIMESLICE-SANDWICH does not require contention on specific shared resources. In our experiments, TIMESLICE-SANDWICH achieves an F1 score of 94.40% in neural network recovery attack; and a Top-1 accuracy of 92.84% in website fingerprinting attack on Google Chrome, both on average, demonstrating its effectiveness. Even in the presence of noise, our attack achieves an average F1 score of 73.74% for neural network recovery. Finally, we discuss potential mitigations to address side-channel risks arising from time-slice patterns in modern GPU resource-sharing architectures.

1 Introduction

Modern GPUs serve as critical computational resources for a wide array of applications [1, 23, 60], from daily web browsing [8] to deep learning [9], providing enhanced performance capabilities for general-purpose computing [62]. Many of these GPU-accelerated applications handle sensitive data, including users' privacy and proprietary intellectual property [36, 61]. Consequently, sufficient awareness should be followed to protect sensitive information in GPU applications from potential threats and data leakage [40].

Microarchitectural side-channel attacks have emerged as a major threat to today's GPU security [34, 41]. Given that modern GPUs commonly support temporal or spatial sharing [7, 11], hardware resources can be shared between multiple programs during execution. By exploiting contention in these shared resources as a source of a side channel, an attacker can retrieve sensitive information from co-located victim programs, such as a secret key of crypto algorithms [2, 16, 28, 29], neural network models [42, 70], or website fingerprints [13, 42, 76].

Prior research on GPU side channels has primarily exploited cache [2, 13, 16, 70, 76] and memory behaviors [28, 29, 42, 70] across different architectures. While these studies have uncovered numerous side-channel vulnerabilities in practical scenarios by targeting specific resource operations, the security implications of GPU resource scheduling mechanisms, which govern how resources are allocated among concurrent applications, remain relatively underexplored. In modern GPUs, scheduling decisions are made dynamically based on the tasks submitted by concurrently running applications, opening new vectors for side-channel leakage.

This dynamic scheduling can introduce measurable patterns that attackers exploit to infer information about executing tasks. Therefore, a thorough investigation of GPU scheduling behavior is essential to identify potential side-channel vulnerabilities and strengthen system defenses. Measuring the actual scheduled time of an application, however, presents a unique challenge. Because an application can only execute during its assigned time slices, conventional timing methods, such as counting threads [13, 16] or monitoring GPU clock cycles [2, 30], are ineffective for capturing precise scheduling behavior. For instance, on Ada Lovelace GPUs, the thread-based counters pause, and the GPU clock ceases to increment during other applications' time slices [76], resulting in discontinuous measurements. To address these limitations, we employ a timing method based on CUDA Runtime API event management functions [47]. This approach allows us to capture timing information across time slices allocated to

different applications, providing deeper insights into GPU scheduling behavior across diverse GPU architectures.

However, incorporating specific compute or memory operations into our timing method may introduce noise unrelated to scheduling behavior, potentially resulting in misinterpretation due to unintended contention on shared GPU resources. To mitigate this issue and isolate the effects of scheduling, we minimize resource utilization during measurement by launching a GPU thread that performs only an empty loop. This lightweight setup allows us to characterize key properties of time-slice behavior without introducing additional load. By analyzing the elapsed time reported by CUDA event management functions while executing this minimal thread, we observe that the measured duration may include contributions from time slices allocated to concurrently running workloads.

By varying the number of loop iterations in the thread, we uncover several undocumented properties of GPU scheduling, including the upper bound of a single time slice and the behavior of time slicing under the spatial sharing feature known as Multi-Process Service (MPS) [48]. These insights allow us to control the number of time slices used during the execution of a thread. Leveraging this knowledge, we develop an attack primitive, TIMESLICE-SANDWICH, that precisely estimates the duration of time slices allocated to a concurrently running workload. This attack captures timing differences induced by GPU time-sharing of a victim application, revealing distinct patterns tied to the victim's execution. In contrast to prior side-channel attacks, TIMESLICE-SANDWICH requires neither shared resource contention, nor co-location strategies [2, 13, 16, 28, 29, 42, 70, 76], nor privileged access to performance counters [42, 43, 70].

This paper demonstrates the effectiveness of TIMESLICE-SANDWICH on two practical GPU applications across three GPU architectures, Turing, Ampere, and Ada Lovelace. First, we conduct model recovery attacks on a deep learning application, a widely adopted, compute-intensive GPU workload. Since training durations vary across model architectures, the GPU schedules time slices differently depending on the model. By analyzing time-slice patterns from 20 distinct models in PyTorch for ImageNet training [63], we develop a classifier capable of accurately identifying the target model based on observed scheduling behavior. To evaluate input sensitivity on scheduling patterns, we repeat experiments using two different ImageNet training datasets [64], achieving average F1 scores of 94.80% and 94.00%, respectively, across the GPUs. These results indicate that time-slice patterns leak sufficient information to recover the trained model, even when different training inputs are used, an aspect largely overlooked by prior attacks [42, 70]. This evaluation highlights the practical security implications of GPU side-channels in deep learning workloads.

Next, we apply our attack to website fingerprinting against the Chrome browser. As modern browsers utilize GPUs differently when rendering webpages, prior studies have exploited shared resource contention to infer visited websites [13, 36, 42, 65, 76]. In contrast, we leverage time-slice patterns during page rendering for fingerprinting. Against Alexa's Top 200 websites, our attack achieves Top-1 and Top-3 classification accuracies of 92.11% and 96.79% under default scheduling, and 93.56% and 97.38% with MPS enabled, on average across the GPUs respectively, demonstrating the robustness of our approach.

Additionally, we evaluate the robustness of our attack under noisy conditions by conducting a model recovery attack. We introduce noise by co-running additional background workloads. When the number of background workloads is fixed, TIMESLICE-SANDWICH achieves an average F1 score of 85.87%. However, when the number of workloads varies substantially, the attack's performance degrades, reaching an F1 score as low as 50.16% in the worst case. We also discuss the broader applicability of our attack to web browsers, GPUs from other vendors, and multi-GPU systems. For example, although prior work has demonstrated website fingerprinting attacks in browsers [13, 65], we find that TIMESLICE-SANDWICH cannot be mounted from JavaScript in practice due to constraints inherent to web browser environments.

While demonstrating the efficacy of TIMESLICE-SANDWICH using established attack scenarios [13, 36, 42, 70, 76], our contribution is primarily to expose and validate a previously unexplored attack surface introduced by GPU scheduling mechanism, rather than to present a more powerful attack. To the best of our knowledge, this is the first study to demonstrate that a side-channel attack that exploits GPU time-sliced scheduling can leak sensitive information from real-world applications. Our findings highlight that securing time-slice behavior is crucial to protecting valuable data in both compute and graphics GPU workloads. Our study makes the following contributions:

- **Identification of a new attack surface:** We uncover the time-sliced scheduling mechanism as a source of side-channel leakage in GPUs. Leveraging CUDA Runtime API functions, we characterize undocumented scheduling properties on NVIDIA GPUs.
- **Introduction of a new attack primitive:** We introduce TIMESLICE-SANDWICH, a new side-channel attack that exploits time slice duration patterns to exfiltrate sensitive information from co-running applications. Unlike prior GPU attacks, our method does not depend on specific resource contention, making it more resilient against existing defenses that target resource-based side channels.
- **Demonstration on real-world applications:** We validate the effectiveness of TIMESLICE-SANDWICH on both compute- and graphics-intensive workloads through two real-world case studies: ① recovering models from deep learning training and ② fingerprinting websites rendered in the Chrome browser.

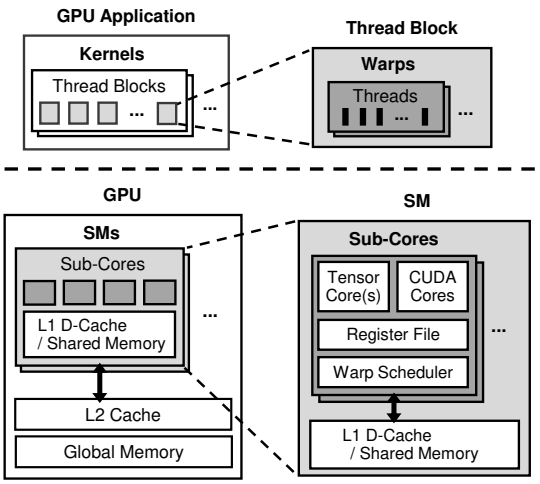


Figure 1: Overview of a GPU compute model

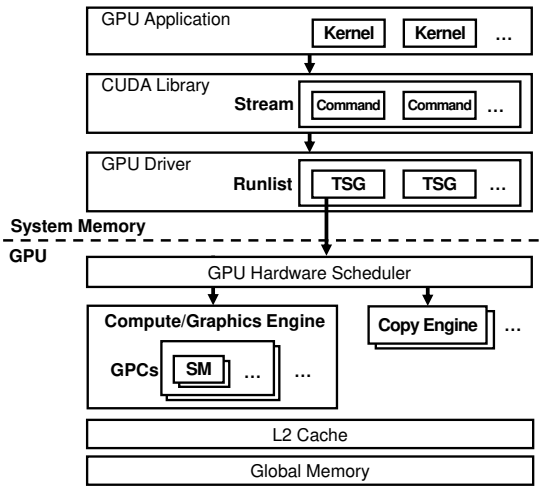


Figure 2: Overview of a GPU time-sliced scheduling

2 Background

This section provides background on the general architecture of GPUs and their applications. We begin by overviewing key components of the GPU programming APIs and computation model that are relevant to our work. We then describe the GPU's time-sliced scheduling mechanism, which is central to our study. Our discussion primarily focuses on NVIDIA GPUs, given their dominance in the GPU market [68]. Other vendors, such as AMD and Intel, employ similar concepts under different terminology.

2.1 GPU Programming

Applications use GPUs to accelerate graphics and compute tasks through specialized APIs. Conventional GPU APIs are typically tailored to a specific purpose: for instance, OpenGL provides a shader-based framework for graphics rendering [20], while OpenCL targets parallel computational tasks [19]. In contrast, recent APIs, such as Vulkan [21] and WebGPU [71], unify these capabilities by supporting both graphics and compute tasks. These APIs offer a broad compatibility across GPUs from various vendors.

NVIDIA provides CUDA, the most widely adopted general-purpose GPU computing platform [54]. CUDA programming requires the CUDA toolkit to be installed, while the NVIDIA graphics driver supplies the runtime libraries needed to execute applications. CUDA applications consist of one or more *kernel*s, which are functions executed on the GPU with a specified number of threads. The remaining parts of the programs typically run on a CPU thread and interact asynchronously with the GPU through CUDA API calls. Launched kernels are enqueued into a *stream*, which is a FIFO software queue that ensures ordered execution [53]. All kernels in a stream operate within a *context*, a virtual address space that is typically shared by the program's kernels.

2.2 GPU Compute Model

NVIDIA GPUs are designed to support massive parallelism, leveraging thousands of computing units. To maintain this scale, the GPU hardware is organized in a hierarchy, as depicted in Figures 1 and 2. A GPU employs several copy engines to handle data transfers between the host system and GPU memory, while the compute/graphics engine consists of general processing clusters (GPCs) responsible for executing computations [7]. Task execution across these engines is coordinated by the GPU hardware scheduler, also known as the host interface. An application typically launches several GPU kernels consisting of a grid of thread blocks. Each GPC contains several streaming multiprocessors (SMs), where thread blocks are assigned for execution. Within a block, 32 threads are tied to a warp, the base unit scheduled by the hardware.

All threads in a warp are simultaneously scheduled on a sub-core of an SM and process the same instruction on different data in a single-instruction, multiple-thread (SIMT) fashion. Each sub-core contains multiple CUDA cores that execute a GPU thread one at a time and represent the smallest unit of computation. Starting with the Volta architecture, sub-cores include tensor core units specialized for multi-threaded matrix operations. Every sub-core has a dedicated warp scheduler that selects a warp to execute every cycle. When a warp is first dispatched, the sub-core's registers are exclusively assigned to each thread in the warp. Within an SM, shared memory and L1 cache are accessible to all threads assigned to the SM. The L2 cache and global memory, on the other hand, are shared across all threads in the GPU.

2.3 GPU Scheduling Mechanism

NVIDIA GPUs support time-sliced scheduling by default through their device drivers. In this temporal sharing mechanism, the GPU allocates time slices to each application and

executes them sequentially. Figure 2 describes a high-level overview of GPU application scheduling. When an application launches kernels, CUDA enqueues the tasks into either the default or a user-specified stream. Each stream contains a sequence of GPU commands, such as kernel launches, memory copies, and synchronization operations, which are executed within a context. The GPU driver groups these tasks into a time slice group (TSG) and manages them in a runlist with their associated context information [7, 12]. On the GPU side, the hardware scheduler iterates through the TSGs in the runlist, selects one to activate, and dispatches its commands to execute on the required engines during a time slice in a round-robin manner [7].

Since only a single context can be active during a time slice, a program fully occupies the GPU's resources while scheduled [52]. However, this exclusivity often leads to inefficiencies when applications require only a fraction of GPU resources. To address the problem, spatial sharing mechanisms can be employed to improve resource utilization. For example, NVIDIA provides MPS [48], which allows CUDA applications to share GPU resources spatially. When this feature is enabled, applications run as clients under the MPS server, sharing the server's context. This client-server model enables simultaneous execution of multiple client programs without time-slicing between them, alleviating the inefficiency of assigning each program to a separate context.

3 Unveiling Time-Sliced Scheduling Behaviors

In this section, we investigate the scheduling behavior of an NVIDIA GPU to uncover key properties that can be exploited through time-slice patterns as a side channel. We introduce a method for accurately measuring timing effects of time-sliced scheduling using unprivileged, user-level functions from the CUDA Runtime API. Leveraging this timing information, we explore previously undocumented scheduling properties, including the upper bound of a time slice and the observability of time slices even when MPS is enabled.

3.1 Timing Method

Time slicing occurs when multiple programs execute concurrently on a GPU. Analyzing timing variations in this environment introduces two main challenges: ① accurately measuring execution time across time slices allocated to different programs, and ② minimizing interference from factors unrelated to scheduling behavior, such as resource contention. Our timing method addresses these challenges by meeting two key requirements. First, it leverages event management functions from the CUDA Runtime API to reliably capture durations across time-slice boundaries. Second, it employs a duration-configurable custom kernel with minimal resource usage, enabling measurements to capture time slices assigned to other programs.

```
1  for (i=0; i<n; i++){
2      cudaEventRecord(start);
3      busyWaitKernel<<<1, 1>>>();
4      cudaEventRecord(end);
5      cudaEventSynchronize(end);
6      cudaEventElapsedTime(&time, start,
           end);
7      execTimeSamples[i] = time;
8  }
```

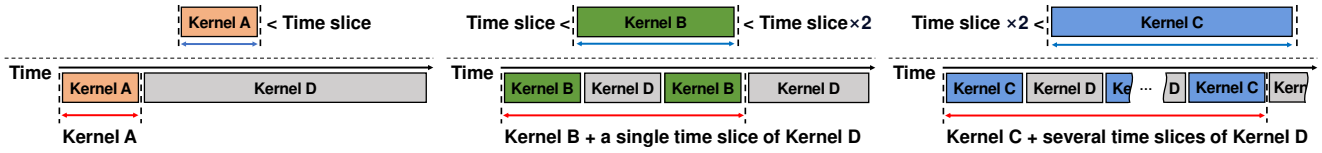
Figure 3: Code snippet for measuring elapsed times executing a kernel, using event management functions in CUDA Runtime API

```
1  __global__ void busyWaitKernel(){
2      for(int x=0; x<NUM_ITER; x++){}
3  }
```

Figure 4: Code snippet of a busy-wait kernel

Figure 3 describes a code snippet of our timing method. In this method, CPU threads call CUDA functions and kernels, which interact with the GPU to perform the corresponding tasks. First, the `cudaEventRecord` function captures a timing point, called an *event*, from the stream where the method is enqueued, and stores it as `start` (line 2). Then, the busy-wait kernel executes in a single GPU thread (line 3). Figure 4 illustrates the design of the busy-wait kernel, which consists solely of an empty loop with no compute or memory operations. The kernel's duration is configured by adjusting the number of loop iterations, ensuring negligible resource contention and stable execution time even when co-running with other workloads. To prevent compiler optimizations from eliminating the loop, the kernel must be compiled with the NVIDIA CUDA Compiler (NVCC) using the `-G` flag [57].

After the kernel execution, `cudaEventRecord` records another timing point, `end`, as shown in Figure 3 (line 4). Because both event recording and kernel execution are issued in the same stream, they execute in order. Then, the `cudaEventSynchronize` function blocks the CPU thread until the `end` is recorded (line 5). Finally, the `cudaEventElapsedTime` function calculates the elapsed time between `start` and `end` and stores the result in `time`, with a resolution of 0.5 µs (line 6). While these event management functions are designed to measure the timing of tasks within a single context [47], the measured duration can also include time slices allocated to other co-running applications. This arises from the GPU hardware scheduler, which serializes tasks across contexts when assigning time slices to multiple applications [7]. For instance, if the busy-wait kernel in line 3 spans more than one time slice, the measured duration will include GPU time used by another application scheduled between those slices.



(a) None (b) A single time-slice (c) Several time-slices

Figure 5: Different number of captured time slices depending on the execution times of kernels.

3.2 Upper Bound on Time-Slice Duration

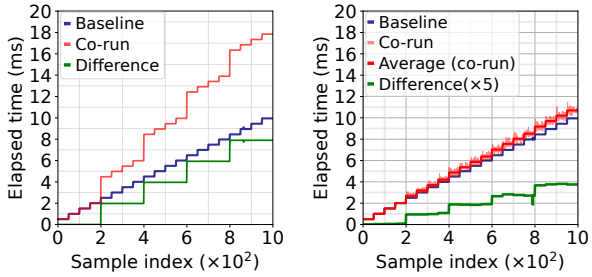
NVIDIA has not fully disclosed the details of time-slice durations under its default scheduling mechanism, specifically whether time slices are equally distributed among programs or can vary in length up to a certain upper bound. This study is motivated by the possibility of inferring the number of time slices in the timing measurement by analyzing the duration of a busy-wait kernel, provided that the scheduler enforces a fixed upper limit on each slice.

When the GPU enforces a fixed upper bound on the duration of a single time slice, any kernel execution that exceeds this limit is split across multiple slices. Figure 5 illustrates three scenarios of interleaved time slices when two applications run concurrently on a GPU. In each case, the kernels represent tasks from different applications with varying durations and run concurrently without resource contention. Blue arrows depict the elapsed time of kernels running alone in isolation, while red arrows represent the measured durations when the same kernels execute concurrently with another application.

As shown in Figure 5a, a kernel whose execution time falls within the upper bound completes entirely within a single slice, avoiding time-slicing. In contrast, Figures 5b and 5c depict cases where the kernel's duration exceeds the time-slice boundary, causing execution to be split across multiple slices. In these cases, the measured elapsed time during co-execution increases proportionally with the number of slices incurred. This scheduling behavior creates measurable patterns, as kernel executions that cross the slice boundary are repeatedly interrupted, resulting in longer overall execution times.

To identify patterns indicative of the upper bound in GPU time-slice duration, we measure and compare the elapsed time of a busy-wait kernel executed with varying durations under two configurations: ① *baseline*, where the busy-wait kernel runs alone in isolation, and ② *co-run*, where it runs concurrently with a background workload.

As concurrent workloads, we employ both compute and graphics programs that repeatedly execute fixed routines. The compute workload, based on the NVIDIA CUDA Sample [45, 46], launches a kernel that performs general matrix multiplication (GEMM) with an execution time of approximately 1.8 ms. The graphics workload, derived from an OpenGL sample [69], continuously renders a rotating square. We first



(a) GEMM (b) OpenGL

Figure 6: Elapsed time of busy-wait kernels with durations from 0.5 ms to 10 ms in 0.5 ms increments under GEMM and OpenGL workloads on an RTX 3080 GPU

present our analysis using the GEMM compute workload and then apply the same methodology to the OpenGL graphics workload. The procedure consists of four steps:

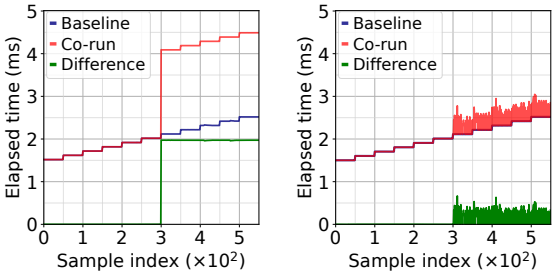
Step 1. Our first step is to determine if time-slicing occurs between the busy-wait kernel and the background workload, which we evaluate by comparing their execution times under co-scheduling.

Step 2. We measure the baseline performance by executing the busy-wait kernel in isolation, incrementing its duration in fixed steps.

Step 3. We repeat the same duration-based tests while running the busy-wait kernel concurrently with the background workload.

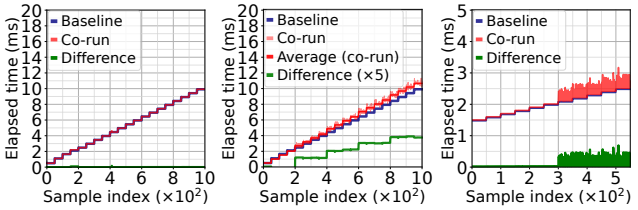
Step 4. We compare baseline and co-run measurements to identify timing differences indicative of a fixed upper bound on time-slice duration.

Figure 6 presents the baseline and co-run measurements for both the GEMM and OpenGL workloads on an NVIDIA RTX 3080 GPU. Unless otherwise noted, figures presenting results in this paper use the sample index on the horizontal axis and the corresponding measured elapsed time on the vertical axis. For each baseline and co-run test set, we collected 1,000 samples by performing 50 measurements per busy-wait kernel duration, incrementing the duration from 0.5 ms to 10.0 ms in 0.5 ms steps.



(a) GEMM (b) OpenGL

Figure 7: Elapsed time of busy-wait kernels with durations from 1.5 ms to 2.5 ms in 0.1 ms increments under GEMM and OpenGL workloads on an RTX 3080 GPU



(a) GEMM ([0.5, 10] ms, Δ = 0.5 ms) (b) OpenGL ([0.5, 10] ms, Δ = 0.5 ms) (c) OpenGL ([1.5, 2.5] ms, Δ = 0.1 ms)

Figure 8: Elapsed time of busy-wait kernels under GEMM and OpenGL workloads on an MPS-enabled NVIDIA RTX 3080 GPU

Figure 6a illustrates distinct measurement patterns under the GEMM workload that reveal the presence of an upper bound on time-slice duration. In the range below 2.5 ms, the elapsed times measured under baseline and co-run configurations substantially overlap, showing no evidence of time-slicing for shorter kernels. For durations from 2.5 ms to 4.0 ms, the co-run results consistently exhibit timing differences of approximately 2 ms compared to the baseline. We interpret this shift as the onset of time-slicing, where the delay reflects one full slice allocated to the GEMM workload (1.8 ms) plus 0.2 ms of timing measurement overheads. Subsequent time-slicing points appear at 4.5 ms, 6.5 ms, and 8.5 ms, with timing differences increasing in a consistent pattern that allows us to infer the number of time slices allocated to the co-running kernels.

Figure 6b shows co-run measurements with the OpenGL graphics workload. Unlike the GEMM case, these measurements exhibit higher variability. To clarify the behavior, the figure amplifies the difference between the average co-run times and the corresponding baseline times by a factor of five. The amplified difference reveals identical time-slicing patterns at the same duration thresholds as those observed with the GEMM workload. These findings indicate that time-slicing is consistently triggered across both compute and graphics workloads when the busy-wait kernel duration surpasses approximately 2.0 ms to 2.5 ms.

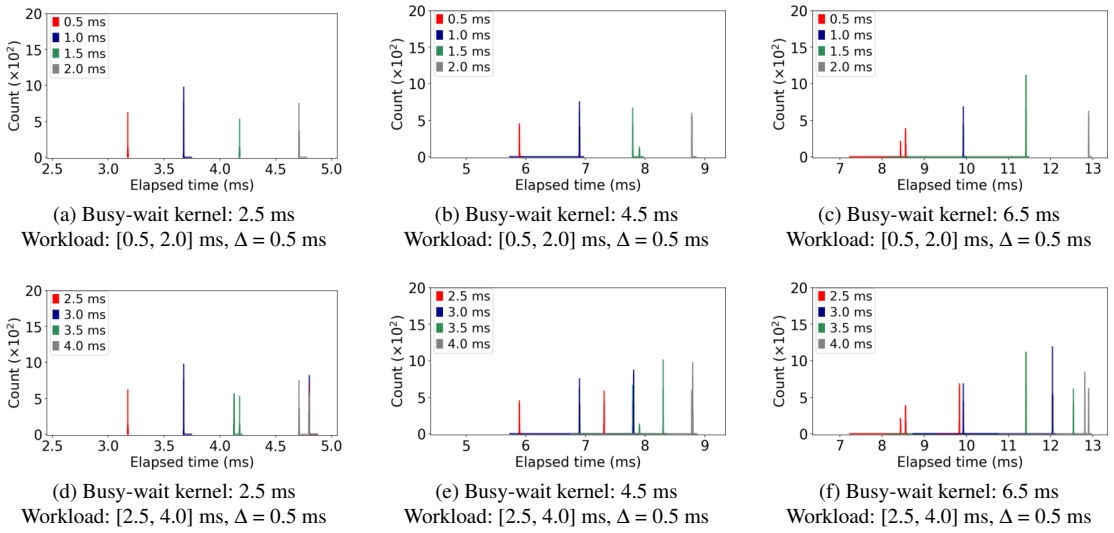
To refine our estimate of the time-slice boundary, Figure 7 presents fine-grained measurements with busy-wait kernel durations incremented in 0.1 ms steps from 1.5 ms to 2.5 ms, targeting the first time-slicing trigger moment. A total of 550 samples were collected for each of the baseline and co-run configurations using the same GPU setup and workloads. These figures confirm that time-slicing begins at a kernel duration of 2.1 ms in both workloads, as indicated by the divergence between the baseline and co-run measurements. Taken together, these observations indicate that the GPU imposes an upper bound of approximately 2.0 ms per time slice during kernel execution.

3.3 Time Slices Under Enabled MPS

This section examines GPU time-slicing behavior under the MPS configuration using the same timing measurement methodology described in the previous section. An MPS server instance allows context sharing among client applications, effectively eliminating the need for time-slicing within the GPU. However, because MPS is designed specifically for CUDA workloads [48], graphics APIs such as OpenGL are not compatible with the MPS client-server model [76]. As a result, when a CUDA application and an OpenGL-based graphics application execute concurrently on an MPS-enabled GPU, time-slicing can still occur between them. This creates an opportunity for a CUDA application operating within the MPS context to monitor timing against a co-running graphics workload outside the MPS context.

Figure 8 shows the baseline and co-run measurements for the GEMM compute workload and the OpenGL graphics workload on an MPS-enabled GPU. We follow the same test procedure and duration configurations as in the previous section, including both coarse-grained (0.5 ms increments) and fine-grained (0.1 ms increments) kernel duration steps. For each configuration, we collect 1,000 samples for coarse-grained and 550 samples for fine-grained tests, with 50 measurements per busy-wait kernel duration.

Figure 8a presents the coarse-grained measurements under the GEMM workload. As anticipated, the results suugest that MPS suppresses time-slicing between the busy-wait kernel and the GEMM workload, implying that additional fine-grained testing is unnecessary. In contrast, Figures 8b and 8c illustrate that time-slicing patterns, previously identified under default scheduling, remain present with MPS enabled. These results confirm that time-slicing continues to occur when the co-running workload is based on OpenGL, even in an MPS-enabled environment, allowing the number of time slices to be inferred through timing measurements.



(a) Busy-wait kernel: 2.5 ms
Workload: [0.5, 2.0] ms, Δ = 0.5 ms

(b) Busy-wait kernel: 4.5 ms
Workload: [0.5, 2.0] ms, Δ = 0.5 ms

(c) Busy-wait kernel: 6.5 ms
Workload: [0.5, 2.0] ms, Δ = 0.5 ms

(d) Busy-wait kernel: 2.5 ms
Workload: [2.5, 4.0] ms, Δ = 0.5 ms

(e) Busy-wait kernel: 4.5 ms
Workload: [2.5, 4.0] ms, Δ = 0.5 ms

(f) Busy-wait kernel: 6.5 ms
Workload: [2.5, 4.0] ms, Δ = 0.5 ms

Figure 9: Elapsed time of fixed-duration busy-wait kernels under workloads with varying kernel duration on an RTX 3080 GPU

4 TIMESLICE-SANDWICH Attack

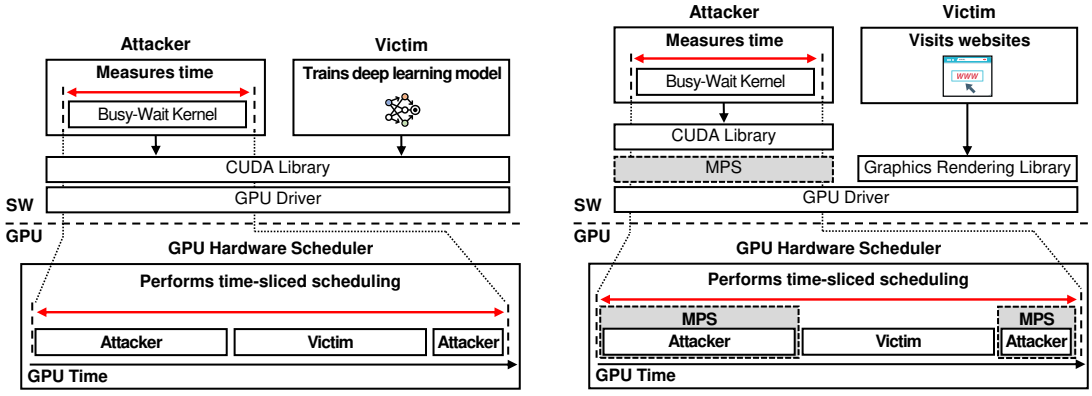
In this section, we construct the TIMESLICE-SANDWICH attack primitive, which enables GPU side-channel attacks by exploiting the time-sliced scheduling mechanism identified in our earlier analysis.

Using the upper bound of a single time slice, we configure a busy-wait kernel with a fixed duration that induces a specific number of time-slicing events when executed concurrently with another workload. As a result, the timing measurements of this kernel consistently include the same number of time slices triggered by its configured duration. By repeatedly launching the kernel and examining its execution times, we capture a sequence of timing measurements that reflect the time slices allocated to the concurrent application. Since the GPU dynamically allocates time slices based on the nature of concurrent tasks, the resulting measurements reveal distinctive allocation patterns that can be exploited to infer sensitive information from the victim application.

Based on this insight, we determine an effective duration for the busy-wait kernel. We first examine how the number of time slices triggered by the busy-wait kernel affects the ability to distinguish slices allocated to concurrent workloads on the NVIDIA RTX 3080 GPU. In this analysis, we run busy-wait kernels with durations of 2.5 ms, 4.5 ms, and 6.5 ms, which trigger one, two, and three time slices, respectively, thereby capturing different allocations to co-running workloads. As target workloads, we deploy eight independent busy-wait kernels with durations ranging from 0.5 ms to 4.0 ms in 0.5 ms increments. During concurrent execution, the GPU allocates a single time slice to target workloads with durations between 0.5 ms and 2.0 ms, while workloads exceeding this range receive two slices.

Figure 9 presents histograms of the timing results collected by running the three busy-wait kernels concurrently with target workloads. In this histogram, the horizontal axis represents the measured elapsed time, while the vertical axis indicates the number of measured samples. For each busy-wait kernel configuration, we measure 2,500 samples per workload duration, resulting in a total of 10,000 measurements per histogram. We consider a busy-wait kernel more effective for distinguishing workloads when its measurements are tightly clustered for the same duration and widely separated across different durations.

Figures 9a, 9b, and 9c show the measurements for busy-wait kernels with durations of 2.5 ms, 4.5 ms, and 6.5 ms, respectively, under workloads that run within a single time slice. The measured elapsed times consist of three components: the execution time of the busy-wait kernel, the duration of the captured workload, and a measurement overhead of approximately 0.2 ms per time slice, as we observed in Section 3.2. For example, a 2.5 ms busy-wait kernel (Figure 9a) captures a single 0.5 ms workload slice and incurs 0.2 ms of overhead, resulting in a total elapsed time of approximately 3.2 ms (shown as a red bar). Similarly, the 4.5 ms and 6.5 ms busy-wait kernels capture two and three slices of the same workload, respectively, resulting in total elapsed times of approximately 5.9 ms and 8.6 ms. As shown in Figure 9a, the 2.5 ms kernel clearly separates workloads of different durations, indicating that this configuration is effective for distinguishing workloads confined to a single time slice. Figures 9b and 9c reveal some noise at the bottom of the distributions, which we attribute to cases where the 4.5 ms and 6.5 ms busy-wait kernels capture workloads across two or three time slices. Nonetheless, the overall results show consis-



(a) DNN model recovery attack

(b) Website fingerprinting attack

Figure 10: Exploiting time-sliced scheduling side-channel information leakage on a GPU

tent patterns, confirming that our method effectively captures co-running workload characteristics.

Figures 9d, 9e, and 9f illustrate that when workloads span across two time slices, the level of noise in the measurements increases. This arises because the GPU divides each workload into two sub-tasks: the first, lasting approximately 2.1 ms, the maximum duration of a single time slice, and the second accounting for the remaining execution time. Consequently, the 2.5 ms busy-wait kernel may capture either the 2.1 ms time slice or the slice corresponding to the residual execution time in Figure 9d. In Figures 9e and 9f, the 4.5 ms and 6.5 ms kernels capture several combinations of the divided sub-tasks, displaying increased noise with additional timing groups for the same workload.

As a result, we empirically select a 2.5 ms busy-wait kernel to construct the TIMESLICE-SANDWICH attack primitive on an NVIDIA RTX 3080 GPU. This primitive comprises the following five steps:

Step 1. Call `cudaEventRecord` to record the first timing point.

Step 2. Launch the busy-wait kernel with a duration of 2.5 ms.

Step 3. Call `cudaEventRecord` again to record the second timing point.

Step 4. Call `cudaEventSynchronize` to wait for the GPU to complete execution.

Step 5. Call `cudaEventElapsedTime` to calculate the elapsed time between the two recorded timing points.

5 Attacks on Practical Applications

In this section, we evaluate the security impact of TIMESLICE-SANDWICH by conducting model recovery attacks on deep learning applications and website fingerprinting attacks. In these scenarios, the attacker aims to exfiltrate sensitive information from a victim's GPU application running concurrently on the same system by capturing time-slice patterns. The system is equipped with an NVIDIA GPU running the vendor-provided graphics driver and CUDA, with all security patches applied. The attacker executes native user-level code without requiring the victim's cooperation or elevated privileges, for example, access to CUDA internals or GPU drivers. In this setting, we focus on demonstrating the practical efficacy of our attack, establishing the scheduling mechanism itself as a new GPU side-channel surface. Broader applicability of this attack is discussed in Section 6.

5.1 Attack Overview

Figure 10 illustrates the overview of the TIMESLICE-SANDWICH attack. In the attack, we focus on two target categories of sensitive information: ① models trained by deep learning applications and ② websites rendered by a web browser. In our setup, the GPU uses the default scheduling mechanism for a deep learning application; while for browser workloads, it operates either in default mode or with MPS enabled. MPS is generally not critical for deep learning applications, as these workloads inherently require significant GPU resources and can achieve high utilization by adjusting parameters such as batch size. By design, MPS primarily addresses the underutilization problem of lightweight GPU applications [48].

In both scenarios, the attacker executes the attack primitive, while the victim runs its application on the same GPU. The GPU allocates time slices of varying durations to the victim applications depending on its behavior, while the attack primitive consistently receives time slices of approximately 2.1 ms and 0.4 ms, capturing a single slice of the victim in between. By repeatedly launching the attack primitive, the attacker collects timing measurements that include variances

Table 1: GPU configurations for Section 5

GPU (Release)	Architecture	OS	NVIDIA Driver (Release)	CUDA (Release)
RTX 2080 Ti (Sep 2018)	Turing	Ubuntu 20.04	495.44 (Oct 2021)	11.5 (Oct 2021)
RTX 3080 (Sep 2020)	Ampere	Ubuntu 20.04	530.03.02 (Feb 2023)	12.1 (Feb 2023)
RTX 4090 (Oct 2022)	Ada Lovelace	Ubuntu 22.04	570.144 (Apr 2025)	12.8 (Jan 2025)

introduced by the victim's slices. These variances produce unique patterns in the GPU's scheduling behavior, which the attacker analyzes to exfiltrate the targeted information.

To reliably identify patterns in the timing measurements, the attacker employs a long short-term memory (LSTM) model as a classifier [31]. LSTMs are well-suited for this task because they automatically learn temporal features, such as periodic patterns and inter-burst intervals, relevant to classification using memory cells and gates within the network, without requiring manual feature selection. The overall attack process consists of three phases:

1. Configuration phase: the attacker collects profiling data by running the attack primitive concurrently with the target application. The attacker then trains the classifier using the profiling data.

2. Online phase: the attacker repeatedly launches the attack primitive to collect elapsed time during the victim's execution.

3. Offline phase: the attacker applies the trained classifier to the collected measurements, identifying patterns introduced by the victim's time slices to exfiltrate the target information.

Using the attack primitive, the attacker profiles a set of candidate models (or websites) that the victim might train (or visit) and builds a classifier to recover them. During data collection, each timing result from the attack primitive is stored in a CSV file in sequential order, maintaining the temporal order of the measurements. This order directly reflects the sequence of time-slice durations allocated to the victim's execution over time.

Experimental setup. We conduct our attacks on a host system equipped with an AMD Ryzen 9 3950X processor and 32GB of RAM across three different GPU settings. Table 1 summarizes the GPU configurations for our experiments. To evaluate the deployability of our attack across diverse GPU setups, we employ NVIDIA RTX 2080 Ti, RTX 3080, and RTX 4090, representing Turing, Ampere, and Ada Lovelace architectures, respectively. For each GPU, we deploy a dedicated attack primitive by configuring an effective busy-wait kernel. Interestingly, all three GPUs enforce the same upper bound on time-slice duration, though the number of loop iterations required in the busy-wait kernels varies across GPUs.

For the software setup, we use Ubuntu 22.04 for the RTX 4090 and 20.04 for the other GPUs. The NVIDIA driver and CUDA versions installed for each GPU are: 495.44 and 11.5 for the RTX 2080 Ti, 530.03.02 and 12.1 for the RTX 3080, and 570.144 and 12.8 for the RTX 4090. We note that each CUDA version is bundled with its respective NVIDIA driver installation.

Table 2: List of target DNN models

Family	Architectural Difference	Model
VGG [66]	Number of Layers	vgg11, vgg13, vgg16, vgg19
	Batch Normalization	vgg11_bn, vgg13_bn, vgg16_bn, vgg19_bn
ResNet [22,73,75]	Number of Layers	resnet18, resnet34, resnet50, resnet101, resnet152
	Number of Filters	wide_resnet50_2, wide_resnet101_2
	Parallel Paths in Building Blocks	resnext50_32x4d
ShuffleNet V2 [39]	Number of Filters, Optimization†	shufflenet_v2_x0_5, shufflenet_v2_x1_0, shufflenet_v2_x1_5, shufflenet_v2_x2_0

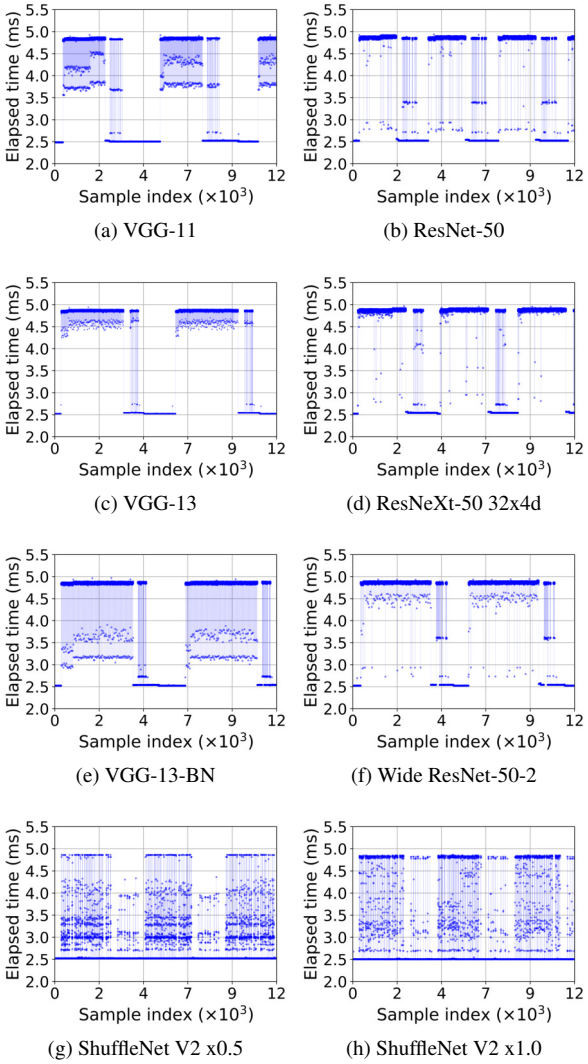
† ShuffleNet V2 models employ optimizations for reducing memory and computation overhead [39].

5.2 DNN Model Recovery Attack

This section demonstrates how an attacker can exploit TIMESLICE-SANDWICH to exfiltrate a victim's trained model while training on the ImageNet dataset with PyTorch [63, 64]. The attack is first carried out on an RTX 3080 and subsequently replicated on RTX 2080 Ti and RTX 4090 GPUs.

Target investigation. Deep learning applications execute several GPU kernels for computations such as matrix multiplications and convolutions to train deep neural networks (DNNs) [46]. Since these computations vary across model architectures, such as layers or filters, GPUs may allocate time slices differently depending on the model being trained. Compared to inference, DNN training typically involves multiple iterative epochs. This repetitive nature allows an attacker to repeatedly attempt to recover the model, increasing the likelihood of a successful attack in practice. We demonstrate the feasibility of this scenario by conducting model recovery using a single training epoch. To investigate this property as a basis for our attack, we select representative candidate models and analyze their time-slice patterns.

Table 2 lists the 20 victim model candidates from the PyTorch ImageNet training implementation [63]. Each model in the VGG family has a different number of layers, ranging from 11 to 19 [66], with BN variants applying batch normalization for each layer. ResNet models employ deeper networks, ranging from 18 to 152 layers, by leveraging building blocks called residual blocks [22]. Wide ResNet variants double the number of filters per convolutional layer [75], while ResNeXt introduces 32 parallel paths within each residual



(a) VGG-11 (b) ResNet-50

(c) VGG-13 (d) ResNeXt-50 32x4d

(e) VGG-13-BN (f) Wide ResNet-50-2

(g) ShuffleNet V2 x0.5 (h) ShuffleNet V2 x1.0

Figure 11: Time-slice patterns of training different models on an RTX 3080 GPU

block [73]. Unlike the other families, ShuffleNet V2 models apply optimizations for reducing memory and computation overheads, maintaining a constant number of layers but varying the number of filters [39].

To collect profiling data, we launch TIMESLICE-SANDWICH primitive while training the 20 candidate models. We measure 100 training epochs for each model, using 4,000 randomly selected images from the ImageNet dataset as training input [64]. We collect 20,000 samples across 20 classes by grouping the measurements of each epoch into a single sample labeled with the corresponding model.

We then analyze how architectural differences affect time-slice behavior by examining eight representative models in the profiled data. Figure 11 illustrates different temporal patterns in the time slices of these models, with 12,000 measurements collected per model. These patterns recur across training iterations of a model, though the number of samples spanned and their elapsed times vary across models. For example, VGG-13 employs two additional layers compared to VGG-11, resulting in patterns that span more samples with elapsed times around 4.5 ms (Figures 11a and 11c). Applying batch normalization further extends the patterns: VGG-13-BN produces additional samples with elapsed times ranging from 3.0 ms to 4.0 ms, compared to VGG-13 (Figures 11c and 11e).

Within the ResNet family, the elapsed times of most samples fall between 4.5 ms and 5.0 ms during training epochs. Variants introduce distinctive expansions of these patterns: ResNeXt, which adopts blocks with parallel paths, produces recurring patterns that span more samples than the ResNet-50 (Figures 11b and 11d), while Wide ResNet-50-2, with more filters, extends the patterns to even more samples (Figures 11b and 11f). In contrast, ShuffleNet V2 models show patterns that are similarly spanned over samples despite using different numbers of filters, though their measured timing values still differ between models (Figures 11g and 11h). We attribute this to the architecture's optimizations for reducing memory and computation overhead. Although some models, such as VGG-11, ResNet-50, and the ShuffleNet V2 variants, produce patterns of similar span, their elapsed-time distributions differ significantly (Figures 11a, 11b, 11g, and 11h). These measurable differences indicate that temporal time-slice patterns provide sufficient information to train an LSTM classifier for accurately identifying a victim's model.

Attack evaluation. To evaluate the effectiveness of TIMESLICE-SANDWICH in a realistic environment, we test the online phase under two scenarios based on the victim's training input of 4,000 ImageNet images: ① *identical input*, where the victim selects the same images used in candidate profiling, and ② *disjoint input*, where the victim selects images not included in the candidate profiling dataset. For each case, we capture 50 training epochs across all candidate models by launching the attack primitive. Treating each epoch as a single sample, we collect 1,000 samples per case, resulting in a total of 2,000 samples in the test set.

We implement an LSTM-based classifier to distinguish the victim's trained model from 20 candidates [31]. This classifier consists of two LSTM layers, each with 128 hidden states, followed by a fully connected output layer with 20 neurons. This output layer uses a Softmax activation function to generate a probability distribution over the 20 candidate model classes. For training this classifier, we use the profiling data of 20,000 samples, 100 samples per model, shuffled with a fixed random seed of 0. We use a window size of 10 for this input data, with a padding of 1 applied to sequences shorter than the window size. The classifier is trained for 50 epochs using the AdamW optimizer [38] and binary cross-entropy with logits loss function [32]. We set the learning rate to 0.0001, the weight decay to 0.01, and apply a dropout rate of 20% [67]. Using this classifier, we evaluate our attack's

Table 3: Result of DNN model recovery attacks

GPU	Training Input	F1 Score (%)	Precision (%)	Recall (%)
RTX 2080 Ti	Identical†	94.90	95.40	94.40
	Disjoint‡	94.04	94.90	93.20
RTX 3080	Identical	93.73	92.50	95.00
	Disjoint	93.53	92.30	94.08
RTX 4090	Identical	95.75	95.91	95.60
	Disjoint	94.43	94.86	94.00
Average		94.40	94.31	94.50

† The victim uses the training input dataset profiled by the attacker.
‡ The victim uses a training input dataset never profiled by the attacker.

performance on the test set under both identical and disjoint input scenarios.

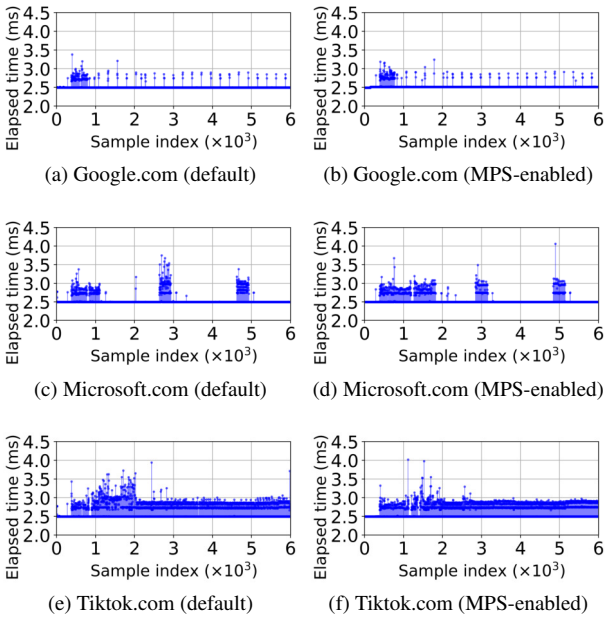
Table 3 summarizes the attack results on RTX 2080 Ti, RTX 3080, and RTX 4090 GPUs. To evaluate the effectiveness of our model recovery attack, we use precision, recall, and F1 score as metrics. Our attack achieves at least 92.30% in every metric, underscoring the effectiveness of TIMESLICE-SANDWICH for DNN model recovery. On average, the attack achieves a precision of 94.31%, indicating that the attacker correctly infers the victim's model in the vast majority of attempts. The achieved recall rate averages 94.50%, showing that when the victim uses a specific model, the attacker correctly identifies it nearly all the time. The F1 score, which represents the harmonic mean of precision and recall, averages 94.40%, confirming that our attack primitive achieves an excellent balance between the two. These results demonstrate that our attack remains effective even under the challenging condition of disjoint inputs.

5.3 Website Fingerprinting Attack

Next, we evaluate our TIMESLICE-SANDWICH primitive in the context of a website fingerprinting attack. In this scenario, the attacker infers which website is being rendered in the victim's browser by launching the attack primitive during the page load. For the victim's browser, we use Chrome version 138.0.7204.168 [17]. We perform this attack with identical process on RTX 2080 Ti, RTX 3080, and RTX 4090 GPUs.

Target investigation. The Chrome web browser utilizes GPUs to render websites by default using several graphics libraries [17, 18, 27] and rendering APIs, including OpenGL [20]. Using these APIs on NVIDIA GPUs requires the installation of the vendor-provided driver [4, 56]. Because the driver includes the CUDA runtime, an attacker can deploy the attack primitive to capture website-rendering activity in the browser without further system setup.

Websites often include various graphical objects to render, such as text cursors, images, and videos. Since the required rendering tasks vary with website designs, time slices allocated to rendering a website can show distinct patterns due to the GPU's scheduling behavior. For example, search engine websites often have simple page designs with minimal loading



(a) Google.com (default) (b) Google.com (MPS-enabled)
(c) Microsoft.com (default) (d) Microsoft.com (MPS-enabled)
(e) Tiktok.com (default) (f) Tiktok.com (MPS-enabled)

Figure 12: Time-slice patterns of rendering different websites on an RTX 3080 GPU

of objects after the initial render, except for the blinking text cursor. In contrast, rendering websites that include videos or dynamic images consistently utilize GPU resources over time. Building on this observation, we first identify such patterns by profiling candidate websites.

For our candidates, we select 200 websites from the Alexa Top Websites [3]. To ensure the quality and consistency of our dataset, we filter out websites that are inaccessible, redirect to a different website, or contain offensive content. We detail the complete list of selected websites in Appendix A. For these 200 websites, we collect our profiling dataset with a total of 40,000 samples, labeling each sample with the corresponding website. This dataset consists of 20,000 samples collected in the default scheduling environment and another 20,000 samples collected under an MPS-enabled environment. For each sample, we collect 6,000 measurements by launching our attack primitive while accessing the corresponding website. We select this specific number of measurements per sample based on the longest access time among our candidates. To account for content updates over time, we capture samples for each website sequentially with approximately a five-second interval between samples. This process introduces approximately a four-day time gap between the first and last samples of each website. Additionally, we retain network delays without applying any post-processing to reflect the practical setting.

We analyze the time-slice measurements for three websites, `google.com`, `microsoft.com`, and `tiktok.com`, to identify distinguishable temporal patterns among them. Figure 12 presents 6,000 measurements per website, illustrating the

Table 4: Results of website fingerprinting attacks on Alexa top 200 websites under different scheduling environments

GPU	Scheduling Mechanism	Top-1 Accuracy (%)	Top-3 Accuracy (%)
RTX 2080 Ti	Default	92.09	96.19
	MPS-enabled	91.86	96.28
RTX 3080	Default	90.70	96.48
	MPS-enabled	93.60	97.63
RTX 4090	Default	93.55	97.71
	MPS-enabled	95.21	98.23
Average	Default	92.11	96.79
	MPS-enabled	93.56	97.38

temporal patterns introduced in time slices while rendering objects. For each website, we observe that the initial rendering spans a different number of measurements, and periodic patterns correspond to the dynamic object rendering on the webpage. In the measurements for the Google website, we observe periodic patterns that correspond to the blinking text cursor, following the initial rendering that spans fewer measurements compared to the other websites (Figures 12a and 12b). For the Microsoft website, we observe two burst patterns appearing after the initial webpage render, which are induced by two dynamic image loads during the measurements (Figures 12c and 12d). Compared to the other websites, the TikTok webpage shows the most consistent GPU utilization patterns across the measurements due to the video playing on the page (Figures 12e and 12f).

Attack evaluation. We implement a classifier using an LSTM model. In this classifier, we configure two LSTM layers, each with 256 hidden states. For the output layer, we use a fully connected layer with 200 neurons and a Softmax activation, which produces a probability distribution over the 200 websites. Using this classifier, we evaluate our attack using a 5-fold cross-validation scheme. For this, we first shuffle our dataset with a fixed random seed of 0. We then split the shuffled samples into five folds, evenly distributing the samples from all 200 website classes across the folds. To prevent data leakage between folds, we separately perform standardization within each training fold. In each iteration of cross-validation, we train the classifier for 300 epochs on the four training folds with a batch size of 64. For the training, we apply the Adam optimizer [38] and a binary cross-entropy with logits loss function [32]. We use an input window size of 20, a learning rate of 0.001, a weight decay of 0.2, and apply a 20% dropout for regularization [67]. At the end of each iteration, we perform validation using the remaining fold.

Table 4 summarizes the attack results across RTX 2080 Ti, RTX 3080, and RTX 4090 GPUs. We evaluate the effectiveness of our attack method by measuring the Top-1 and Top-3 accuracies under two scheduling environments. Each metric is measured by averaging the accuracies over the five folds. Across all GPUs, our attack achieves average Top-1 and Top-3 accuracies of 92.11% and 96.79% under default scheduling, and 93.56% and 97.38% under MPS-enabled scheduling. Notably, TIMESLICE-SANDWICH consistently achieves at least 90.70% in both Top-1 and Top-3 accuracies across GPUs, demonstrating its effectiveness for website fingerprinting on a practical web browser, even when MPS is enabled.

Table 5: Results of DNN model recovery attacks on an RTX 4090 GPU under noisy conditions

Noise Level*		F1 Score (%)	Precision (%)	Recall (%)
Profiling	Attack			
1	1	86.99	85.62	88.40
2	2	82.82	84.50	81.20
3	3	87.82	87.84	87.80
Consistent† (average)		85.87	85.99	85.80
1	2	75.39	76.84	74.00
	3	50.16	50.62	49.70
2	1	69.47	66.93	72.20
	3	70.84	72.13	69.60
3	1	61.88	60.62	63.19
	2	78.28	77.67	78.89
Different‡ (average)		67.67	67.47	67.93
Average		73.74	73.64	73.89

∗ The number of the Rodinia [10] `gaussian` benchmark applications co-running on the GPU.
† The noise level remains unchanged between the profiling and attack phases.
‡ The noise level differs between the profiling and attack phases.

5.4 Attacks Under Noisy Conditions

When multiple processes run on a GPU, their interleaved execution introduces varying levels of noise into the measurements captured by our attack primitive. To evaluate the applicability of our attack in such realistic environments, we perform the model recovery attack under noisy conditions using the same experimental setup and identical input scenario as in Section 5.2, where the victim selects the same images used in profiling. Specifically, on an RTX 4090 GPU, we vary the noise level by co-running one, two, or three instances of the `gaussian` application from the Rodinia benchmark [10]. We empirically filter out measurements with elapsed times exceeding 5.3 ms as outliers, removing approximately 0.08% of the collected measurements.

Table 5 summarizes the attack results. TIMESLICE-SANDWICH achieves at least 81% across all metrics when the noise level is consistent between the profiling and model recovery phases. In the worst case, when the noise level differs substantially between the two phases, performance degrades to approximately 50% across all metrics. This degradation indicates that our outlier filtering alone is insufficient to fully alleviate the impact of significantly fluctuating noise levels. Nevertheless, an attacker can estimate the level of background activity using user-accessible tools such as `nvidia-smi` (e.g., the number of active processes and GPU utilization) [49]. Using this information, the attacker can adapt their strategy to achieve successful model recovery in practice.

6 Discussion

In this section, we discuss the broader applicability of TIMESLICE-SANDWICH to other environments to further explore this newly identified GPU side-channel surface.

Attacks in web browsers. Applying our attack in web browser environments is expected to present practical challenges due to compiler optimizations and timing sources. For example, such a busy-wait GPU workload is unlikely to survive compile-time optimizations in browsers, even when initiated from JavaScript. In contrast, our native CUDA implementation prevents this elimination by compiling with the `-G` flag using the NVCC compiler. However, any settings that reduce or disable such optimizations are governed by the browser or administrator rather than attacker-controlled JavaScript code. Prior browser-based GPU attacks commonly rely on GPU counting threads that increment shared counters to obtain timing infromation [13, 16]. However, this timing source is inherently discontinuous when the GPU executes other workloads, making it unreliable for accurately measuring a victim's time slices. Although WebGPU may expose optional GPU timestamp queries (e.g., via `timestamp-query`) to JavaScript, their availability depends on the platform and browser configuration [72]. More importantly, even when such timestamp queries are available, the WebGPU API does not guarantee that these timestamps are aligned with GPU scheduling events. As a result, implementing our attack in a browser-based variant requires new mechanisms and is beyond the scope of this work.

GPUs from other vendors. Although we demonstrated our approach on NVIDIA GPUs, we expect the proposed methodology to be applicable to GPUs from other vendors. AMD and Intel provide comparable GPU programming APIs, such as AMD's Heterogeneous Computing Interface for Portability (HIP) [5] and Intel's oneAPI [25]. Our method can be implemented using their corresponding timing functions, such as `hipEventRecord()`, `hipEventElapsedTime()`, and `zeEventQueryKernelTimestamp()`, which operate on work queues [6, 24] analogous to CUDA streams. Furthermore, GPUs from AMD and Intel also employ time-sliced scheduling [26,59] (e.g., Intel drivers use a 5 ms time slice by default), indicating our findings can be extended to their platforms.

Attacks on multi-GPU systems. Our demonstrations are conducted on a system with a single NVIDIA GPU. However, TIMESLICE-SANDWICH can also be applied to multi-GPU environments that rely on NVIDIA drivers. In such systems, each GPU still employs time-sliced scheduling independently, unless scheduling is explicitly managed by higher-level frameworks, such as Kubernetes [51]. The CUDA runtime API provides functions like `cudaGetDeviceCount()` to query available GPUs and `cudaSetDevice()` to launch kernels on a specific device [55]. Using these APIs, an attacker can launch a spy process on the target GPU in a multi-GPU system and capture the victim program's time-slice patterns, with no substantial modifications to the proposed attack primitive.

7 Mitigation

Our attack introduces a new source of GPU side-channel leakage that does not primarily rely on specific shared resources. Existing mitigation against GPU side-channel attacks, such as cache partitioning or limiting access to performance counters [13,43], focus primarily on preventing precise monitoring of shared resources. Since our attack primitive exploits time-sliced scheduling, it bypasses such resource-centric mitigation approaches that have little impact on GPU scheduling behaviors. Restricting unprivileged access to `cudaEventRecord` or lowering its timing resolution can mitigate our attack. However, because the scheduling behavior underlying the side channel remains unchanged, this mitigation may be bypassed via emerging alternative timing methods. To effectively counter TIMESLICE-SANDWICH in practice, we outline mitigation strategies that address GPU scheduling behaviors from several aspects.

Detection-based defense mechanism. Countermeasures against microarchitectural side channels often incur significant performance overhead. Detection-based defense strategies can reduce this overhead by selectively enabling protections only when an attack is detected. Previous studies on detecting side-channel attacks that exploit resource contention on CPUs [33, 35] and GPUs [74] have relied on real-time detectors that monitor hardware event counts. Since such attacks repeatedly trigger specific hardware events to capture a victim's behavior, metrics related to shared resources, such as cache misses and retired instructions, can serve as useful indicators for detecting their presence.

Our attack, however, induces minimal activity on shared resources and is therefore unlikely to trigger such hardware events that could be monitored, limiting the effectiveness of similar detection-based approaches. Unlike benign programs such as web browsers or deep learning applications, the proposed attack primitive repeatedly launches fixed-duration kernels with minimal and consistent resource usage. Detecting such patterns may require the identification of new indicators beyond traditional hardware events.

Time-slicing policy. The effectiveness of TIMESLICE-SANDWICH relies on precisely identifying the upper bound of a time slice. Altering time-slicing policies can therefore hinder or prevent the construction of an effective primitive. For instance, NVIDIA DRIVE OS on Tegra GPUs allows configuration of three different time-slice upper bounds depending on workload priority in a specific environment [50]. In these environments, a detection-based defense mechanism could dynamically alter the enforced time-slice upper bound by modifying application priority. As a result, an attack primitive configured for the identified upper bound would fail to capture accurate time-slice patterns.

Another potential defense is to enforce strict equal-time scheduling across applications, preventing attackers from exploiting variable time-slice durations. For instance, NVIDIA's GPU Operator plugin for Kubernetes [51] supports time-sliced scheduling based on a logical unit called a replica, each mapped to a fixed-duration time slice. In this setup, each application runs on a pod, which is a container object of Kubernetes, under the provision of the GPU Operator framework. Assigning the same number of replicas to each application pod ensures that the slice durations are identical across all applications. Consequently, an attacker always observes constant slice durations, which prevents the accurate calibration of a busy-wait kernel.

Scheduling isolation. Our attack primitive relies on observing time-slice allocations that are globally visible across applications sharing the GPU. Isolating the scheduling environment between applications can prevent such cross-application leakage. Beginning with Ampere architectures, server-grade NVIDIA GPUs support Multi-Instance GPU (MIG) technology [58], a spatial partitioning mechanism for GPUs. Unlike MPS [48], MIG physically partitions a GPU into multiple instances, ensuring that tasks on one instance do not affect those on other instances [58]. As a result, an attacker cannot capture the time-slice patterns of a victim running in a separate instance. To leverage this in practice, we must strictly enforce tenant isolation and prevent the attacker from launching spy processes within the victim's MIG partition; otherwise, isolation guarantees can be bypassed.

8 Related Work

This section surveys prior work on microarchitectural side-channel attacks against GPUs and examines research on scheduling mechanisms in the context of side-channel vulnerabilities.

Hardware performance counters. Naghibijouybari et al. proposed side-channel attacks that leveraged monitoring tools, such as a memory management API and a performance counter profiler called CUPTI [42]. They measured variations in performance counters caused by a victim co-located on the same GPU, demonstrating keystroke recovery, website fingerprinting, and inference of the number of neurons in the Rodina backpropagation benchmark. In response to these attacks, NVIDIA restricted access to the counters to privileged users [43]. Wei et al. later presented another CUPTI-based attack by downgrading the graphics driver to an unpatched version on a VM, thereby bypassing this patch [70]. Using the TensorFlow timeline profiler and CUPTI, they inferred hyperparameters from the training of MLP, ZFNet, and VGG16 models running on another VM that shared the same GPU.

Cache eviction. Prime+Probe and cache occupancy channel attacks, which have long been studied on CPUs, are classic representatives of side-channel attacks through cache eviction [37, 65]. Prime+Probe measures eviction patterns in specific cache sets, while the cache occupancy channel monitors activity across the entire cache. Recent studies have adapted these techniques to GPUs in substantially different microarchitectural contexts [13, 16, 76]. Zhang et al. proposed a timer-free method that uses the `discard` PTX instruction to observe eviction patterns in cache sets, demonstrating website fingerprinting and keystroke recovery attacks [76]. Giner et al. exploited the WebGPU API to implement a thread-based counter for timing and construct eviction sets, demonstrating keystroke recovery, secret key extraction from a CUDA AES implementation, and covert channel attacks [16]. Similarly, Ferguson et al. used WebGPU to build cache occupancy channels for website fingerprinting, employing a timing method based on a thread-based counter [13].

Scheduling mechanisms. Prior studies on CPUs investigated contention in scheduler queues as a source of side-channel leakage by priming queues with specific instructions [14, 15]. However, these attacks rely heavily on CPU-specific features, such as simultaneous multi-threading (SMT) [15] and out-of-order execution [14], which makes them difficult to extend to GPUs. In contrast, our work leverages GPU scheduling behaviors, which are not readily observed in CPUs. These fundamental microarchitectural differences underscore the distinct contributions of this study. Prior work on GPU side-channel attacks explored time-sliced scheduling to support the attack process within the allocated time slices [76], thereby achieving higher sampling rates of performance counter readings [70] or confirming attacker-victim co-location [42]. However, none of these studies examined the GPU scheduling mechanism itself as a direct source of side-channel leakage, which is the central focus of this study.

9 Conclusion

In this paper, we uncovered the scheduling policies of NVIDIA GPUs and presented TIMESLICE-SANDWICH, a novel side-channel attack exploiting GPU time-sliced scheduling behavior. Our attack primitive measures timing differences introduced by the GPU hardware scheduler during time slice allocation for victim tasks. The resulting time-slice patterns enable attackers to exfiltrate sensitive information from co-located applications without depending on shared-resource contention. We demonstrated the efficacy and security impact of TIMESLICE-SANDWICH through two practical attack scenarios: DNN model recovery and website fingerprinting. To the best of our knowledge, this is the first work to explore GPU time-sliced scheduling as a direct side channel, demonstrate its efficacy, and discuss its mitigation strategies.

Acknowledgments

This work was supported by NSF CAREER Award CCF-2146475. This work was also supported by Institute of Infor-

mation & communications Technology Planning & Evaluation (IITP) grant (No.2022-0-00411) and National Research Foundation of Korea (NRF) grant (RS-2025-00563143, RS-2023-NR077166, RS-2021-NR060143) funded by the Korea government (MSIT).

Ethical Considerations

Stakeholders. Our attack targets environments where a GPU is shared among multiple users, such as cloud platforms and virtualized desktop environments. We identify three key stakeholders that may be affected by this attack: ① cloud service providers, who may face service disruptions and associated reputational damage; ② DNN model developers, whose proprietary models could be exposed; and ③ GPU users, who may unintentionally leak sensitive information during execution. To strengthen security against such threats, we provide several mitigation strategies in Section 7 that address the root causes of the attack and can be readily applied to current GPU computing systems. These strategies include controlling GPU scheduling policies with Kubernetes [51] and isolating scheduling environments from other GPU workloads through NVIDIA MIG [58].

Dual-use concerns. Since our findings expose a previously unexplored attack surface, they carry an inherent risk of malicious misuse. To minimize potential harm, all experiments in this study were conducted on strictly controlled, in-house systems without involving real-world cloud deployments or external users. In line with responsible research practices, we have reported the proposed attack to NVIDIA. While they acknowledged our findings, NVIDIA also noted that this threat, like other microarchitectural side-channel attacks, is difficult to eliminate entirely without incurring significant performance costs in practical deployments. In their response, NVIDIA recommended that users operating in highly privacy-sensitive environments adopt workload isolation techniques available on recent server-grade GPUs, such as MIG [58] and Confidential Compute [44], to prevent untrusted workloads from sharing the same GPU. To maximize research utility while addressing dual-use concerns, we will make our results and code publicly available, excluding specific components that could enable sensitive information leakage if misused. With the precautions taken throughout this study, our findings can improve the security of GPU computing systems and motivate future research on scheduling-aware attacks and defenses. Accordingly, we believe these benefits outweigh the residual risks.

Open Science

To facilitate reproducibility and encourage further research, we make our code available in a publicly accessible repository at https://doi.org/10.5281/zenodo.17987226. This repository provides our attack implementations and measurement-collection scripts, excluding specific components that may facilitate sensitive information leakage, to minimize potential misuse (e.g., classifiers trained for model recovery and website fingerprinting). These artifacts enable time-slice measurements for identifying the time-slice upper bound (Sections 3.2 and 3.3), profiling the training of candidate models (Section 5.2), and capturing the rendering activity of candidate websites (Section 5.3). We describe detailed experimental settings in this paper, including classifier designs and training parameters, to reconstruct the excluded components for research purposes (Sections 5.2 and 5.3).

References

[1] Ahmed A Abdelrahman, Mohamed M Fouad, Hisham Dahshan, and Ahmed M Mousa. High performance cuda aes implementation: A quantitative performance analysis approach. In *2017 Computing Conference*, pages 1077–1085. IEEE, 2017.

[2] Jaeguk Ahn, Cheolgyu Jin, Jiho Kim, Minsoo Rhu, Yunsi Fei, David Kaeli, and John Kim. Trident: A hybrid correlation-collision gpu cache timing attack for aes key recovery. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 332–344. IEEE, 2021.

[3] Alexa. Alexa top websites, 2023. https://www.expireddomains.net/alexa-top-websites/ (Accessed: 2026-01-01).

[4] Amazon Web Services. Amazon elastic compute cloud user guide - nvidia drivers for your amazon ec2 instance, 2025. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-nvidia-driver.html (Accessed: 2026-01-01).

[5] AMD. HIP documentation, 2025. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html (Accessed: 2026-01-01).

[6] AMD. HIP documentation - HIP runtime api - modules - stream management, 2025. https://rocm.docs.amd.com/projects/HIP/en/latest/reference/hip_runtime_api/modules/stream_management.html (Accessed: 2026-01-01).

[7] Joshua Bakita and James H Anderson. Demystifying nvidia gpu internals to enable reliable gpu management. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 294–305. IEEE, 2024.

[8] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur,

Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 316–331, 2018.

[9] Ebubekir Buber and DIRI Banu. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6. IEEE, 2018.

[10] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[11] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

[12] Samuel H Duncan, Lacky V Shah, Sean J Treichler, Daniel Elliot Wexler, Jerome F Duluk Jr, Philip Browning Johnson, and Jonathon Stuart Ramsay Evans. Concurrent execution of independent streams in multi-channel time slice groups, September 13 2016. US Patent 9,442,759.

[13] Ethan Ferguson, Adam Wilson, and Hoda Naghibijouybari. Webgpu-spy: Finding fingerprints in the sandbox through gpu cache attacks. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 158–171, 2024.

[14] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote scheduler contention attacks. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 365–383. Springer, 2024.

[15] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. Squip: Exploiting the scheduler queue contention side channel. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2256–2272. IEEE, 2023.

[16] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. Generic and automated drive-by gpu cache attacks from the browser. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 128–140, 2024.

[17] Google. Chrome web browser. https://www.google.com/chrome/ (Accessed: 2026-01-01).

[18] Google. About skialskia, 2024. https://skia.org/docs/ (Accessed: 2026-01-01).

[19] Khronos Group. Opencl overview, 2025. https://www.khronos.org/opencl/ (Accessed: 2026-01-01).

[20] Khronos Group. Opengl - the industry's foundation for high performance graphics, 2025. https://www.opengl.org/ (Accessed: 2026-01-01).

[21] Khronos Group. Vulkan | cross platform 3d graphics, 2025. https://www.vulkan.org/ (Accessed: 2026-01-01).

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[23] Tsuyoshi Ichimura, Kohei Fujita, Takuma Yamaguchi, Akira Naruse, Jack C Wells, Thomas C Schulthess, Tjerk P Straatsma, Christopher J Zimmer, Maxime Martinasso, Kengo Nakajima, et al. A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 627–637. IEEE, 2018.

[24] Intel. Level zero specification documentation - core api - cmdqueue, 2024. https://oneapi-src.github.io/level-zero-spec/level-zero/1.11/core/api.html#cmdqueue (Accessed: 2026-01-01).

[25] Intel. oneAPI: A viable alternative to cuda* lock-in, 2024. https://www.intel.com/content/www/us/en/developer/articles/technical/oneapi-a-viable-alternative-to-cuda-lock-in.html (Accessed: 2026-01-01).

[26] Intel. oneAPI gpu optimization guide - time slice considerations, 2024. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/configuring-gpu-device.html#TIME-SLICE-CONSIDERATIONS (Accessed: 2026-01-01).

[27] Paul Irish and Paul Lewis. Developer faq - why blink?, 2013. https://www.chromium.org/blink/developer-faq/ (Accessed: 2026-01-01).

[28] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In *2016 IEEE International Symposium on High Performance Computer architecture (HPCA)*, pages 394–405. IEEE, 2016.

[29] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 167–172, 2017.

[30] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. Exploiting bank conflict-based side-channel timing leakage of gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–24, 2019.

[31] Zahra Karevan and Johan AK Suykens. Transductive lstm for time-series prediction: An application to weather forecasting. *Neural Networks*, 125:1–9, 2020.

[32] Gil Keren, Sivan Sabato, and Björn Schuller. Fast single-class classification and the principle of logit separation. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 227–236. IEEE, 2018.

[33] Hodong Kim, Changhee Hahn, Hyunwoo J Kim, Youngjoo Shin, and Junbeom Hur. Deep learning based detection for multiple cache side-channel attacks. *IEEE Transactions on Information Forensics and Security*, 2023.

[34] Taehun Kim and Youngjoo Shin. Gpu side-channel attacks are everywhere: A survey. In *2020 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4. IEEE, 2020.

[35] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. Sok: Can we really detect cache side-channel attacks by monitoring performance counters? In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 172–185, 2024.

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

[37] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 605–622. IEEE, 2015.

[38] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019.

[39] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[40] Sparsh Mittal, SB Abhinaya, Manish Reddy, and Irfan Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2:266–285, 2018.

[41] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. Microarchitectural attacks in heterogeneous systems: A survey. *ACM Computing Surveys*, 55(7):1–40, 2022.

[42] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2139–2153, 2018.

[43] NVIDIA. Nvidia development tools solutions - CUPTI_ERROR_INSUFFICIENT_PRIVILEGES: Cupti permission issue with performance counters, 2018. https://developer.nvidia.com/nvidia-development-tools-solutions-err-nvgpuctrperm-cupti (Accessed: 2026-01-01).

[44] NVIDIA. Confidential compute on nvidia hopper h100, 2023. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf (Accessed: 2026-01-01).

[45] NVIDIA. Cuda samples v12.3 - samples for cuda developers which demonstrates features in cuda toolkit, 2023. https://github.com/NVIDIA/cuda-samples (Accessed: 2026-01-01).

[46] NVIDIA. Matrix multiplication background user's guide, 2023. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication (Accessed: 2026-01-01).

[47] NVIDIA. Cuda toolkit v12.6.1 - cuda runtime api - 6.6. event management, 2024. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html (Accessed: 2026-01-01).

[48] NVIDIA. Multi-process service, 2024. https://docs.nvidia.com/deploy/mps/ (Accessed: 2026-01-01).

[49] NVIDIA. System Management Interface SMI, 2024. https://developer.nvidia.com/system-management-interface (Accessed: 2026-01-01).

[50] NVIDIA. Tegra gpu scheduling improvements, 2024. https://developer.nvidia.com/docs/drive/

drive-os/6.0.10/public/drive-os-linux-sdk/common/topics/graphics_content/SettingtheTimeslice4.html (Accessed: 2026-01-01).

[51] NVIDIA. Time-slicing gpus in kubernetes, 2024. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html#about-configuring-gpu-time-slicing (Accessed: 2026-01-01).

[52] NVIDIA. Cuda c++ best practices guide - 10.6. multiple contexts, 2025. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#multiple-contexts (Accessed: 2026-01-01).

[53] NVIDIA. Cuda c++ programming guide - 3.2.8.5. streams, 2025. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#streams (Accessed: 2026-01-01).

[54] NVIDIA. Cuda toolkit - free tools and training, 2025. https://developer.nvidia.com/cuda-toolkit (Accessed: 2026-01-01).

[55] NVIDIA. Cuda toolkit v12.6.1 - cuda runtime api - 4.1. device management, 2025. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html (Accessed: 2026-01-01).

[56] NVIDIA. Nvidia accelerated linux graphics driver readme and installation guide - chapter 5. listing of installed components, 2025. https://download.nvidia.com/XFree86/Linux-x86_64/570.144/README/installedcomponents.html (Accessed: 2026-01-01).

[57] NVIDIA. Nvidia cuda compiler driver nvcc - 4.2.3.3. –device-debug (-G), 2025. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html?highlight=G#device-debug-g (Accessed: 2026-01-01).

[58] NVIDIA. Nvidia multi-instance gpu, 2025. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/introduction.html#introduction (Accessed: 2026-01-01).

[59] Nathan Otterness and James H Anderson. Exploring amd gpu scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, pages 24–34, 2021.

[60] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C Stern, and Artem Cherkasov. The transformational role of gpu computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, 2022.

[61] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414. IEEE, 2018.

[62] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

[63] PyTorch. Imagenet training in pytorch. https://github.com/pytorch/examples/tree/main/imagenet (Accessed: 2026-01-01).

[64] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[65] Anatoly Shusterman, Zohar Avraham, Eliezer Croitoru, Yarden Haskal, Lachlan Kang, Dvir Levi, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Website fingerprinting through the cache occupancy channel and its real world practicality. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2042–2060, 2020.

[66] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[67] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[68] Statista. Graphics add-in-board (AIB) supplier shipment share worldwide 2010 to 2024, by quarter, 2025. https://www.statista.com/statistics/274005/market-share-of-global-graphics-card-shipments-since-3rd-quarter-2010/ (Accessed: 2026-01-01).

[69] Ray Toal. Opengl examples - spinning square. https://cs.lmu.edu/~ray/notes/openglexamples/ (Accessed: 2026-01-01).

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

[71] World Wide Web Consortium. Webgpu, 2025. https://www.w3.org/TR/webgpu/ (Accessed: 2026-01-01).

[72] World Wide Web Consortium. Webgpu- 20.4. timestamp query, 2025. https://www.w3.org/TR/webgpu/#timestamp (Accessed: 2026-01-01).

[73] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1492–1500, 2017.

[74] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, pages 497–509, 2019.

[75] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

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

A Target websites for fingerprinting attack

After filtering out 124 inaccessible or offensive websites, we selected the 200 websites listed in Table 6 from the Alexa Top Websites [3] for our website fingerprinting attack.

Table 6: List of target websites

1	google.com	41	adobe.com	81	ok.ru	121	yandex.com	161	scribd.com
2	youtube.com	42	telegram.org	82	eastmoney.com	122	deviantart.com	162	craigslist.org
3	baidu.com	43	quora.com	83	shopify.com	123	indiatimes.com	163	zol.com.cn
4	bilibili.com	44	stackoverflow.com	84	toutiao.com	124	hubspot.com	164	steampowered.com
5	qq.com	45	sohu.com	85	bbc.com	125	quillbot.com	165	hdfcbank.com
6	twitter.com	46	spotify.com	86	okta.com	126	uol.com.br	166	pconline.com.cn
7	wikipedia.org	47	1688.com	87	digikala.com	127	dailymail.co.uk	167	1337x.to
8	amazon.com	48	tmall.com	88	w3schools.com	128	pexels.com	168	9gag.com
9	instagram.com	49	indeed.com	89	medium.com	129	yuque.com	169	asana.com
10	linkedin.com	50	deepl.com	90	so.com	130	ifeng.com	170	www.gov.uk
11	whatsapp.com	51	pixiv.net	91	fc2.com	131	zoho.com	171	360.com
12	openai.com	52	nytimes.com	92	cnn.com	132	samsung.com	172	dell.com
13	yahoo.com	53	duckduckgo.com	93	weather.com	133	theguardian.com	173	wikimedia.org
14	bing.com	54	freepik.com	94	udemy.com	134	remove.bg	174	cupfox.app
15	taobao.com	55	booking.com	95	gitee.com	135	yts.mx	175	wellsfargo.com
16	notion.so	56	imgur.com	96	quizlet.com	136	bankofamerica.com	176	similarweb.com
17	microsoft.com	57	jianshu.com	97	archive.org	137	daum.net	177	xueqiu.com
18	vk.com	58	ilovepdf.com	98	intuit.com	138	greasyfork.org	178	blog.jp
19	zoom.us	59	twitch.tv	99	speedtest.net	139	coinmarketcap.com	179	chegg.com
20	github.com	60	atlassian.net	100	mega.nz	140	eastday.com	180	myanimelist.net
21	jd.com	61	fiverr.com	101	researchgate.net	141	yiyouliao.com	181	varzesh3.com
22	weibo.com	62	dropbox.com	102	wetransfer.com	142	investing.com	182	dailymotion.com
23	tiktok.com	63	office365.com	103	vimeo.com	143	gosuslugi.ru	183	wix.com
24	canva.com	64	discord.com	104	chase.com	144	shein.com	184	foxnews.com
25	fandom.com	65	namu.wiki	105	rakuten.co.jp	145	blogger.com	185	patreon.com
26	office.com	66	nih.gov	106	binance.com	146	chess.com	186	reverso.net
27	trello.com	67	avito.ru	107	walmart.com	147	envato.com	187	book118.com
28	naver.com	68	3dmgame.com	108	behance.net	148	line.me	188	kakao.com
29	apple.com	69	instructure.com	109	grammarly.com	149	shopee.tw	189	huawei.com
30	sina.com.cn	70	flipkart.com	110	51cto.com	150	nicovideo.jp	190	geeksforgeeks.org
31	aliexpress.com	71	alibaba.com	111	upwork.com	151	coupang.com	191	ups.com
32	paypal.com	72	cnki.net	112	fmkorea.com	152	nvidia.com	192	cambridge.org
33	iqiyi.com	73	mediafire.com	113	ali213.net	153	aparat.com	193	mi.com
34	pinterest.com	74	tistory.com	114	salesforce.com	154	animeflv.net	194	nexon.com
35	mail.ru	75	figma.com	115	espn.com	155	mercadolibre.com.ar	195	forbes.com
36	ebay.com	76	smzdm.com	116	shutterstock.com	156	crunchyroll.com	196	hp.com
37	douban.com	77	tumblr.com	117	zendesk.com	157	coursera.org	197	usps.com
38	msn.com	78	youdao.com	118	savefrom.net	158	unsplash.com	198	wordpress.org
39	imdb.com	79	52pojie.cn	119	mozilla.org	159	v2ex.com	199	gitlab.com
40	netflix.com	80	sogou.com	120	zillow.com	160	soundcloud.com	200	healthline.com