Exploiting TLBs in Virtualized GPUs for Cross-VM Side-Channel Attacks

Hongyue Jin
Clemson University
hongyue@g.clemson.edu

Yanan Guo
University of Rochester
yguo51@cs.rochester.edu

Zhenkai Zhang
Clemson University
zhenkai@clemson.edu

***Abstract*—With the growing adoption of virtualized GPUs in cloud computing, the potential security implications associated with GPU sharing among multiple tenants have largely been overlooked. This paper takes a foundational step in revealing these risks by investigating information leakage through GPU microarchitectural components. Specifically, we develop a `Prime+Probe` attack primitive tailored to the translation lookaside buffers (TLBs) in virtualized NVIDIA GPUs. We discuss several unique challenges posed by the GPU virtualization environment and demonstrate how our design effectively overcomes them. Leveraging this primitive, we conduct two cross-VM side-channel attack case studies in a cloud setting: a cheating exploit in the game Counter-Strike 2 that reveals hidden opponents and a website fingerprinting attack that identifies web pages browsed by users of virtual desktops. To the best of our knowledge, these are the first side-channel attacks demonstrated against virtualized GPUs in cloud settings, highlighting previously unknown security risks that warrant further investigation.**

I. Introduction

Modern graphics processing units (GPUs) have evolved from fixed-function rendering pipelines into fully programmable parallel processors capable of accelerating a diverse range of workloads. In tandem with this evolution, there has been a rapid shift towards the adoption of such devices in data centers. As of now, a large number of cloud service providers (CSPs) around the world have made GPU-powered virtual machines (VMs) available to their customers.

Initially, GPU offerings in the cloud were limited to dedicated instances. Driven by the objectives of reducing operational costs and increasing price competitiveness, many CSPs have recently started relying on state-of-the-art virtualization technologies (e.g., NVIDIA vGPU [1]) to efficiently allocate and share GPU resources among multiple users. Examples of major CSPs that offer VMs powered by virtualized GPUs include Microsoft Azure [2], Vultr [3], Alibaba Cloud [4], and Tencent Cloud [5].

Certainly, given the economic advantages and reliable performance, tenants who do not need access to a dedicated GPU also find it appealing to share a virtualized GPU in the cloud. However, despite the considerable benefits and broad deployment of GPU virtualization, its security implications, especially potential information leakage, have rarely been investigated. Such an oversight may be problematic as various use cases of virtualized GPUs actually involve sensitive information from the tenants, and it can lead to unanticipated breaches of confidentiality.

For instance, Desktop-as-a-Service (DaaS) solutions, which have lately gained significant popularity in the cloud computing market, often incorporate virtualized GPUs to enhance user experience and cater to visually demanding applications [6]. Like personal computers, the virtual desktops of DaaS users typically contain a wealth of private information, and precautions should be taken if information leakage via virtualized GPUs is possible. This and other similar scenarios highlight the necessity to start studying any conceivably overlooked implications.

In this paper, we present the first investigation into this matter through the lens of microarchitectural components in virtualized NVIDIA GPUs, which are widely deployed in clouds. Specifically, we focus on exploiting the translation lookaside buffers (TLBs) in these GPUs to mount cross-VM side-channel attacks. To this end, a `Prime+Probe` attack primitive tailored to such a component is needed. Even though `Prime+Probe` is a well-established technique, we find that formulating an effective primitive for virtualized GPU environments is actually very challenging.

One major technical hurdle lies in how to precisely manipulate GPU TLB states. This, in turn, requires two key capabilities: (1) deriving the TLB set hash functions, and (2) obtaining enough GPU memory pages to populate the targeted TLB sets. While prior work has demonstrated how to reverse-engineer the TLB set hash functions in NVIDIA GPUs [7], achieving the second capability is in fact non-trivial under the memory management constraints of the NVIDIA vGPU runtime. (vGPU is the technology for NVIDIA GPU virtualization [1].) Specifically, vGPU restricts CUDA applications to allocating memory only in 2 MB pages. With this page size and a modest portion of GPU memory granted to each VM, it becomes extremely difficult to fully evict any TLB sets at the lower levels of the GPU TLB hierarchy in CUDA.

Beyond the challenge introduced by vGPU's memory management constraints, the execution model of vGPU presents additional complexities that are not encountered in traditional

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231480
www.ndss-symposium.org

`Prime+Probe` scenarios. Under vGPU, GPU contexts from different VMs never run in parallel; instead, they are scheduled in a time-sharing fashion with millisecond-scale quanta. Hence, priming and probing operations should be carefully orchestrated to straddle the vGPU time slice boundaries and remain isolated from each other. Moreover, a vGPU time slice allows a graphics-intensive workload to exercise many TLB sets, so a simple "was this set accessed?" metric may reveal little victim information.

Further to the intricacies already discussed, simultaneously monitoring many TLB sets also demands a list of GPU-specific refinements. For instance, the assignment of threads for priming and probing targeted TLB sets needs careful planning due to the GPU's single instruction multiple thread execution paradigm, the hierarchical properties of the GPU TLB structure, and the CUDA thread block distribution policy; otherwise, threads may interfere with each other and thus distort timing measurements. Even with a proper thread-to-TLB set arrangement, the probing phase still needs more coordination, as we discover that launching too many probes at once creates spurious TLB misses.

In this work, we have addressed all these challenges and developed an effective `Prime+Probe` attack primitive for monitoring GPU TLB access patterns under vGPU settings commonly employed in clouds. Although there has been prior work applying `Prime+Probe` to GPU TLBs [7], we should highlight that many of the aforementioned challenges do not exist in the previously studied environments; hence our design actually differs substantially from earlier efforts (see Section VIII for details). To demonstrate the practicality of the primitive, we perform the first-ever cross-VM side-channel attacks on virtualized GPUs. Our findings emphasize the need for further investigation into the security implications of GPU sharing, a topic that has so far received insufficient attention.

The main contributions of this work include:

1) We have developed a method for reliably constructing TLB eviction sets despite the memory management constraints imposed by the vGPU runtime. In particular, we discover that GPU memory allocated through graphics rendering APIs like Vulkan uses 64KB pages and maintains this page size when imported into CUDA. This allows us to overcome the technical barriers to precisely manipulating TLB states in vGPU setups.
2) We have formulated a `Prime+Probe` attack primitive tailored for TLBs in virtualized NVIDIA GPUs. The primitive addresses several unique challenges in the vGPU environment, including orchestrating parallel monitoring of multiple TLB sets while accounting for time-sharing execution of GPU contexts, achieving proper synchronization between thread blocks to avoid interference during priming and probing operations, and reliably measuring fine-grained contention patterns in TLB sets.
3) We have demonstrated the practicality of our attack primitive using two case studies. The first one is a cheating exploit in the popular esports game Counter-Strike 2 that allows detecting hidden opponents, showing that even with modern occlusion culling techniques, sensitive information can still leak through GPU TLB access patterns. The second one is a website fingerprinting attack, capable of identifying web pages browsed by users of virtual desktops with an accuracy of up to 91%.

To our knowledge, this is the first work on microarchitectural side-channel attacks targeting virtualized GPUs currently available in real public cloud environments.

Responsible disclosure: We have disclosed our findings to NVIDIA and shared a proof-of-concept (PoC) with its development team. NVIDIA initially placed an embargo on the disclosure, yet lifted it on March 21, 2025, after determining that the issue is not a bug. NVIDIA also mentioned that TLB isolation will be enhanced in its newer GPU architectures.

Availability: The PoC implementation of our attack primitive is available at https://github.com/0x5ec1ab/vgpu-tlb-exploit.

II. Background

In this section, we briefly describe the architecture of modern GPUs, their programming, and their virtualization. To maintain clarity, we focus on elements essential to understanding our work.

A. GPU Architecture

In NVIDIA GPUs, compute resources are organized hierarchically. At the top level of the structure are several Graphics Processing Clusters (GPCs). Each GPC comprises multiple Texture Processing Clusters (TPCs), and each TPC houses two Streaming Multiprocessors (SMs). SMs are the fundamental compute units in GPUs. Each SM is highly multi-threaded and executes threads in groups called warps, following the Single Instruction, Multiple Threads (SIMT) model (i.e., the threads in a warp execute in lockstep). On NVIDIA GPUs, each warp consists of 32 threads. A hardware scheduler within each SM switches the execution of multiple warps in a fine-grained manner.

GPUs are equipped with dedicated on-board memory, independent of the CPU's main memory. Prior to the execution of a GPU program, its code and data are first copied to the GPU's on-board memory. GPU memory is virtualized using paging. SMs generate virtual addresses, which are translated to physical addresses by the GPU's memory management unit (MMU) using page tables set up by the GPU driver. Each running GPU program, termed as a GPU context, has its own page table. When translating virtual addresses, the MMU walks through the page tables.

Page table walks are expensive, and like its counterpart in CPUs, the MMU in GPUs employs TLBs to cache recently used translations to avoid many such walks. In [7], Zhang *et al.* revealed that NVIDIA GPUs feature three levels of TLBs: Each TPC has an L1 iTLB and an L1 dTLB for instruction and data translations, respectively. Each GPC has a unified L2 TLB, and all GPCs share a unified L3 TLB. Both L1 TLBs are fully-associative with 16 entries each, whereas the L2 and L3 TLBs are 8-way set-associative. Interestingly, each entry in the L2 and L3 TLBs further contains 16 sub-entries, effectively increasing their reach. The least recently used (LRU) replacement policy is used at all TLB levels (L1, L2, and L3).

B. GPU Programming

GPUs can be programmed in two primary ways: through traditional graphics rendering APIs and via general-purpose computing frameworks. Graphics rendering APIs, such as OpenGL [8], DirectX [9], and Vulkan [10], enable developers to write shader programs. Shader programs are specialized pieces of code that run on a GPU and handle tasks ranging from transforming 3D vertex coordinates to applying textures and producing visual effects. In essence, they define how graphics are processed and rendered on the screen.

Alternatively, GPUs can be leveraged for general-purpose computing using frameworks like CUDA [11] and OpenCL [12]. These frameworks allow developers to write parallel programs for a wide range of computational tasks beyond graphics rendering. In this approach, computations are defined in functions called kernels. When a kernel is launched, it executes as a configured grid of thread blocks, with each block containing a specified number of threads that operate concurrently. The framework's runtime system automatically distributes the thread blocks across the GPU's SMs for parallel execution. Currently, the maximum number of threads per thread block is 1024 (i.e., 32 warps). Note that compute shaders in graphics rendering APIs can also perform arbitrary computations; however, they are more cumbersome to use than these frameworks.

C. GPU Virtualization

A number of CSPs have adopted NVIDIA's virtual GPU (vGPU) technology in their clouds to enable GPU sharing among tenants. As illustrated in Figure 1, the vGPU architecture centers upon a manager program running in the hypervisor. This manager creates vGPU instances and partitions GPU memory for them. Each vGPU instance can be assigned to a VM, and from the viewpoint of the VM, the instance appears like a directly attached physical GPU.

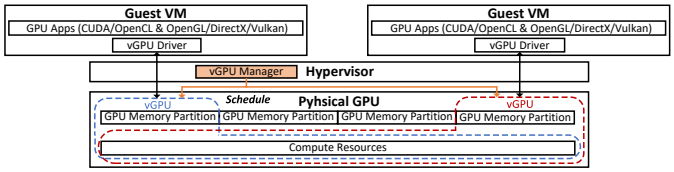


Fig. 1: NVIDIA vGPU architecture.

Although vGPU enables multiple tenants to share a GPU, it is important to note that the execution of GPU programs in VMs is not carried out in parallel but rather in a time-sharing manner. Fundamentally, there is a scheduler that allocates time slices to each VM's vGPU instance, and during the allocated periods, the GPU contexts in the corresponding VM gain exclusive access to the underlying GPU compute resources. Overall, three scheduling policies are supported [1]:

- *Best effort*, which enables vGPU instances to compete for available GPU processing cycles.
- *Equal share*, which equally assigns GPU processing cycles to each turned-on vGPU instance.
- *Fixed share*, which allocates a constant proportion of GPU processing cycles to a vGPU instance as per the maximum number of such instances that can be created.

Unless explicitly configured in the hypervisor to adopt equal or fixed share, the best effort policy is used by default. Appendix B provides some examples to elucidate these policies.

Note that NVIDIA also offers another virtualization technology called multi-instance GPU (MIG) for some of their high-end GPUs like A100 and H100. However, MIG-created GPU instances currently do not support graphics rendering APIs [13], and thus they are primarily used for DNN training and high-performance computing workloads, but not suitable for use cases like virtual desktops. In this paper, we focus on vGPU technology since it supports virtualization across all server-grade GPUs (including A100 and H100) and has been widely adopted in public cloud environments.

III. Threat Model

Our focus is on cloud environments where users can rent vGPU-powered VMs. This service model has become prevalent in practice, with notable examples including Microsoft Azure's NVadsA10-v5 series [2], Alibaba Cloud's sgn/vgn instance families [4], Tencent Cloud's GNV4v rendering instances [5], and Vultr's shared GPU offerings [3].

A legitimate user, whom we refer to as the victim, leverages such a VM to run GPU-accelerated applications like virtual desktops or cloud gaming. An attacker in our model is a malicious user who operates within the same cloud environment. We assume that the attacker achieves GPU co-residency with the victim (i.e., the vGPUs assigned to them are instances of the same physical GPU). The attacker has no special privileges beyond standard control over their own VM. We do not assume the existence of any software vulnerabilities or access to shared resources other than the GPU. The attacker's objective is to steal sensitive information from the victim through the victim's use of the virtualized GPU.

IV. Attack Overview

Under the outlined threat model, we aim to exploit GPU TLBs to mount cross-VM side-channel attacks. Specifically, we develop a `Prime+Probe`-based technique that enables us to effectively manipulate and monitor GPU TLB states to infer sensitive information across VM boundaries. In the following, we first justify our choice of the GPU TLB as the attack vector, and then we discuss several unique challenges that need to be addressed when implementing the corresponding `Prime+Probe` attack primitive.

A. Why GPU TLBs?

The rationale behind leveraging the GPU TLB side channel stems from vGPU's operational characteristics: (1) Since vGPU employs time-sharing execution, where GPU contexts from different VMs cannot run in parallel, stateless side channels that rely on simultaneous resource contention (e.g., the NoC [14] or PCIe [15]) are ruled out, making stateful channels necessary; (2) While both GPU caches [16] and TLBs [7] can serve as

stateful side channels due to their persistent nature, GPU TLBs can offer more stable and less noisy information leakage in our scenario for multiple reasons.

First, GPU contexts typically operate on large datasets that can easily overwhelm GPU data caches, particularly given that a VM's allocated time slice often spans milliseconds, long enough for GPU contexts to access substantial amounts of data. In contrast, GPU TLBs are designed with extensive reach [7], making them far more resilient to capacity pressure. Second, GPU caches are physically addressed and their set hash functions are highly non-linear and remain unknown despite reverse-engineering attempts, whereas GPU TLBs are virtually addressed with fully-revealed hash functions, giving attackers greater convenience and control.

Furthermore, we discover that GPU contexts in the vGPU setting retain fixed virtual addresses, regardless of whether address space layout randomization (ASLR) is enabled. While ASLR can randomize addresses in native GPU environments, our experiments show that on vGPUs, each GPU program consistently uses the same virtual addresses. This benefits attackers as the victim's GPU TLB access pattern becomes more deterministic. Thus, GPU TLBs are preferred over GPU caches for side-channel attacks in vGPU environments.

B. Challenges

To enable cross-VM side-channel attacks that exploit TLBs in virtualized GPUs, we devise a `Prime+Probe` primitive. While the basic concept follows the traditional `Prime+Probe` methodology [17], [18], implementing such a primitive under the vGPU setting poses several unique challenges:

- As with other `Prime+Probe` attack primitives, ours requires the ability to construct proper eviction sets. While a GPU's TLB set hash functions can be fully reverse-engineered, the real difficulty lies in the runtime constraints imposed by vGPU and intrinsic safeguards, which make it infeasible to reliably fill arbitrary target sets within a single programming paradigm (see Section V-A).
- Our `Prime+Probe` primitive aims to simultaneously monitor all sets of the last-level TLB. While the GPU's massive threads allow parallel manipulation of multiple sets, the special properties of GPU TLBs and the vGPU's time-sharing execution model demand careful consideration during priming and probing operations, including their preparation, orchestration, and synchronization (see Section V-B).
- A single vGPU time slice is typically long enough for graphics-intensive workloads to access many TLB sets, making it difficult to derive meaningful information solely by determining whether a set is accessed or not through `Prime+Probe`. Moreover, cache activities during each vGPU time slice significantly complicate the reliable determination of TLB access patterns (see Section V-C).

V. Attack Primitive Construction

In this section, we present the construction of our attack primitive that enables `Prime+Probe` on GPU TLBs in the vGPU-supported cloud environment. To facilitate our discussion, we use the NVIDIA A10 GPU as our running example, as it has become the *de facto* choice for vGPU offerings in practice [2], [4], [5]. Utilizing the toolset from [7], we reverse-engineered the TLB structure of the A10 GPU, which is illustrated in Figure 2.

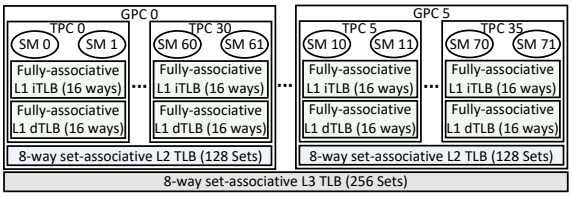


Fig. 2: TLB structure of NVIDIA A10 GPU.

The A10 GPU consists of 6 GPCs, which share an L3 TLB with 256 sets. Each GPC contains 6 TPCs that share an L2 TLB with 128 sets. As usual, each TPC includes two SMs, which share an L1 iTLB and an L1 dTLB. Regarding TLB properties, only the number of sets in the L2 and L3 TLBs varies by GPU model, but other ones (e.g., associativity, replacement policy, and sub-entries) remain consistent across all NVIDIA GPUs, as described in Section II-A.

$$H_{\text{L2}} = \begin{bmatrix} ****1******1******1******1 \\ ****1******1******1******1* \\ ***1******1******1******1** \\ **1******1******1******1*** \\ *1******1******1******1**** \\ 1******1******1******1***** \\ ******1******1******1****** \end{bmatrix} \quad H_{\text{L3}} = \begin{bmatrix} **1*******1*******1*******1 \\ *1*******1*******1*******1* \\ 1*******1*******1*******1** \\ *******1*******1*******1*** \\ ******1*******1*******1**** \\ *****1*******1*******1***** \\ ****1*******1*******1****** \\ ***1*******1*******1******* \end{bmatrix}$$

Fig. 3: A10's L2 and L3 TLB set hash functions for 64KB pages, where '$*$' denotes '0'. Each bit in set index is computed by XORing the results of each matrix row's bitwise multiplication with 27 virtual address bits $[46:20]$.

Both L2 and L3 TLBs use hash functions to map GPU virtual addresses to specific sets. According to [7], these hash functions vary depending on the page size. Using the reverse-engineering method described in [7], we obtained the A10's L2 and L3 TLB set hash functions for 64KB pages, which are presented in Figure 3.

A. Establishment of Priming Capability

As the foundational step, we need to establish our ability to reliably prime any GPU TLB sets of interest. Although the already-derived TLB set hash functions allow us to determine which specific L2/L3 TLB set a GPU virtual address is mapped to, we find that priming a set in a vGPU-powered VM is actually a non-trivial problem.

First, programming directly in CUDA cannot achieve the priming goal. By default, CUDA uses 2MB GPU memory pages, and for this page size, each L2/L3 TLB entry covers a 32MB-aligned GPU virtual address range. Given the linearity of GPU TLB set hash functions, it indicates that to prime an arbitrary L2 TLB set on an A10 GPU, we need the availability of at least a 32GB GPU virtual address range (32MB per entry $\times$ 8 entries per set $\times$ 128 sets). Similarly, as the L3 TLB has 256 sets, priming one of its sets requires the availability of at least a 64GB GPU virtual address range. In CUDA, unless a special feature named unified virtual memory (UVM) is enabled and used, there is no support for demand paging (i.e., each page must have its page frame allocated). Consequently,

having a 32GB/64GB virtual address range requires allocating the same amount of GPU physical memory.

However, the common size of an A10 vGPU instance's physical memory ranges from 4GB to 12GB, far too small to accommodate our priming needs. Actually, even a non-virtualized A10 GPU, with its total 24GB physical memory, cannot meet such requirements. The reverse-engineering approach in [7] primes GPU TLBs with the help of UVM, which supports not only demand paging but also the use of 64KB pages. While the UVM-based solution works in non-virtualized native settings, it is unfortunately not applicable in our case, because *UVM is disabled in vGPU environments* (refer to Appendix C for more details).

We observe that, in contrast to CUDA, graphics rendering APIs like Vulkan use 64KB memory pages by default. With this smaller page size, each L2/L3 TLB entry covers only a 1MB-aligned GPU virtual address range. Accordingly, priming an arbitrary L2 TLB set on an A10 needs only a 1GB virtual address range, and priming an arbitrary L3 TLB set requires just 2GB. These reduced requirements fall well within the physical GPU memory limits of typical vGPU instances. This observation suggests relying on graphics rendering APIs to construct the primitive. However, unlike in CUDA, retrieval of GPU virtual addresses is not well supported in these APIs[1], and more problematically, graphics rendering runtimes employ a Timeout Detection and Recovery (TDR) mechanism that forcibly terminates any shader executing beyond a very short time limit (e.g., 1–2 seconds), making it impractical to continuously perform `Prime+Probe` operations.

The complementary strengths of CUDA and graphics rendering APIs motivate us to try integrating their features. In particular, we notice that Vulkan provides functionality to export its allocated memory objects, while CUDA supports importing external memory objects. Hence, we consider having Vulkan allocate GPU memory structured in 64KB pages and then importing it into CUDA for better programmatic control. Nevertheless, CUDA has been shown to merge multiple 64KB pages into 2MB pages in UVM contexts [7], raising a critical question: When importing Vulkan-allocated GPU memory into CUDA, will the original 64KB pages be preserved or merged into CUDA's default 2MB pages?

To examine whether the 64KB page size is maintained or not, we conduct experiments in both native and virtualized environments. In the native case, we dump GPU memory to extract page tables and confirm that all the imported 64KB pages are not merged. However, this does not guarantee the same outcome for the vGPU setting, where runtime features can differ (e.g., vGPU does not have UVM). Since directly dumping GPU memory is more challenging in vGPU environments, we rely on timing measurements for verification.

To this end, in a vGPU-powered VM, we measure the access times for imported Vulkan-allocated GPU memory of varying sizes. Given an imported memory chunk, we sequentially traverse it with a 1MB stride and then perform 1,000 iterations of this traversal to compute the average access time. The chunk size starts at 256MB and is incremented by 256MB in each experiment until reaching 3GB. For comparison, we also measure the access times for native CUDA-allocated GPU memory. The results are depicted in Figure 4. For the imported memory, we observe that access time jumps noticeably when the chunk size exceeds 1GB, indicating frequent L2 TLB misses, followed by even larger jumps beyond 2GB due to L3 TLB misses. In contrast, access times for CUDA-allocated memory increase almost linearly with chunk size. This behavior confirms that Vulkan-allocated memory maintains its 64KB page size when imported into CUDA in the vGPU setting.

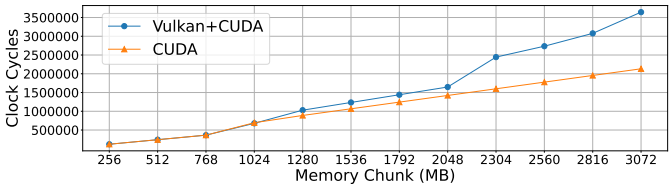


Fig. 4: Access time comparison between imported Vulkan-allocated memory and native CUDA-allocated memory for varying chunk sizes.

We also empirically verified that such Vulkan-CUDA interoperability is available in public cloud vGPU offerings (see Appendix C). Thus, our approach effectively removes the hurdle.

B. Formulation of `Prime+Probe`

With the needed priming capability in place, we now turn to designing an effective `Prime+Probe` attack primitive on GPU TLBs. Our focus is on exploiting the shared last-level L3 TLB as a channel for information leakage. Taking advantage of the GPU's massively parallel thread execution, we can simultaneously monitor the access patterns across all its sets. Figure 5 depicts our design in a nutshell.

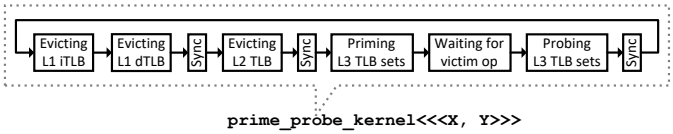


Fig. 5: Design of the attack primitive in a nutshell.

The CUDA kernel responsible for `Prime+Probe` is launched with the configuration `<<<X, Y>>>`, where `X` specifies the number of thread blocks and `Y` specifies the number of threads per block. On NVIDIA GPUs, thread blocks are distributed across SMs in a round-robin fashion, first cycling through even-numbered SMs (0, 2, 4, ...) and then odd-numbered SMs (1, 3, 5, ...), with each thread block assigned sequentially to the next SM. In our design, we require `X` to be at least half of the total SMs, which ensures complete coverage of all TPCs since each TPC houses two consecutively numbered SMs (one even and one odd). This full TPC coverage inherently means that all the GPCs, as the higher-level structure, are also covered.

We also aim to evenly divide the workload of priming and probing L3 TLB sets among the participating SMs, with each

[1]Although Vulkan provides an extension `VK_KHR_buffer_device_address` to query a buffer's device address, we find that it actually returns incorrect GPU virtual addresses.

SM in charge of an equal number of sets. As described later, the launch parameter `Y` is set to 32 times the number of L3 TLB sets assigned to each SM, reflecting the 32 threads in a warp. For instance, on the A10 GPU, which has 72 SMs and 256 L3 TLB sets, we utilize 64 SMs, with each SM handling the priming and probing of exactly four L3 TLB sets; accordingly, the kernel is launched with `<<<64, 128>>>` (i.e., each SM running 4 warps). In the following, we will discuss our primitive's design in detail.

1) Evicting L1 and L2 TLBs: Although the L3 TLB is the target of our `Prime+Probe`, its non-inclusive nature necessitates consideration of both the L1 and L2 TLBs. In each `Prime+Probe` cycle, entries in these upper-level TLBs must be evicted to ensure access to the L3 TLB. Hence, in our primitive, we first fully evict the L1 and L2 TLBs before priming L3.

The L1 dTLB of a TPC and the unified L2 TLB of a GPC can be easily evicted by accessing a sufficient number of data pages. Since the L1 dTLB is always 16-way fully-associative [7], we evict its entries by having each thread in an SM access one of 16 64KB pages, with the specific page indexed by (thread-id % 16). The L2 TLB, on the other hand, has significantly more entries, but the abundance of threads in each SM suggests that each thread only needs to access a few pages to achieve eviction. For example, each L2 TLB in A10 has 1024 entries, and thus each of the 128 threads in an SM just needs to access 8 data pages to fully evict the L2 TLB of the SM's associated GPC, with the pages selected according to the reverse-engineered hash function H_{L2} shown in Figure 3.

To evict the L1 iTLB of each TPC, which is also fully-associative and always has 16 entries [7], an SM shall access at least 16 code pages. CUDA does not provide direct support for creating arbitrary code pages, but we work around this by defining 16 distinct dummy functions, each of which occupies a single page. Since CUDA also uses the 2MB page size for code, each dummy function must be large enough to fill an entire page. As shown in Figure 6, we have the body of each dummy function leverage CUDA's `#pragma unroll` directive for loop unrolling to achieve the 2MB size. Nested loops are used because CUDA's compiler `nvcc` imposes a maximum limit on unrolling a single loop. For the A10 GPU (Ampere architecture), we find that setting `OUTER_NUM` to 2 and `INNER_NUM` to 6000 suffices.

```
1 if (!jump_over) {
2     #pragma unroll OUTER_NUM       // e.g., 2 for Ampere GPUs
3     for (i = 0; i < OUTER_NUM; ++i) {
4         #pragma unroll INNER_NUM   // e.g., 6000 for Ampere GPUs
5         for (j = 0; j < INNER_NUM; ++j)
6             sum += clock64();
7     }
8 }
```

Fig. 6: 2MB page-sized dummy function for L1 iTLB eviction.

Every dummy function takes a `jump_over` parameter that allows us to skip the huge loop body while still accessing the code page. In an SM, each thread only needs to call the dummy function indexed by (thread-id % 16), with `jump_over` set to `true`, to completely evict the L1 iTLB of the SM's associated TPC.

2) Priming and Probing L3 TLB Sets: After evicting entries from the L1 and L2 TLBs, we proceed to prime individual L3 TLB sets. Note that, unlike the eviction of L1 and L2 TLBs where threads in a warp are allowed to access different pages at the same time, we use warps as the operational unit for priming and probing L3 TLB sets, with all 32 threads in a warp working together in lockstep. This warp-based design is essential for the probing phase due to the execution semantics of the GPU's SIMT model, where all threads in a warp take the same amount of time to execute even if some may require less. To maintain a uniform access pattern, warps are adopted as the operational unit for priming as well.

For each L3 TLB set, the reverse-engineered hash function (e.g., H_{L3} for the A10 GPU shown in Figure 3) is applied to construct an eviction set – a sequence of data pages linked via pointer chasing whose translations populate the L3 TLB set. With its eviction set, each L3 TLB set is primed by a dedicated warp in an SM. Specifically, when priming an L3 TLB set, all 32 threads in the assigned warp collectively traverse the corresponding eviction set by following the pointer chain.

After priming the L3 TLB sets, we must wait for our GPU context to be switched out so another VM's GPU context can be scheduled to execute. Only when our context regains control can we perform the probing operation. In [16], two methods for detecting GPU context switches in native environments are introduced, and we find both applicable in vGPU environments as well. For the A10 GPU, the simpler loop-based waiting method is effective. This method leverages the GPU's timestamp counter to monitor the time taken by iterations in a tight loop; when a loop iteration takes significantly longer than previous ones, it indicates that our GPU context was switched away and has now regained control. Note that the large counter difference observed in such cases approximates the duration of the time slice allocated to the other VM.

The probing operation begins once our GPU context is resumed. As mentioned earlier, each warp is responsible for probing one L3 TLB set, that is, all 32 threads in a warp traverse the corresponding eviction set in lockstep to measure access time. However, note that, when an L3 TLB set is primed by a warp in an SM, the translations for the pages in the eviction set are also cached in the L1 dTLB of the SM's TPC and the L2 TLB of its GPC. As a result, warps in the same SM or any other SMs within the same GPC should not be used to probe that L3 TLB set to avoid accessing these upper-level TLBs.

To address this issue, we carefully orchestrate the priming and probing operations. Specifically, for a group of L3 TLB sets, if warps in thread block n are assigned to prime them, we dedicate their probing to the counterpart warps in thread block $(n+1)$ % `X`. This strategy effectively ensures that the priming and probing of a given L3 TLB set are assigned to SMs in different GPCs. As previously mentioned, the thread block-to-SM assignment scheme follows an all-even-then-all-odd pattern (i.e., 0, 2, 4, ..., 1, 3, 5, ...), where the alternating SM numbers are distributed across distinct GPCs. For example, Figure 7 illustrates this arrangement in our A10 case, showing

the relationship between priming and probing of L3 TLB sets, warps, SMs, and GPCs.

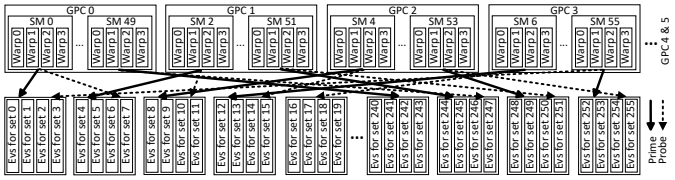


Fig. 7: Orchestration of priming and probing operations on the A10 GPU.

While the current design of probing appears plausible, an interesting problem arises: We observe that when some warps probe victim-accessed L3 TLB sets (resulting in longer access times), other warps probing unaccessed sets also experience similarly long access times, despite no L3 TLB misses in those sets. This unexpected behavior contradicts our expectation that warps operate independently and do not interfere with each other. After conducting experiments, we conjecture that each TLB employs a structure to track outstanding misses, similar to the miss-status handling registers found in caches. When this structure becomes full, new misses in that TLB must stall until space is freed. During probing, every access misses in both the L1 dTLB and the L2 TLB, and if their tracking structures are full, even accesses that will hit in the L3 TLB (such as when probing unaccessed sets) are unable to proceed past the L1/L2 level, leading to prolonged access times.

As a solution to this problem, we serialize warps in each SM for probing. Specifically, we enforce staged execution where only one warp is allowed to probe in each stage. The stages are coordinated using CUDA's `__syncthreads()` function, which acts as a block-level barrier such that no warps can proceed to the next stage until the currently probing warp has completed its measurements.

3) Synchronization: Proper synchronization between threads is critical for the accuracy of our primitive; without it, threads can inadvertently interfere with each other. For example, if warps in one SM complete their probing and start evicting L1 TLBs for the next round while others are still probing, the eviction accesses may disrupt ongoing measurements. In particular, this may introduce translations into L3 TLB sets not accessed by the victim but actively probed by others, leading to false positives. Therefore, we explicitly synchronize potentially interfering operations as shown in Figure 5. Our primitive requires coordination across all thread blocks, and for this, we implement a global synchronization approach where threads perform atomic increment operations on a shared volatile counter and wait until the counter matches the total number of threads before proceeding. (The `__syncthreads()` function does not work, as it only synchronizes threads in the same thread block.)[2]

[2]Since CUDA 9.0, a feature named cooperative groups has been available that can be used to achieve grid-wide synchronization.

C. Enhancement of the Primitive

Our `Prime+Probe` attack primitive in its present form actually faces challenges in practice. First, graphics-intensive workloads in victim VMs tend to access most of the L3 TLB sets during typical vGPU usage, making it hard to derive meaningful information. Second, even with consistent L3 TLB access patterns, we find that significant fluctuations in timing measurements can occur and thus undermine reliability. To enhance the practical utility of the primitive, we need to address these limitations through refinements to our approach.

1) Improving Measurement Resolution: Using the current primitive to extract sensitive information from workloads with heavy graphics rendering is challenging due to possible saturation in measurements. For instance, in cloud gaming scenarios, the victim's GPU context manages several gigabytes of memory in 64KB pages, and within a single vGPU time slice, its graphics rendering operations can generate memory access patterns that engage nearly all L3 TLB sets. Thus, merely determining whether an L3 TLB set was accessed may provide little insight into the victim's behavior. To address this problem, we aim to increase the resolution by going beyond binary accessed/unaccessed states.

Essentially, we revise the probing operation to have each warp traverse the corresponding eviction set in the reverse order. Since GPU TLBs employ a strict LRU replacement policy, traversing the eviction set backwards allows us to directly project measured access times to the number of entries evicted by victim activity. To build the mapping between access times and the number of evicted entries, we run a dummy GPU program alongside our primitive. The dummy program deliberately accesses 0, 1, ..., 8 entries in each L3 TLB set to establish baseline measurements.

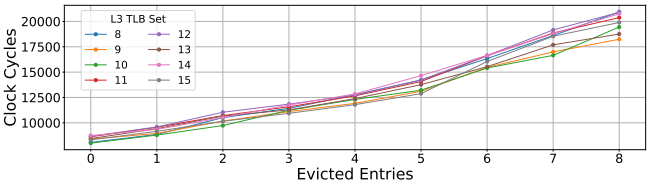


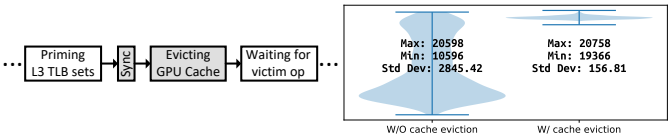
Fig. 8: Timing measurements for eight L3 TLB sets on A10 with different numbers of evicted entries.

Figure 8 illustrates the timing measurements for 8 different L3 TLB sets on the A10 GPU, where we collected 1001 samples per set and used their median values. The results reveal an approximate linear relationship between access time and the number of evicted entries, though some variation is observed across different sets. This linear correlation enables us to quantify L3 TLB contention at a finer granularity than simply detecting whether a set was accessed.

2) Improving Measurement Reliability: The reliability of timing measurements is inevitably affected by GPU cache behavior. First, it is apparent that when accessing memory locations during probing, cache hits take less time than cache misses, introducing timing variations unrelated to TLB behavior. Second, the caching state of page table entries (PTEs) can be another source of timing fluctuation. During page table

walks, PTEs are also stored in the GPU's L2 cache, similar to regular data accesses. When a miss occurs in the L3 TLB, it triggers a page table walk. The walk, however, completes much faster when the required PTEs are already cached in L2, compared to when they need to be fetched from GPU memory.

Such timing variations can lead to incorrect inferences about L3 TLB access patterns. Hence, we should make timing measurements independent of the data in the GPU cache. For this purpose, we evict all data from the L2 cache after completing the priming operation, as shown in Figure 9a, to guarantee a consistently clean cache state regardless of how the victim utilizes the GPU cache.



(a) Revised attack primitive. (b) Probing time variation comparison.

Fig. 9: Synchronized cache eviction for improving probing reliability.

It is important to highlight that when evicting the GPU cache, we must avoid disturbing the primed L3 TLB sets. To this end, each participating SM should use the last pages from its TLB priming eviction sets for cache eviction. As GPU cache sizes are relatively modest (e.g., the A10 has a 6MB L2 cache), effective eviction can be achieved by having all threads in each warp access several different memory blocks of the last pages. For example, in our A10 scenario, with the 128B GPU cache line size and a 6MB cache, each of the 128 threads in an SM theoretically only needs to access 6 distinct memory blocks in the corresponding page. However, due to the GPU cache's non-linear addressing function, we should conservatively access more memory to ensure complete cache eviction, and thus in the A10 case we let each thread access 48 rather than just 6 memory blocks. Moreover, as shown in Figure 9a, we add synchronization after TLB priming to make sure that no threads begin cache eviction while others are still priming.

Figure 9b illustrates the effect of our revised primitive design in A10. Given an L3 TLB set under varying GPU cache pressure but the same TLB access condition (8 entries evicted), we observe that with the original primitive (left side), timing measurements for probing display significant variability, which can lead to incorrect inferences about L3 TLB set access patterns. In contrast, incorporating full cache eviction (right side) yields very stable measurements, enabling reliable determination of L3 TLB contention and emphasizing the necessity of this revision. Note that the times presented in Figure 8 were actually measured with this revision in place.

Nevertheless, when probing all L3 TLB sets concurrently, a potential concern arises: Even with the GPU cache fully evicted beforehand, could earlier accesses during probing preload data required by subsequent accesses into the GPU cache? If this is the case, large timing variations caused by GPU cache behavior will persist, compromising the initial purpose of our cache eviction. For normal data items, this issue does not occur, as any two accessed items are separated by at least a 64KB page frame and therefore do not reside in the same cache line. The question, however, pertains to PTEs, because each 128B cache line contains 16 PTEs. Fortunately, our use of 1MB-aligned pages ensures that although one page table walk loads 16 PTEs into the cache at once, only the PTE required by the access triggering the walk will be used, while the other 15 will remain unused. Thus, once evicted, the GPU cache will no longer affect our timing measurements during probing.

D. Measurement of Execution Time

The attack primitive performs a number of GPU memory accesses. Considering the limited execution time in each vGPU time slice, we need to verify that the primitive can complete all its necessary steps within this time constraint. For the primitive to function correctly, it suffices to confirm that, after regaining control following a context switch, we can successfully finish the probing operation as well as prepare for the next round (i.e., evicting L1/L2 TLBs and priming all L3 TLB sets) before the allocated time slice expires.

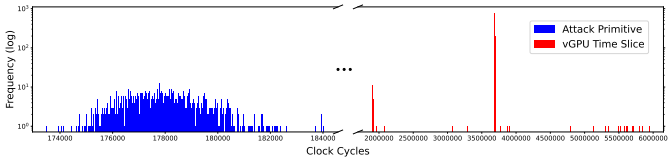


Fig. 10: Execution time of the attack primitive vs. vGPU time slices on A10.

Using vGPU instances created from A10, we conducted empirical measurements. We first measured the execution time of our attack primitive over 1,000 iterations, specifically the duration from the probing operation to the completion of priming for the next round. As illustrated in the left side of Figure 10, the primitive's execution time is quite stable, ranging from 173,483 to 184,044 clock cycles with an average of 177,984 cycles (about 105μs). We then measured 1,000 samples of the time slice allocated under the default best effort vGPU scheduling policy. While the measurements exhibit considerable variation, as shown in the right side of Figure 10, they mostly cluster around 3,670,000 cycles, with the minimum one even exceeding 1,700,000 cycles (approximately 1ms). Therefore, we can confirm that each vGPU time slice is long enough for the attack primitive to complete its operations.

VI. Case Studies

In this section, we present two case studies to demonstrate how our designed `Prime+Probe` primitive can be employed to mount cross-VM side-channel attacks through the TLB of a virtualized GPU. These studies focus on typical vGPU use cases encountered in real-world clouds. Before delving into the specifics, we first describe the cloud platform setup used in our investigations.

A. Platform Setup

As mentioned earlier, NVIDIA A10 GPUs are predominantly used for virtualization in public clouds [2], [4], [5], and we align our setup to use the same GPU model. We have an A10 GPU installed in a Dell PowerEdge R740 server running

Red Hat Enterprise Linux 9.2, with QEMU/KVM deployed as the hypervisor. The server is equipped with two Intel Xeon Silver 4114 CPUs and 192GB memory. On this server, we have installed the NVIDIA vGPU 16.8 package that was released in October 2024, and we use the vWS license. We employ the default best effort vGPU scheduling policy, which is also the case in public clouds (see Appendix B).

The case studies involve VMs on this platform. In line with the profile of Microsoft Azure's NV12ads_A10-v5 series, each VM is attached to an A10-8Q vGPU instance, which provides an 8GB GPU memory partition. Since an A10 has a total of 24GB on-board memory, each vGPU instance is granted a 1/3 portion. In each case study, the attacker utilizes a VM running Ubuntu 20.04, while the victim is assumed to operate a VM with Windows 11, as it is the most common OS used in daily life.

B. Cheating in Multiplayer Games

One of the most popular use cases for vGPU technology is cloud gaming [4], [19]. In the first case study, we focus on this vGPU application area and demonstrate how the GPU TLB side channel can be exploited to assist an attacker in cheating during competitive multiplayer games hosted on cloud platforms.

In many multiplayer games, especially first-person shooter ones, players can gain significant tactical advantages by identifying their opponents' hidden locations and viewpoints. A notable example is Counter-Strike (CS) that has been very commercially successful. As a major esports title, CS is featured in numerous tournaments with prize pools reaching millions of dollars [20].

We specifically use CS as the target game in this case study. The victim is a CS player who uses a vGPU-powered VM cloud gaming service to play. The attacker, while playing CS with the victim, has also rented a VM in the same cloud, sharing the same physical GPU as the victim's VM. Note that this VM is used exclusively to execute the side-channel attack and not for playing the game.

In [21], Genkin *et al.* showed how electromagnetic (EM) radiation from the CPU's voltage regulator, captured by a laptop's internal microphone, can enable a CS player to detect an opponent hidden behind an obstacle. Similarly, we demonstrate that exploiting GPU TLB contention in virtualized cloud gaming environments can also provide such capabilities. Notably, our attack aims at newer versions of CS, whose game engines may neutralize the one described in [21].

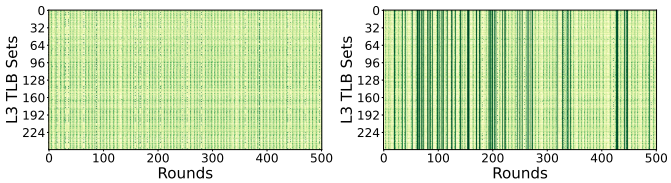
1) Game World Rendering in CS: As a first-person shooter game, CS places multiple players in a 3D virtual environment called a map, where each player controls an avatar that can navigate complex terrains, take cover behind obstacles, and engage in tactical combat with the goal of eliminating opposing players. The immersive 3D settings are fundamental to CS's gameplay, powered by a backbone engine responsible for rendering all visual elements. Outdated CS versions utilize the GoldSrc engine, while modern versions, CS:GO and CS-2, have adopted the more advanced Source and Source 2 engines, respectively.

A CS game begins with players connected to a server (self-hosted or officially provided) that maintains the authoritative game state, including each avatar's continuously updated location and viewing angle. Players retrieve data about objects in their view from the server to render their individual perspectives of the 3D game world. Note that each player's avatar operates within a cone-shaped field of vision, known as the frustum, and the server supplies data about all objects within this frustum, whether they are visible or occluded.

The GoldSrc engine used in early CS versions renders all objects received from the server's data, including those hidden behind walls or other obstacles. This behavior created a vulnerability exploited by many cheating plugins as well as the attack demonstrated in [21]. In contrast, new engines in CS like Source and Source 2 implement occlusion culling techniques to ensure that objects invisible to the player are excluded from the rendering pipeline.

2) Camping Detection: A classic tactic in CS, known as *camping*, involves a player hiding behind an obstacle and waiting to ambush unsuspecting opponents. For camping, players often exploit a map's environmental features, such as walls, vehicles, or crates, to take cover. When an opponent falls into the ambush, the hiding player can strike first, gaining a significant tactical advantage.

We first investigate whether the attacker can determine if the victim's avatar is lying in wait. When an avatar remains stationary, the game engine receives similar scene data from the server across multiple frames. For performance optimization, the engine will skip rendering many elements. However, when the avatar is moving, the scene changes dynamically, which requires the engine to render new content and thus heavily utilize the GPU. Based on this difference in rendering behavior, we hypothesize that an attacker can detect whether the victim's avatar is camping by simply monitoring the GPU's L3 TLB access patterns.



(a) Avatar stays still. (b) Avatar moves around.

Fig. 11: L3 TLB access patterns when a player is camping vs. moving (under the latest CS-2).

Results. We have experimentally verified that GPU L3 TLB access patterns can reliably indicate whether the victim's avatar is camping or moving. Figure 11 shows the access patterns observed over 500 `Prime+Probe` rounds (approximately 1.5 seconds) in both scenarios. Comparing Figures 11a and 11b reveals that when the avatar is in motion, L3 TLB sets experience higher levels of contention across multiple `Prime+Probe` iterations (with darker colors representing a higher number of accesses).

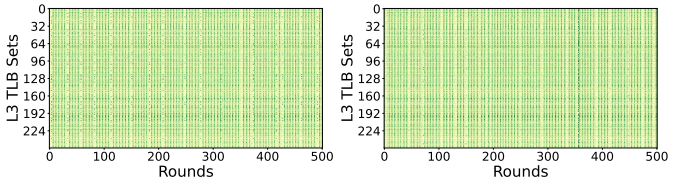
Note that the visualization underscores the importance of our enhanced measurement resolution, which quantifies the degree

of contention in each L3 TLB set rather than merely detecting whether a set was accessed (see Section V-C). Without this enhancement, the observed patterns would appear uniform across multiple rounds, as most L3 TLB sets are accessed consistently.

3) Camper Revelation: On a specific CS map, the camping tactic is effective in certain locations that offer both good concealment and strategic positioning. Note that these spots are typically well-known, which makes a player's view frustum predictable when they are suspected of camping there. Given that the attacker can straightforwardly utilize GPU L3 TLB access patterns to detect if the victim is camping, we now examine whether the attacker can determine if the victim is hiding at a specific spot.

In [21], Genkin *et al.* illustrate that an attacker can move their avatar in and out of a suspected camper's presumed view frustum while staying unseen by the camper (e.g., by maneuvering near the obstacle between them). This movement causes variations in graphics rendering on the camper's side, as the game engine processes objects irrespective of their visibility from the player's perspective. The rendering variations also create distinct EM signals, which are captured by the camper's laptop microphone and transmitted to the attacker via VoIP. The attacker can then analyze these audio signals to determine whether someone is hiding behind the obstacle.

In our case, we can also have the attacker move in and out of the suspected view frustum to induce distinct GPU TLB access patterns through rendering variations on the victim's side. However, we discover that the work by Genkin *et al.* likely targeted earlier CS versions that used the GoldSrc engine, even though this detail was not explicitly stated in [21]. Since 2012, CS has transitioned to more advanced game engines. As aforementioned, these newer engines incorporate occlusion culling techniques, which exclude objects not visible to the player from the rendering pipeline. Accordingly, their original attack strategy may not be directly applicable to modern CS versions.

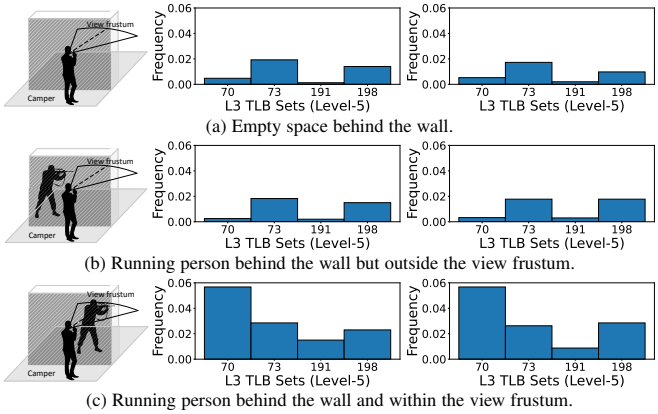


(a) Empty space behind the wall. (b) Running person behind the wall.

Fig. 12: L3 TLB access patterns when a player's view frustum has an empty space vs. an avatar running behind the wall (under the latest CS-2).

To verify this, we conducted an experiment using the latest CS-2, where one player uses a wall as cover for ambushing. We measured the GPU's L3 TLB access patterns in two scenarios: (1) when no other player is present and (2) when another player moves behind the wall and within the ambushing player's view frustum. The results are depicted in Figure 12a for the first scenario and in Figure 12b for the second scenario. We can observe that the patterns appear remarkably similar in both cases, and in fact, they closely align with the stationary access patterns shown in Figure 11a. This similarity can be attributed to the occlusion culling techniques implemented in the Source 2 engine, and it suggests that determining if a camper is behind an obstacle based **directly** on the TLB access patterns is not feasible. (It also indicates that the attack in [21] may be infeasible in up-to-date CS versions.)

Although we cannot detect a camper hiding at a spot directly from the GPU TLB access patterns, we discover that analyzing the statistical distribution of L3 TLB set contention can achieve this goal. Specifically, we construct histograms from `Prime+Probe` measurements collected over a short time window (e.g., 4–5 seconds). For each L3 TLB set, we create a histogram with 9 bins (0–8), where each bin represents a different level of contention observed in that set. Over multiple `Prime+Probe` rounds, we update each set's histogram by incrementing the bin corresponding to the measured contention level. For each L3 TLB set, we calculate the frequency of each contention level by dividing the bin's count by the total number of `Prime+Probe` rounds.



(a) Empty space behind the wall.

(b) Running person behind the wall but outside the view frustum.

(c) Running person behind the wall and within the view frustum.

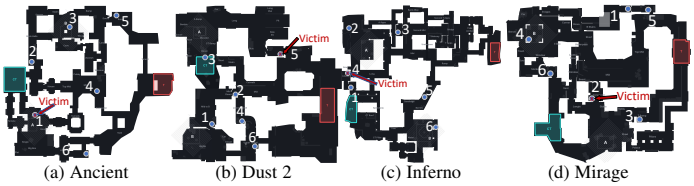
Fig. 13: Frequency of level-5 contention across three L3 TLB sets in two game rounds of CS-2, showing three scenarios: (a) completely empty behind the wall, (b) an avatar running behind the wall but outside the view frustum, and (c) an avatar running behind the wall and within the view frustum.

Our experiments reveal that for certain TLB sets, the frequency of specific contention levels can indicate whether another avatar is moving within the view frustum, even though it is occluded by an obstacle. Figure 13 presents the results for three scenarios across two game rounds of CS-2 on the Mirage map, with a camper hidden behind a wall: (a) no other player is near the camper; (b) another player moves around the wall but remains outside the camper's view frustum; (c) the player moves into the camper's view frustum while staying hidden behind the other side of the wall. The figure highlights four representative L3 TLB sets at contention level-5 (that are sets 70, 73, 191, and 198), showing the frequency distribution of measurements taken over a 5-second span. By comparison, we observe that when a player moves within the camper's view frustum while remaining occluded, these sets show distinctive patterns of level-5 contention (higher frequency in the sets shown in the figure, though potentially lower in others).

Our conjecture for this phenomenon is that, although occlusion culling prevents the hidden avatar from being fully rendered, the game engine still performs minimal GPU operations

for occlusion testing. For example, to determine if the avatar should be culled, its simplified bounding box (or occluder geometry) will be rendered in an invisible pass. While these computations are very lightweight, they still introduce small variations in GPU TLB access patterns. Given the minimal impact of these computations, we need to accumulate multiple such events over time to perform statistical analysis and identify meaningful differences.

Evaluation. To evaluate whether the frequency variation patterns can reliably detect hidden campers, we conduct a systematic evaluation across various maps and obstacles. We choose four classic CS maps, that are Ancient, Dust 2, Inferno, and Mirage, as our testing grounds. Figure 14 outlines these maps. On each map, we have identified six potential camping spots that provide tactical advantages, and the victim player randomly selects one spot to camp.



(a) Ancient (b) Dust 2 (c) Inferno (d) Mirage

Fig. 14: CS-2 maps showing six potential camping spots in each, with the actual victim's camping spot indicated.

Given a spot on a map, the attacker player takes two measurements. The first measurement is taken as the attacker approaches the general area of the spot, and the second measurement is taken when the attacker moves into the presumed view frustum while remaining hidden behind the spot's obstacle. Each measurement lasts 5 seconds. The frequency vectors corresponding to a certain contention level for a set of L3 TLB sets are derived from the measurements. The Euclidean distance between these frequency vectors is then computed to determine whether the victim player is camping behind the spot's obstacle.

Results. First, we find that different CS maps may require different sets of L3 TLB sets for effective detection. However, once identified for a particular map, the applicability of the L3 TLB sets remains consistent across game rounds and power cycles. Furthermore, we have validated that these selected TLB sets remain effective when testing on *another Windows 11 VM*, demonstrating that the sets are map-specific rather than system-dependent. Table I lists several reliable L3 TLB sets for each map (10 for each, except for Mirage), along with their corresponding contention levels to focus on.

TABLE I: Identified L3 TLB sets and contention level for each CS map.

	L3 TLB Sets (Contention Level)
Ancient	43 (4), 78 (5), 96 (5), 125 (4), 129 (4), 130 (4), 131 (4), 145 (4), 150 (4), 162 (4)
Dust 2	43 (4), 70 (4), 130 (4), 154 (4), 160 (4), 172 (4), 176 (4), 230 (4), 202 (5), 123 (6)
Inferno	36 (4), 76 (4), 131 (4), 149 (4), 172 (4), 186 (4), 126 (5), 149 (5), 203 (5), 221 (5)
Mirage	220 (4), 49 (5), 70 (5), 73 (5), 191 (5), 198 (5)

We observe that the contention level for most reliable L3 TLB sets centers around 4 and 5. In the case of a CS map, the determined set of L3 TLB sets is consistently used, and their frequency vectors are derived based on the measurements. Table II gives the evaluation results, showing how the Euclidean distance between the frequency vectors of two measurements can be used to detect whether a camper is ambushing at a particular location.

TABLE II: Evaluation results for identifying the camping spot of the victim.

	Spot 1	Spot 2	Spot 3	Spot 4	Spot 5	Spot 6	Threshold
Ancient	0.14079	0.00857	0.00986	0.01286	0.01044	0.01574	≥ 0.10
Dust 2	0.01191	0.02396	0.02231	0.02005	0.12584	0.01738	≥ 0.10
Inferno	0.01454	0.02104	0.01671	0.15692	0.02433	0.00887	≥ 0.10
Mirage	0.00513	0.06224	0.00798	0.00582	0.00843	0.00674	≥ 0.03

For each map, a threshold is determined through profiling. As shown in the results, the distance at the correct spot always exceeds the threshold when the attacker performs the detection. In contrast, at incorrect spots, the distance remains significantly smaller, clearly distinguishing the correct location. Hence, the attacker can exploit the presented GPU TLB side channel in the new version of CS-2 to gain a significant advantage by identifying whether the victim is camping and pinpointing their exact location.

C. Website Fingerprinting

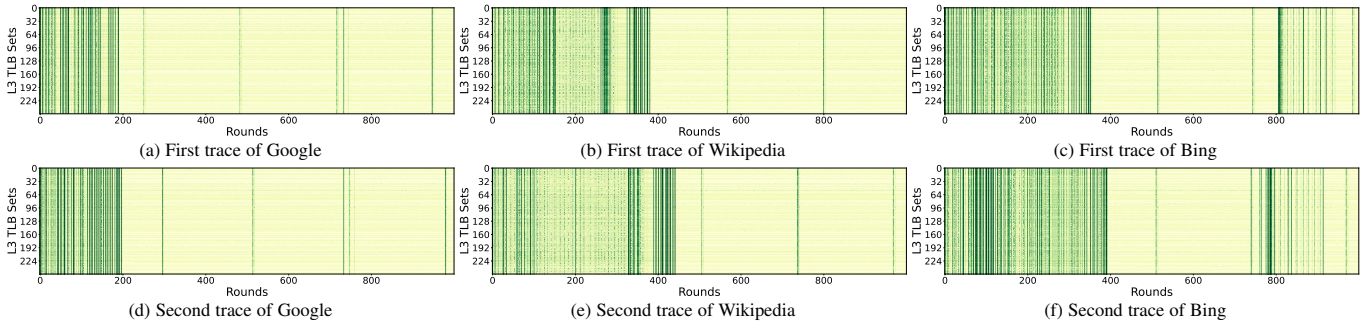
Another very common adoption of vGPU technology in clouds is DaaS [6], [22], [23]. As the second case study, we demonstrate a website fingerprinting attack that leverages GPU TLB access patterns to identify which web pages a user is visiting in their browser on a vGPU-powered virtual desktop.

It is known that a user's browsing activity can reveal highly sensitive information, such as political views, financial statuses, and medical conditions, making it a critical privacy concern. While prior works have explored website fingerprinting via GPUs [15], [16], [24]–[26], these approaches generally necessitate the deployment of malware on the victim's machine. By contrast, our work represents a new contribution by showing how such attacks can be mounted in cloud environments without requiring direct system access.

1) Web Page Rendering in Browsers: Web page rendering is the process by which a browser interprets HTML, along with associated CSS and JavaScript, and displays the resulting content on the screen. This process involves multiple steps, including DOM tree construction, style calculation, layout, rasterization, and composition. Among these steps, rasterization, which converts web page elements into pixels, and composition, which combines rendered textures into a final screen image, are particularly computationally intensive due to the large amounts of data they need to process.

As GPUs excel at processing massive amounts of data in parallel, modern browsers like Chrome, Firefox, and Edge leverage them by default to handle computation-heavy steps in web page rendering. Many rendering steps, particularly rasterization and composition, are offloaded to the GPU, freeing the CPU for other operations and improving overall performance.

2) Web Page Inference: Given the GPU's involvement in web page rendering, different websites naturally generate varying GPU workloads due to their distinct designs, contents, layout complexities, and visual effects. We expect that these workload variations create distinct patterns in GPU memory accesses, enabling website fingerprinting through monitoring address translation activities in GPU TLBs. To validate our



(a) First trace of Google (b) First trace of Wikipedia (c) First trace of Bing

(d) Second trace of Google (e) Second trace of Wikipedia (f) Second trace of Bing

Fig. 15: Traces of the A10's L3 TLB access patterns when visiting three websites in Edge browser.

hypothesis, we use the Edge browser within a Windows 11 VM to visit three popular websites and collect traces of the A10's L3 TLB access patterns in another VM. (One trace consists of 1,000 `Prime+Probe` measurements.) For each website, we capture two sets of traces, as illustrated in Figure 15.

From these traces, we can observe distinctive patterns associated with each website, despite some functional or aesthetic similarities they may share (e.g., Google and Bing being search engines, and Google and Wikipedia having minimalist layouts). While traces from the same website exhibit general consistency in their patterns, they still show some variations, particularly in the timing and duration of intensive GPU computations.

Evaluation. In our evaluation, we, as the attacker, capture traces of L3 TLB access patterns for 100 websites in a profiling VM to create a dataset. All traces were collected using the Edge browser, as it is the default browser on Windows systems and commonly used among typical virtual desktop users. The websites are selected from the Tranco list [27], and their details are provided in Appendix D.

For each website, we collect 100 traces on the profiling VM, resulting in a dataset of 5,000 traces in total. For the task of classifying traces into their respective websites, we utilize the standard ResNet-50 model. To evaluate the performance of our website fingerprinting attack, we use traces captured from the victim's VM, with 20 traces per website.

Results. First of all, we perform a straightforward 5-fold cross-validation on the profiling traces to verify that our method is practical. The procedure yields an average accuracy of 92.3%, with the five folds scoring 92.1%, 92.7%, 92.3%, 92.2%, and 92.1%, respectively. Our main objective, however, is to assess how well a model trained on the profiling VM traces performs when tested on traces collected from the victim's VM.

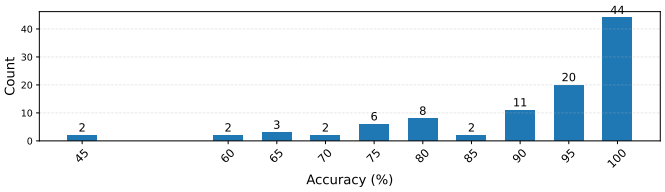


Fig. 16: Distribution of per-website classification accuracy.

The aggregate results are presented in Figure 16, while the full confusion matrix appears in Figure 19 of Appendix D. The results demonstrate good classification performance, with most websites being correctly identified with very high accuracy. Specifically, 44 out of 100 websites achieve 100% accuracy (i.e., 20 out of 20 test samples correctly classified), and another 31 websites achieve at least 90% accuracy (i.e., $\geq$ 18 out of 20 test samples correctly classified). The lowest accuracy for any individual website is 45% for two websites. Overall, the model achieves an average accuracy of 91% across all 100 websites.

The few misclassifications that do occur tend to happen between websites with similar content delivery patterns. For example, some confusion exists between different sites from the same company (like outlook.com and live.com) and between social media platforms. This suggests that websites with similar types of content and rendering workflows can produce comparable GPU TLB access patterns. Nevertheless, the high accuracy across diverse website categories, including search engines, social media, news sites, and e-commerce platforms, demonstrates the robustness of our fingerprinting attack.

D. Discussion

While TLB-based side channels have inherent limitations compared to cache-based approaches, particularly since TLB-based attacks operate at a page-level granularity, this limitation is less significant in virtualized GPU environments. The fundamental constraint in vGPU settings stems from the time-sharing execution model, where GPU contexts are scheduled in millisecond-range time slices. This coarse temporal granularity already restricts the achievable spatial resolution of any microarchitectural side channel, whether cache- or TLB-based. Our work demonstrates that despite these architectural and scheduling constraints, carefully engineered TLB-based attacks can still effectively extract meaningful sensitive information across VM boundaries.

Like all the microarchitectural side channels, our attacks can suffer from noise introduced by other irrelevant co-resident VMs. In practice, however, the economics of cloud GPU leasing greatly reduce this interference. vGPU-powered VM instances are more expensive than CPU-only VMs, so providers colocate far fewer tenants on each physical GPU than on a CPU socket, naturally lowering background activity. In addition, tenants often release their GPU instances as soon as their workloads finish to avoid the high hourly charges, creating long intervals during which an attacker's VM shares the device only

with the victim. As a result, we argue that the GPU TLB side channel we have presented is likely to be much quieter (and therefore easier to exploit) in real clouds than the CPU-based side channels studied in prior work. In Appendix E, we include several trace samples captured on real-world cloud platforms to back up our argument.

VII. Countermeasures

The root cause of the cross-VM side-channel attacks discussed in this paper is the shared GPU TLB in cloud environments. To address this vulnerability, NVIDIA should enhance the security of the TLB hierarchy in its GPUs. For this purpose, NVIDIA could implement either a static-partition (SP) TLB or a random-fill (RF) TLB design, as proposed in [28]. The SP TLB can effectively prevent cross-VM interference, while the RF TLB aims to break the correlation between memory accesses and TLB state changes, making attack patterns unpredictable. Both TLB designs offer strong security guarantees, but they require hardware modifications and thus cannot be applied to existing GPUs.

Without any GPU hardware modification, the most direct method for completely eliminating these attacks would be for the vGPU runtime to initiate a TLB shootdown at the end of each time slice. (Since the vGPU software package is not open-sourced, NVIDIA needs to add this functionality.) However, such a TLB invalidation method can incur significant performance overhead, as every GPU context would need to rebuild its TLB states through expensive page table walks in each vGPU time slice. As a result, this approach may not be practical for deployment in real-world production environments.

At the VM level, tenants can attempt to mitigate the attacks by introducing obfuscation through noise injection in GPU TLB access patterns. Specifically, when performing sensitive GPU workloads, a thread may randomly access certain GPU virtual addresses whose translations are mapped to different L3 TLB sets to obfuscate the original patterns. To provide sufficient coverage of the L3 TLB sets, the GPU memory used here should be allocated in 64KB pages (e.g., using the technique described in Section V-A if in CUDA).

Another potential countermeasure is based on detection. Under normal conditions, frequent L3 TLB misses for a set of sufficiently separated GPU virtual addresses (e.g., 1MB apart in Vulkan) are extremely rare. A tenant can periodically monitor the access times of these addresses, and if unusually frequent L3 TLB misses are observed, it may indicate an ongoing attack like the one described in this paper. Upon detecting such suspicious activity, the tenant may consider pausing operations or migrating their VM to mitigate the threat.

VIII. Related Work

Over the last decade, TLBs have been commonly exploited for side-channel attacks on the CPU side. A key prerequisite for these attacks is the reverse-engineering of this microarchitectural component [29]–[31]. While most reverse-engineering methods rely on timing measurements, Tatar *et al.* introduced a highly accurate approach that leverages TLB incoherence to gain detailed insights [30].

Multiple works have demonstrated how CPU TLBs can be exploited to bypass KASLR [31]–[34]. Moreover, Gras *et al.* showed that CPU TLBs can even be abused to leak fine-grained information such as cryptographic keys [29]. In addition to TLBs in the MMU, CPUs also feature IOTLBs in the IOMMU. The structures of these IOTLBs have also been reverse-engineered and leveraged in various side-channel attacks [35], [36].

During page table walks, PTEs are brought into the data cache and can be exploited for mounting side-channel attacks directly [37] or facilitating circumvention of software-based cache partitions [38]. In our `Prime+Probe` attack primitive, cached PTEs can interfere with timing measurements, and therefore we evict the GPU cache to mitigate this effect.

In recent years, GPU security has drawn growing research attention. GPU TLBs have been reverse-engineered, but their exploitation has primarily focused on constructing covert channels for data exfiltration [7], [39]. Similar to the work in [30], Zhang *et al.* leveraged TLB incoherence to fully reverse-engineer NVIDIA GPU TLBs [7]. Building on their findings, our work advances this line of research by exploring potential side-channel attacks in cloud settings.

Note that while `Prime+Probe` appears in [7], their method is quite straightforward and not applicable in vGPU scenarios: *First*, they directly leverage UVM to acquire 64KB pages in CUDA, but UVM is disabled under vGPU. We overcome this limitation in an innovative way by importing Vulkan-allocated memory into CUDA. *Second*, their environment allows GPU contexts from different VMs to execute in parallel, whereas vGPU enforces temporal partitioning that serializes those contexts. We therefore develop sophisticated techniques for context switch detection, synchronization, and priming/probing orchestration under vGPU's time-sharing model. *Third*, their implementation only detects whether a TLB set is accessed or not, while ours offers finer-grained TLB contention quantification, essential for our case studies. Furthermore, we examine how to significantly improve the reliability of `Prime+Probe` on GPU TLBs through systematic cache eviction, which is a topic not studied in [7]. *Finally*, they only demonstrate inference of six ML frameworks, while our work explores cross-VM side-channel attacks in cloud gaming and virtual desktop scenarios, providing a more extensive study.

Aside from TLBs, other GPU components have also been exploited to leak sensitive information. These components include data caches [16], [40], [41], codec engines [15], NoC interconnects [14], [42], and the PCIe bus [15], [26]. However, most of these works focus on covert channel communication in non-virtualized environments, rather than cross-VM side-channel attacks in cloud settings.

Regarding the case studies performed in our work, Genkin *et al.* showed how physical side-channel information can be exploited by a CS player to detect a hidden camper at a spot [21]. However, as discussed in the paper, their method may not be effective in newer versions of CS, such as CS-2, which

we evaluated. Several studies have investigated website fingerprinting attacks on GPUs [16], [24]–[26], [43]–[45]. Unlike these approaches, which generally require deploying malware on the victim's machine, our work is the first to be conducted on VMs used in practice, with no need for access beyond sharing the virtualized GPU (see Table III in Appendix D for a side-by-side comparison).

IX. Conclusion

In this work, we have developed a `Prime+Probe` attack primitive, enabling cross-VM side-channel attacks through TLBs in virtualized GPUs. The effectiveness of this attack primitive is demonstrated through two case studies that exploit common vGPU use cases, revealing previously overlooked implications. Our findings highlight the security risks introduced by GPU sharing in virtualized environments, urging the need for stronger isolation mechanisms and proactive countermeasures to safeguard against such threats.

Acknowledgment

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

References

[1] NVIDIA, "Virtual GPU Software User Guide," 0. [Online]. Available: https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/

[2] Microsoft, "NVadsA10 v5-series," 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/nva10v5-series

[3] Vultr, "Cloud GPU." [Online]. Available: https://www.vultr.com/products/cloud-gpu/

[4] Alibaba, "vGPU-accelerated instance families (vgn and sgn series)." [Online]. Available: https://www.alibabacloud.com/help/en/egs/vgpu-accelerated-instance-families

[5] Tencent, "GPU Cloud Computing Instance Types: Rendering Instance." [Online]. Available: https://www.tencentcloud.com/document/product/560/19700

[6] Cloudalize, "Why GPU Acceleration is Essential for Virtual Desktop Performance." [Online]. Available: https://www.cloudalize.com/blog/why-gpu-acceleration-is-essential-for-virtual-desktop-performance/

[7] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, 2023, pp. 960–974.

[8] K. Group, "OpenGL - The Industry Standard for High Performance Graphics," 0. [Online]. Available: https://www.opengl.org/

[9] Microsoft, "DirectX programming," 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows/uwp/gaming/directx-programming

[10] K. Group, " Vulkan - Cross platform 3D Graphics," 0. [Online]. Available: https://www.vulkan.org/

[11] NVIDIA, "CUDA C++ Programming Guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] K. Group, "OpenCL - Open Standard for Parallel Programming of Heterogeneous Systems," 0. [Online]. Available: https://www.khronos.org/opencl/

[13] NVIDIA, "NVIDIA Multi-Instance GPU User Guide," 0. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

[14] Z. Jin, C. Rocca, J. Kim, H. Kasan, M. Rhu, A. Bakhoda, T. M. Aamodt, and J. Kim, "Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture," in *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*, 2024, pp. 885–898.

[15] Y. Miao, Y. Zhang, D. Wu, D. Zhang, G. Tan, R. Zhang, and M. T. Kandemir, "Veiled Pathways: Investigating Covert and Side Channels within GPU Uncore," in *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO '24)*, 2024, pp. 1169–1183.

[16] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, "Invalidate+Compare: A Timer-Free GPU Cache Attack Primitive," in *33rd USENIX Security Symposium (USENIX Security 24)*, Aug. 2024, pp. 2101–2118.

[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006, pp. 1–20.

[18] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*, 2015, pp. 605–622.

[19] NVIDIA, "NVIDIA RTX Server Powering the Future of Cloud Gaming and AR/VR," 0. [Online]. Available: https://www.nvidia.com/en-au/data-center/cloud-gaming-server/

[20] C. Gough, "Annual cumulative Counter-Strike: Global Offensive (CS:GO) and Counter-Strike 2 (CS-2) tournament prize pool worldwide from 2015 to 2024," 0. [Online]. Available: https://www.statista.com/statistics/807908/csgo-tournament-prize-pool/

[21] D. Genkin, N. Nissan, R. Schuster, and E. Tromer, "Lend Me Your Ear: Passive Remote Physical Side Channels on PCs," in *31st USENIX Security Symposium (USENIX Security 22)*, Aug. 2022, pp. 4437–4454.

[22] Apps4Rent, "Hosted Virtual Desktop with vGPU." [Online]. Available: https://www.apps4rent.com/gpu-virtual-desktop/

[23] R. Spruijt, "Maximizing Performance with Frame: A Comprehensive GPU Solution Guide." [Online]. Available: https://docs.dizzion.com/solution-guides/2024/02/16/maximizing-performance-with-frame-comprehensive-gpu-solution-guide

[24] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P '14)*, 2014, pp. 19–33.

[25] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks Are Practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018, pp. 2139–2153.

[26] M. Side, F. Yao, and Z. Zhang, "LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit," in *Proceedings of the 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P '22)*, 2022, pp. 270–285.

[27] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," *arXiv preprint arXiv:1806.01156*, 2018.

[28] S. Deng, W. Xiong, and J. Szefer, "Secure TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*, 2019, pp. 346–359.

[29] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, Aug. 2018, pp. 955–972.

[30] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering," in *31st USENIX Security Symposium (USENIX Security 22)*, Aug. 2022.

[31] H. Jang, T. Kim, and Y. Shin, "SysBumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, 2024, pp. 64–78.

[32] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P '13)*, 2013, pp. 191–205.

[33] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016, pp. 368–379.

[34] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P '20)*, 2020, pp. 309–321.

[35] T. Tiemann, Z. Weissman, T. Eisenbarth, and B. Sunar, "IOTLB-SC: An Accelerator-Independent Leakage Source in Modern Cloud Systems,"

in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIA CCS '23)*, 2023, pp. 827–840.
[36] T. Kim, H. Park, S. Lee, S. Shin, J. Hur, and Y. Shin, "DevIOus: Device-Driven Side-Channel Attacks on the IOMMU," in *Proceedings of the 2023 IEEE Symposium on Security and Privacy (S&P '23)*, 2023, pp. 2288–2305.
[37] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, vol. 17, 2017, p. 26.
[38] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *27th USENIX Security Symposium (USENIX Security 18)*, Aug. 2018, pp. 937–954.
[39] A. Nayak, P. B., V. Ganapathy, and A. Basu, "(Mis)Managed: A Novel TLB-Based Covert Channel on GPUs," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*, 2021, pp. 872–885.
[40] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and Characterizing Covert Channels on GPGPUs," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, 2017, pp. 354–366.
[41] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.
[42] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim, "Network-on-Chip Microarchitecture-Based Covert Channel in GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021, pp. 565–577.
[43] S. Wu, J. Yu, M. Yang, and Y. Cao, "Rendering Contention Channel Made Practical in Web Browsers," in *31st USENIX Security Symposium (USENIX Security 22)*, Aug. 2022, pp. 3183–3199.
[44] M. Tan, J. Wan, Z. Zhou, and Z. Li, "Invisible Probe: Timing Attacks with PCIe Congestion Side-channel," in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (S&P '21)*, 2021, pp. 322–338.
[45] Z. Zhan, Z. Zhang, S. Liang, F. Yao, and X. Koutsoukos, "Graphics Peeping Unit: Exploiting EM Side-Channel Information of GPUs to Eavesdrop on Your Neighbors," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P '22)*, 2022, pp. 1440–1457.
[46] "Deploying Siemens NX/X on Azure Virtual Desktop." [Online]. Available: https://techcommunity.microsoft.com/blog/azurehighperformancecomputingblog/deploying-siemens-nxx-on-azure-virtual-desktop-multi-session-gpu-sharing-for-cad/4423817
[47] "NVIDIA and Microsoft are helping TBI streamline construction processes and create the best collaborative workspaces possible." [Online]. Available: https://www.nvidia.com/en-us/customer-stories/rtx-virtual-workstations-and-azure/

Appendix A
vGPU Market Growth

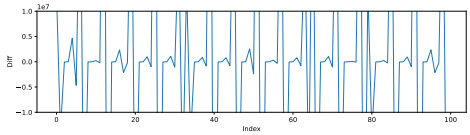
The adoption of vGPU technology has progressed from experimental deployments to widespread commercial availability. Because CSPs typically do not publish detailed usage metrics for specific VM instance types, the pace and breadth of regional expansion for a given instance type provide a reliable proxy for market demand.

Microsoft Azure's NVadsA10-v5 series, which offers A10 vGPU instances, exemplifies this trajectory. The series debuted in 2022 in only five regions (three in the US and two in Europe). As of 2025, Azure lists the NVadsA10-v5 series as available in 48 regions worldwide. Because a CSP expands capacity only when sustained customer demand justifies the investment, this roughly ten-fold geographic expansion in under three years indicates that vGPU-powered VM instances have gained broad adoption. This rapid growth, together with published customer success stories (e.g., Siemens NX [46] and TBI Construction [47]), shows that vGPU use becomes common in practice.

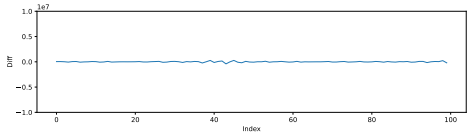
Appendix B
vGPU Scheduling

We can use the timing information from our `Prime+Probe` primitive's context switch detection mechanism to check which scheduling policy is configured in the hypervisor. As previously discussed, when a context switch occurs, the difference in successive timestamp counter readings approximates the duration of the time slice allocated to the other VM's GPU context.

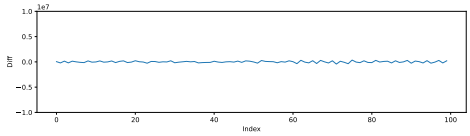
We measure a sequence of vGPU time-slice durations and compute the difference between successive slices to quantify their variability. Figure 17 presents the results obtained on our platform under the three scheduling policies. As the figures indicate, both the equal share and fixed share policies deliver highly stable slices with negligible variation, whereas the best effort policy produces time slices that fluctuate noticeably. Therefore, we can determine the vGPU scheduling policy by analyzing the degree of time slice variability.



(a) Best effort.



(b) Equal share.



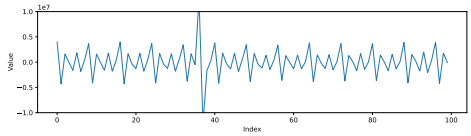
(c) Fixed share.

Fig. 17: Comparison of time slice variability across different vGPU scheduling policies.

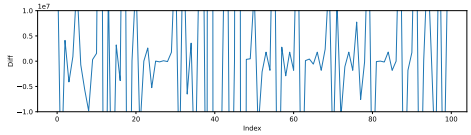
We have applied this method to vGPU-powered VMs from two public clouds, Microsoft Azure and Vultr. As illustrated in Figure 18, we observe substantial fluctuations in both traces. While the magnitude of variation in time slice durations depends on the computational load of co-located VMs, the presence of significant fluctuation strongly indicates the use of the best effort scheduling policy. Based on these measurements, we can confidently determine that both cloud providers employ the default best effort scheduling approach rather than equal share or fixed share alternatives.

Appendix C
Extended Discussion of UVM and Vulkan-CUDA Interoperability in vGPU Environments

UVM requires special system-level support beyond the base CUDA functionality (e.g., additional integration for page fault handling and memory page migration). As mentioned in Section V-A, UVM is disabled under vGPU. NVIDIA's documentation notes that hypervisors can, in principle, enable UVM via a configuration flag, although the flag is off by default [1]. However, in practice, *the option proved ineffective on our*



(a) Best effort on Azure.



(b) Best effort on Vultr.

Fig. 18: Time slice variability on two public clouds.

testbed: setting the parameter had no observable effect at all and UVM remained unavailable.

To rule out vendor-specific builds in public clouds, we also attempted to use UVM in vGPU-powered VMs on Microsoft Azure, Vultr, and Alibaba Cloud. On both cloud platforms, the NVIDIA driver aborted during UVM initialization and reported identical fault codes. The repeated failure across on-premise and cloud setups shows that UVM is not supported in real-world vGPU deployments.

On the other hand, Vulkan-CUDA interoperability is a native capability of the NVIDIA software stack. This interoperability is universally available in public cloud vGPU environments, which we have empirically verified on major providers including Microsoft Azure, Vultr, and Alibaba Cloud. Such universal availability exists because Vulkan runtime support is built into the NVIDIA driver, and CUDA's external-memory import API is a standard CUDA feature rather than something that requires additional system-level support. Hence, our technique works consistently across different vGPU cloud offerings.

While we demonstrate our attack using Vulkan-CUDA interoperability, the core requirement is simply a graphics rendering API that allocates 64KB device pages importable into CUDA. Other APIs such as OpenGL can serve as potential alternatives. Note that OpenGL-CUDA interoperability is also universally available in vGPU environments and can be similarly exploited for priming and probing GPU TLBs.

Appendix D
Supplementary Material for Website Fingerprinting

Table IV lists the websites that are used in our evaluations on website fingerprinting. Table III compares our work with representative website fingerprinting attacks on GPUs. Figure 19 presents the confusion matrix of our website fingerprinting evaluation.

TABLE III: Comparison of website fingerprinting attacks on GPUs

	Leakage Source	Virtualized GPU	Cross-VM	#Websites	Accuracy
[25]	GPU memory allocation	✗	✗	200	90.0%
[24]	GPU memory residue	✗	✗	100	95.4%
[43]	Rendering contention	✗	✗	100	>70%
[44]	PCIe contention	✗	✗	100	96.3%
[26]	PCIe contention	✗	✗	100	95.2%
[16]	GPU L2 cache	✗	✗	50	>98%
[45]	DVFS-induced EM	✗	✗	50	85.3%
[15]	NVDEC utilization	✗	✗	40	89.2%
Ours	GPU TLB	✓	✓	100	91%

TABLE IV: List of fingerprinted websites

1. adobe.com
2. amazon.com
3. android.com
4. apache.org
5. apple.com
6. archive.org
7. baidu.com
8. bankofamerica.com
9. bbc.com
10. bestbuy.com
11. bing.com
12. bit.ly
13. booking.com
14. canva.com
15. cdc.gov
16. chase.com
17. cloudflare.com
18. cnn.com
19. criteo.com
20. dailymail.co.uk
21. digicert.com
22. discord.com
23. doi.org
24. dropbox.com
25. duckduckgo.com
26. ebay.com
27. epicgames.com
28. espn.com
29. europa.eu
30. facebook.com
31. fastly.net
32. forbes.com
33. foxnews.com
34. github.com
35. github.io
36. gmail.com
37. godaddy.com
38. google.com
39. googledomains.com
40. gravatar.com
41. harvard.edu
42. health.mil
43. hubspot.com
44. ibm.com
45. icloud.com
46. imdb.com
47. instagram.com
48. intuit.com
49. linkedin.com
50. live.com
51. medium.com
52. microsoft.com
53. mozilla.org
54. msn.com
55. nature.com
56. nginx.com
57. nih.gov
58. nytimes.com
59. office.com
60. openai.com
61. opera.com
62. oracle.com
63. outlook.com
64. paypal.com
65. pinterest.com
66. reddit.com
67. reuters.com
68. roku.com
69. salesforce.com
70. samsung.com
71. sciencedirect.com
72. sharepoint.com
73. shopify.com
74. skype.com
75. slack.com
76. slideshare.net
77. snapchat.com
78. sourceforge.net
79. t.me
80. theguardian.com
81. tumblr.com
82. twitch.tv
83. twitter.com
84. ubuntu.com
85. ui.com
86. unity3d.com
87. vimeo.com
88. vk.com
89. weather.com
90. whatsapp.com
91. who.int
92. wikipedia.org
93. windows.com
94. wordpress.com
95. wordpress.org
96. x.com
97. yahoo.com
98. youtube.com
99. zillow.com
100. zoom.us

Appendix E
Traces in Real-World Clouds

As discussed in Section VI-D, we anticipate that our cross-VM side channel is likely to be quieter than typical CPU-based channels in practice. To validate this claim, we captured traces on both Microsoft Azure and Alibaba Cloud to demonstrate the low noise characteristics of the GPU TLB side channel in real-world cloud environments.

Figure 20 shows traces captured in an NV12ads_A10-v5 VM on Microsoft Azure in the UAE North region across two availability zones. As shown in Figure 20 (a) and (b), the L3 TLB

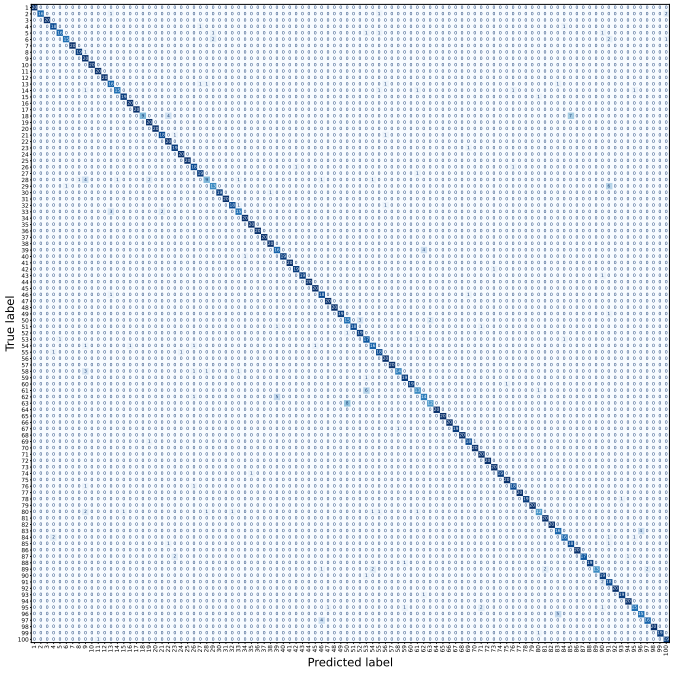
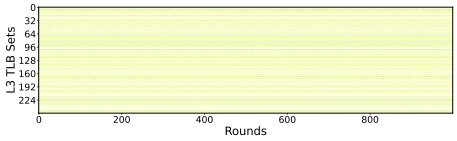
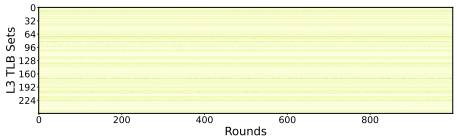


Fig. 19: Confusion matrix corresponding to testing traces collected from the victim's VM.

access patterns exhibit minimal background noise. The vast majority of TLB sets show zero or near-zero contention levels, indicated by the uniform light coloring across the figures. This clean baseline demonstrates that during typical cloud operation, the GPU TLB side channel remains largely undisturbed by other tenants' activities.



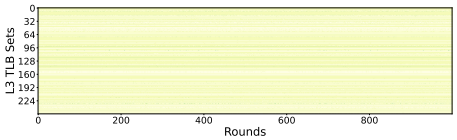
(a) UAE north region - zone 2.



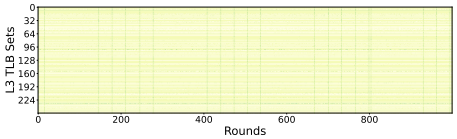
(b) UAE north region - zone 3.

Fig. 20: Traces captured on Microsoft Azure.

Figure 21 presents similar measurements from Alibaba Cloud's ecs.sgn7i-vws-m8.4xlarge VM instances. We collected traces in both the Beijing region and Hangzhou region. The captured traces also show comparably low noise levels. The sparse TLB activity observed in these traces further supports our argument that GPU TLB side channels benefit from quieter environments compared to CPU-based attacks in real-world clouds.



(a) Beijing region.



(b) Hangzhou region.

Fig. 21: Traces captured on Alibaba Cloud.