TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG

Zhenkai Zhang
Clemson University
Clemson, SC, USA
zhenkai@clemson.edu

Tyler Allen
University of North Carolina at Charlotte
Charlotte, NC, USA
t.allen@charlotte.edu

Fan Yao
University of Central Florida
Orlando, FL, USA
fan.yao@ucf.edu

Xing Gao
University of Delaware
Newark, DE, USA
xgao@udel.edu

Rong Ge
Clemson University
Clemson, SC, USA
rge@clemson.edu

ABSTRACT

Recent studies have revealed much detailed information about the translation lookaside buffers (TLBs) of modern CPUs, but we find that many properties of such components in modern GPUs still remain unknown or unclear. To fill this knowledge gap, we develop a new GPU TLB reverse-engineering method and apply it to a variety of consumer- and server-grade GPUs in Turing and Ampere generations. Aside from learning significantly more comprehensive and accurate GPU TLB properties, we discover a design flaw of NVIDIA Multi-Instance GPU (MIG) feature. MIG claims full partitioning of the entire GPU memory system for secure GPU sharing in cloud computing. However, we surprisingly find that MIG does not partition the last-level TLB, which is shared by all the compute units in a GPU. Exploiting this design flaw and learned TLB properties, we are able to construct a covert channel for data exfiltration across MIG-enforced isolation. To the best of our knowledge, this is the first attack on MIG. We evaluate the proposed attack on a commercial cloud platform, and we successfully achieve reliable data exfiltration from a victim tenant at a speed of up to 31 kbps with a very high accuracy around 99.8%. Even when the victim is using the GPU for deep neural network training, the transmission can still reach more than 25 kbps with a more than 99.5% accuracy. We propose and implement a mitigation approach that can effectively thwart data exfiltration through this covert channel. Additionally, we present a preliminary study on exploiting the access patterns of the last-level TLB to infer the identity of applications running in other MIG-created GPU instances.

CCS CONCEPTS

• **Security and privacy → Hardware reverse engineering**; **Side-channel analysis and countermeasures**.

KEYWORDS

GPU TLB; Information Leakage; NVIDIA MIG

ACM Reference Format:
Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616672

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616672

1 INTRODUCTION

Over the last decade or so, people have witnessed the rising popularity of Graphics Processing Units (GPUs). It is reported that the sales of standalone GPUs had reached 22 million units in just the first quarter of the year 2021 [44]. Aside from being needed for high-quality graphics rendering, GPUs have evolved into a type of highly programmable, massively parallel coprocessors that became broadly utilized to accelerate the execution of many compute-intensive applications ranging from medical imaging to deep learning.

The wide employment of GPUs inevitably urges a thorough study on their security implications from various perspectives, and indeed a number of works have been conducted lately [3, 18, 20, 24–26, 55, 58]. Many of these studies have signified that potential information leakage will be a major security concern if the use of a GPU is shared. Nevertheless, sharing a powerful GPU between multiple users has been an ever increasing trend in cloud computing.

To maximally fortify the security of GPU sharing (and also ensure several other desired properties like quality of service), NVIDIA has introduced a new feature named Multi-Instance GPU (MIG) since its Ampere generation in the server-grade GPUs (e.g., A100 and A30). According to NVIDIA [29, 31], MIG creates multiple GPU instances through spatially partitioning hardware so that each instance is entirely isolated with its own compute, memory, and interconnect resources. Undoubtedly, such strong isolation makes each GPU instance behave like a standalone device to provide predictable performance without being affected by others, and can help eradicate those potential cross-instance information leakage attacks that are achieved by manipulating shared hardware components. Given its great benefits, GPU-as-a-Service (GaaS) based on MIG has rapidly emerged [4, 19, 36, 40, 41, 50].

In this work, we investigate whether each GPU instance created using MIG really has separate and isolated paths through the entire memory system as claimed.[1] NVIDIA has demonstrated solid partitioning of the last-level cache, memory controllers, on-chip crossbar ports, and DRAM buses [31, 32], and thus we decide not to examine these components. Instead, we focus on one class of important but non-exemplified microarchitectural components, the translation lookaside buffers (TLBs), to understand how MIG handles them and further check if the guaranteed isolation for security is still fully credible.

To conduct the intended investigation, we need to know the details of the TLBs in such modern GPUs. However, NVIDIA does

[1] Appendix A gives the excerpts from NVIDIA documents regarding such claims.

not disclose any information about them. In the past, a number of reverse-engineering attempts have been made to find out the possible structures of the GPU TLBs [14–16, 23, 26, 52]. Even though these studies provide certain insights, the knowledge they have revealed is either too coarse-grained or only related to obsolete GPUs, and a more critical problem is that some of their results may not be accurate as shown later in this paper. Very recently, Tatar *et al.* have illustrated that it is possible to exploit TLB incoherence for very accurately reverse-engineering the TLBs of modern CPUs [47]. Inspired by their work, we determine to leverage TLB incoherence to learn the properties of TLBs in modern GPUs.

While this direction sounds feasible and promising, many challenges exist. First, we need to understand the GPU page tables and how they are used in practice. Yet, unlike the well-documented CPU page tables, detailed knowledge of the GPU page tables is barely known to the public, especially as to the GPUs in the post-Pascal generations like Turing and Ampere. Second, we need to know how to introduce incoherence to the GPU TLBs of interest. Because the involved page table entries are in the GPU memory which is independent of the host memory, a primitive for correctly editing such entries at run time should be constructed. Third, aiming at thoroughly reverse-engineering the GPU TLBs, we need to carry out the investigation from the perspective of not only data accesses but also instruction fetches. Although data pages can be easily allocated and accessed in CUDA, there is no support for us to create code pages, to which the control flow can be arbitrarily transferred.

In this paper, we have addressed all the above-mentioned challenges and successfully managed to exploit the injected incoherence to fully reverse-engineer the TLBs in those MIG-supported GPUs (and also several up-to-date consumer-grade GPUs). Interestingly, and perhaps surprisingly, we discover that MIG does not partition the GPU TLB. More specifically, we find that all the GPU instances created by MIG actually share the use of the entire last-level TLB. Even though the translation coverage of the upper-level unshared TLBs in each GPU instance is so large that accesses to the last-level TLB are rare in a normal circumstance, we show that an attacker can deliberately create contention on the shared TLB to enable a cross-GPU-instance covert channel for data exfiltration. To our knowledge, this is the first microarchitectural attack against MIG. We also briefly demonstrate that such contention may be leveraged to identify the applications running in another GPU instance.

Note that, apart from MIG, sharing a (server-grade) GPU between multiple contexts may be enabled by using the Multi-Process Service (MPS) feature in CUDA or NVIDIA's special virtualization software named the virtual GPU (vGPU) [28, 33]. Nevertheless, we need to mention that sharing a GPU between mutually distrusting users in a cloud must not be built on MPS (see Section 2.3 for its reasons), even though many recently proposed GPU data exfiltration attacks have unrealistically assumed the opposite [3, 24]. On the other hand, vGPU can be leveraged to achieve GPU sharing in multi-tenancy scenarios, but at the time of this writing, we had not found a single cloud provider really using vGPU for this purpose. By contrast, there are increasing commercial clouds relying on MIG to provide their customers with virtualized GPUs, and we have mounted our attack on one of such platforms, Puzl Cloud [40], to demonstrate the possibility of this threat in practice.

The main contributions of this paper are:

- We formulate a systematic method for comprehensively reverse-engineering the TLBs of modern GPUs. The method is built on injecting TLB incoherence, for which we have demystified the GPU page tables and managed to identify and modify corresponding page table entries in GPU memory.
- We apply this reverse-engineering method to a variety of consumer- and server-grade GPUs in Turing and Ampere generations. We disclose some previously undiscovered components (e.g., instruction TLB and TLB slices) and rectify certain inaccurate properties given in prior studies (e.g., set selection functions and TLB sub-entries).
- We discover that NVIDIA MIG does not partition the last-level TLB even though it claims full partitioning of the entire memory system for the created GPU instances. We exploit this design flaw and the reverse-engineered TLB properties to create a covert channel for cross-GPU-instance data exfiltration. It is the first microarchitectural attack breaking the isolation guarantee of MIG.
- We evaluate the proposed covert channel not only in a lab environment but also against a *real cloud system* that allows its users to rent MIG-created GPU instances. The results show that the bandwidth of the channel can reach more than 31 kbps and the accuracy is around 99.8%. Even in situations where the tenants are using the GPU to train deep neural network models, it can still reach more than 25 kbps with a more than 99.5% accuracy.
- We present our preliminary study on a side-channel attack that infers which machine learning (ML) framework is being used in another GPU instance. Using six publicly available frameworks, we show that it is possible to achieve 100% inference accuracy even without specific information about the models or datasets being operated within the framework.

We also release our reverse-engineering tools at https://github.com/0x5ec1ab/gpu-tlb.git.

Responsible disclosure: We have reported the findings to NVIDIA in October 2022. NVIDIA informed us that a ticket has been opened for its development team to investigate the issue. So far, we have not received any updates.

2 BACKGROUND

2.1 GPU Hardware Architecture

Over the years, GPUs have evolved from hardwired graphics accelerators into highly programmable, massively parallel coprocessors, whose basic compute units are called Streaming Multiprocessors (SMs). Each SM has a set of simple cores, and it executes groups of parallel threads (known as warps) in a Single-Instruction Multiple-Thread (SIMT) fashion. As technology improves, there are commonly tens of SMs in a GPU capable of running thousands of threads simultaneously. In terms of NVIDIA GPUs, SMs are further organized into two layers of abstraction based on how they are connected to other hardware components. Each pair of tightly related SMs first forms a unit called Texture Processing Clusters (TPC), and then multiple TPCs are grouped together to form a bigger unit called Graphics Processing Cluster (GPC).

To serve the memory bandwidth demands of a large amount of threads, a GPU has its dedicated memory system, which consists

of on-chip caches and off-chip GPU memory. Similar to the host memory on the CPU side, GPU memory is also based on DRAM. Currently, GDDR6 and HBM2 are the two most widely used DRAM types in NVIDIA GPUs. Note that the memory systems of the CPU and GPU are independent of each other, and before a program starts running on a GPU, the corresponding code and data need to be implicitly or explicitly copied to the GPU memory first.

It needs to be mentioned that GPU memory is virtualized, and its management is based on paging. SMs generate virtual addresses, and there is a memory management unit (MMU) on the GPU that performs virtual-to-physical address translation in accordance with page tables of the running GPU programs. Each running GPU program, terms as a GPU context, has one page table that is regulated by the GPU driver. A page table has multiple levels, and given a virtual memory address, the MMU walks through the levels to find the entry containing the desired translation information. Page table walks are expensive, and the MMU uses TLBs to cache the recently referenced page table entries to possibly avoid many such walks.

2.2 CUDA Programming Model

Originally, GPUs could only be programmed with graphics rendering APIs. As the need for leveraging GPUs to perform non-graphics computing grows, several general-purpose GPU programming models have been developed, among which CUDA is arguably the most successful and prevailing one [27].

In the CUDA programming model, code that runs on the GPU is specified in functions called kernels. A GPU program consists of one or more kernels. To have the GPU perform the computation defined in a kernel, the driver sends a corresponding kernel launch command to the GPU, which includes the configuration of the needed threads. When the kernel is launched on the GPU, it is executed as the user-configured grid of blocks of threads on SMs.

It is worth highlighting that CUDA uses a feature named Unified Virtual Addressing (UVA) to provide a single virtual address space for both the host memory and the GPU memory in the system [43]. (The feature made its debut in CUDA 4.) The CUDA runtime can determine which physical memory a pointer refers to simply from its value. With the support of UVA, another feature called Unified Virtual Memory (UVM) has been made available since CUDA 6 [13]. Traditionally, data needs to be transferred between the host memory and the GPU memory explicitly using specialized CUDA runtime functions. UVM removes this explicit data transfer requirement through automatically migrating data from one physical memory to the other when needed. Although UVA is always in force, the use of UVM needs to be specifically coded in a CUDA program.

2.3 GPU Virtualization

By default, a GPU disallows multiple kernels to execute in parallel. Multi-Process Service (MPS) is a CUDA runtime mode that allows multiple kernels to run simultaneously on the same GPU [28]. Even so, MPS is not a GPU virtualization mechanism, as it combines and executes all user kernels under the same GPU context. In a multi-tenanted system, MPS should not be used, because it does not provide address space isolation or error containment, allowing clients to interfere with each other.

NVIDIA has introduced two mechanisms for GPU virtualization. The first is vGPU, which enables multiple virtual machines (VMs) to share a GPU in the cloud [31, 33]. Unlike MPS, vGPU ensures separate GPU address spaces and execution contexts. Note that, when virtualizing a GPU, vGPU spatially partitions its memory but not compute resources. All the GPU compute resources are scheduled among VMs in a time-sharing manner.

Multi-Instance GPU (MIG) is the other virtualization mechanism available on more recent GPUs, and it enhances vGPU by spatially partitioning the GPU compute resources as well [31, 32]. According to NVIDIA, SMs in each MIG-created GPU instance have separate and isolated paths through the entire memory system. The on-chip crossbar ports, memory controllers, low-level cache banks, and DRAM buses are all assigned uniquely to an individual instance. Such strong isolation provides both security and predictable performance guarantees to users of a shared GPU. With MIG, users can treat GPU instances as if they were standalone GPUs.

2.4 TLB Incoherence

In general, there is a cache coherence protocol implemented in hardware to maintain coherent (data) cache states. However, there is usually no hardware support for keeping TLB entries coherent with their corresponding page table entries. To provide the necessary coherence, system software (that is normally the operating system) needs to perform TLB shootdowns to invalidate stale TLB entries before any of them is used for address translation.

While TLB incoherence is considered erroneous in normal operating situations, it has been exploited to form a method for accurately deducing the properties of TLBs in CPU cores [47]. To check if a TLB has a certain property, the main steps in the method are to trigger a TLB fill with an address translation, modify the corresponding in-memory page table entry (which introduces TLB incoherence), carry out experiments designed to evict the in-TLB entry according to the property, and invoke the translation again to see whether the stale one or the newly modified one is used. If the new translation is used, it implies that the expected eviction has occurred and thus confirms that the property holds. Compared with other reverse-engineering approaches that leverage timing measurements or hardware performance counters, this incoherence-based method is much more accurate and reliable.

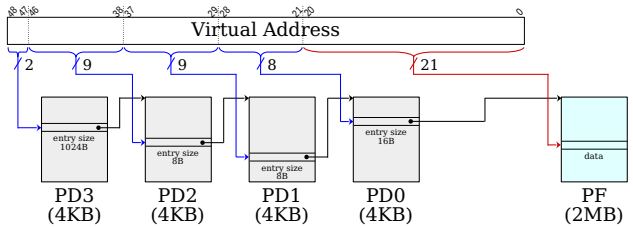
3 DEMYSTIFYING GPU PAGE TABLES

Modern GPUs use paged virtual memory. Each GPU context has its own virtual address space and operates solely on virtual addresses. As we know, there should be some page table data structure for virtual-to-physical address translation. To exploit TLB incoherence for our reverse-engineering purpose, we first need to understand the details about the GPU page tables as well as how to modify their entries at run time. However, unlike the page tables used on the CPU side whose details are all well-known, there is not too much publicly available information on the GPU page tables. (We have only found one document that is for the outdated Pascal GPUs [34].) Thus, we take efforts to unveil these details.

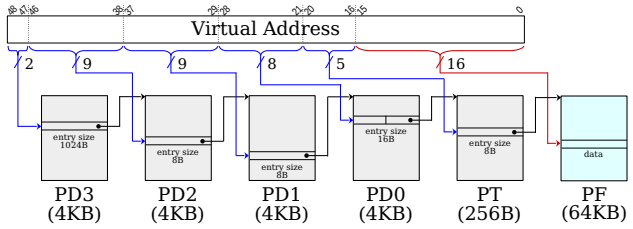
3.1 GPU Page Table Organization

The page table for a GPU context is constructed by the GPU driver. Although NVIDIA is reluctant to fully release its driver's source code, part of the code is in fact open-sourced and embedded in the official driver. From the uncompressed driver, we have found pieces

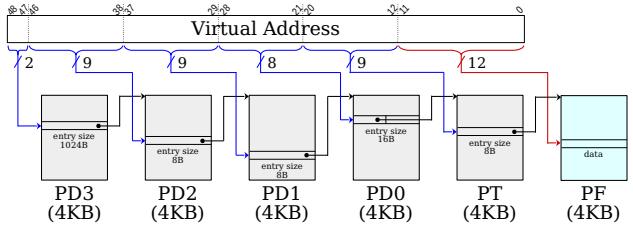
of code responsible for creating page table entries. By analyzing such driver code and the only available document [34], we gain knowledge about the basic organization of the page tables used in all the post-Pascal generations (e.g., Turing and the newest Ampere). Figure 1 illustrates the GPU page table formats.



(a) 2MB-page translation.



(b) 64KB-page translation.



(c) 4KB-page translation.

Figure 1: GPU 5-level page table hierarchy.

As shown in the figure, a modern GPU supports multiple page sizes including 4KB, 64KB, and 2MB. The page table hierarchy has 5 levels, and a 49-bit virtual address is divided to select a walking path through the hierarchy. Appendix B explains this hierarchy as well as path selection in details, and here we only focus on where the final address translations are stored, whose modifications introduce TLB incoherence.

Figure 1a shows that the address translations for 2MB pages are kept in entries of page directory 0 (PD0). Note that each PD0 entry is 16B, and if it points to a 2MB huge page, its 28 bits $[35:8]$ give the 4KB page frame number to which the 2MB page's first 4KB part is mapped. As shown in Figure 1b and 1c, if a PD0 entry does not refer to a 2MB page, it is divided into two 8B halves, each of which points to a last-level page table (PT). The PT pointed by the lower half is for translating 64KB large pages, and the PT pointed by the upper half is for translating 4KB normal pages. An entry in either type of PT has a size of 8B, and its 28 bits $[35:8]$ give the number of 4KB page frame to which the 4KB page or the 64KB page's first 4KB part is mapped.

Apparently, with respect to 2MB, 64KB, and 4KB pages, the least significant 21, 16, and 12 bits of the virtual address form the page offset, respectively. We also need to mention that, according to the driver, the newest Ampere GPUs support 512MB super pages as well. In terms of 512MB pages, PD1 serves as the last-level page table. However, we find that the driver does not use them currently.

3.2 GPU Page Table in Action

To modify the translation for a given virtual address at run time, we first need to find out where the corresponding table entry locates. Even though the page table hierarchy for a GPU context is created by the driver on the CPU side, considering performance, it is expected that this data structure should be loaded somewhere in the GPU memory for use. To verify this, we dump the GPU memory and check if such page tables can be found in the dumped memory.

Our GPU memory dumper is implemented on top of the official NVIDIA driver (version 470.63.01). More description of our dumper is given in Appendix C. From a GPU memory dump, we can indeed find and extract all the page tables, each of which is associated with a running GPU context. Due to the well-structured nature of such GPU page tables, the extraction can be efficiently and effectively achieved. Given the obtained page tables, we can accurately pinpoint the PT (or PD0) entry (and its GPU memory address) for translating a virtual address by performing the corresponding page table walk.

From the extracted page tables, we can also ascertain how pages in different sizes are used in a running CUDA program. Our observation is that, for both code and data, 2MB pages are allocated in general, except for those UVM-managed data pages whose size can be 64KB (and 4KB; see Appendix D). This observation justifies the arguments made by Nayak *et al.* in [26], but we also notice that multiple UVM-managed 64KB pages can be merged into one UVM-managed 2MB page if certain conditions are met, about which Appendix D presents more details. As demonstrated later, we can take advantage of this unveiled merging behavior to analyze more intricate TLB properties that have been overlooked in [26].

3.3 GPU Page Table Entry Modification

Knowing the specific GPU memory address of the target entry, we also need a write primitive to modify it. In our work, we achieve this through the memory-mapped I/O (MMIO) [1].

GPUs are connected to their hosts via the peripheral component interconnect express (PCIe) interface. Given a PCIe device, there can be up to 6 base address registers (BARs) for mapping the registers, I/O ports, and memory of the device into regions of the host memory address space. In terms of an NVIDIA GPU, its control registers are memory-mapped into a 16MB region defined by BAR0 for this PCIe device. Among these registers, one named PMC_BAR0_PRAMIN, which is mapped to the offset `0x1700` in the BAR0-defined region, specifies the physical GPU memory address of a 1MB GPU memory window.[2] This 1MB GPU memory is also mapped to the BAR0-defined 16MB region at the offsets ranging from `0x700000` to `0x7FFFFF` [1, 20]. By changing the address in the PMC_BAR0_PRAMIN register appropriately, we can access the

[2]This 1MB GPU memory is called private RAM instance (PRAMIN) which is used for storing states of the running GPU contexts. In our page table entry modification approach, this window of GPU memory is repurposed for gaining direct access to GPU memory.

target entry through the 1MB GPU memory window to perform modification at run time.

4 FULLY REVERSE-ENGINEERING GPU TLBS

Armed with knowledge of GPU page tables and how to modify the mappings defined in them, we have effortlessly verified that the TLBs of modern GPUs do not enforce coherence either. Therefore, we can leverage such incoherency to infer the properties of these GPU TLBs. In this section, we describe our reverse-engineering process and show that all the details about the TLBs of either consumer-grade GPUs (e.g., RTX 2080 and RTX 3080) or server-grade ones (e.g., A100 and A30) can be accurately learned in a systematic way. Table 1 lists the GPUs studied in this work.

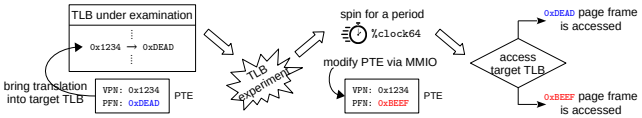


Figure 2: Generic operation for learning GPU TLB properties.

The generic operation performed in the reverse-engineering process is illustrated in Figure 2. We ❶ manipulate the state of the TLB under inspection to have a known address translation, ❷ conduct some experiments designed to use the eviction of this translation to signify certain properties, ❸ modify the translation in the GPU page table, and ❹ reference the TLB again to check if the eviction of interest has happened, which is indicated by the new translation being used. Notice that we need to rely on a host process to modify GPU page table entries in the middle of a running kernel through the MMIO (see Section 3.3), and thus synchronization is needed between the active kernel and the host. Unfortunately, CUDA does not provide any primitive for such kind of synchronization. We circumvent this problem by making the kernel repeatedly read the `%clock64` register until certain cycles have passed, during which we can modify the page table properly.

Additionally, it is worth mentioning that the address space layout randomization (ASLR) can cause some inconvenience in the reverse-engineering process, as it also randomizes the virtual address ranges allocated to the GPU driver. For a better control over the virtual addresses needed in several steps, we turn off the ASLR during the entire process.

4.1 TLB Structure

Given a GPU, we start with investigating several fundamental properties of its TLB structure, such as how many levels the TLB has, what organization a level uses, and whether a level is shared by SMs. We focus on the topmost TLB level first, and then go to examine the lower levels.

4.1.1 L1-iTLB and L1-dTLB. Although many efforts have been made to reverse-engineer the GPU TLBs [14–16, 23, 26, 52], the following basic question has never been answered: "Is the L1 TLB of a GPU split into instruction and data TLBs (i.e., similar to that of a CPU) or just unified?". Former studies had trouble to resolve it due to the lack of knowledge on how to create and execute arbitrary GPU code pages. Here we can easily find a quick answer to the question via exploiting TLB incoherence.

First, we access a very large set of UVM-managed 64KB pages in an infinite loop of a launched kernel. We make sure that the set of pages is large enough to overwhelm the TLB. (Appendix E gives details on how to check this condition.) Next, we modify the address mapping for the code page where the kernel resides. The newly mapped code page frame is just filled with zeros, and when it is executed, an illegal instruction exception will be raised to terminate the GPU context. We observe that the execution of the infinite loop on all the tested GPUs is not affected after modifying the address translation for the code page, which implies that (at least) the L1 TLB of a GPU is split into an L1-iTLB and an L1-dTLB.

To help thoroughly explore the properties related to iTLBs, it is better for us to solve the problem of fabricating GPU code pages and rendering SMs to execute them, which has never been addressed before. Since essentially there is no difference between code and data pages except for the semantics of their contents, we believe that SMs can execute a data page containing GPU instructions as long as we can divert the control flow to it. To this end, we have studied and reverse-engineered the encoding of the unconditional branch instruction used in modern Turing and Ampere GPUs.

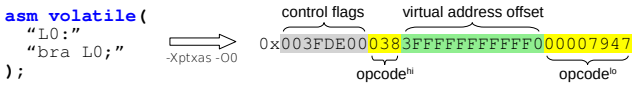


Figure 3: The encoding of an unconditional `bra` instruction whose target is itself. To keep this inline PTX assembly code from being optimized out, "`-Xptxas -O0`" is passed to nvcc.

Figure 3 shows the encoding of an unconditional branch instruction whose target is just itself (namely, self-looping). Through experiments with adding several dummy instructions before/after `bra` and moving the label `L0` around, we find that its encoding can be divided into three parts, and in terms of our goal, we are only interested in the offset part. An instruction has 128 bits and is 16-byte aligned in both Turing and Ampere GPUs. The offset part of a `bra` instruction spans 50 bits from the 33rd bit to the 82nd bit inclusive and the most significant bit in the offset part is the sign bit. (Thus, the offset in Figure 3 is -16.) We discover that the offset is the difference between the *virtual address* of the target instruction and the *virtual address* of the instruction next to the branch instruction.

With the understanding of this branch instruction, we can construct and chain simple code pages, as shown in Figure 4. Let $\{A_1, A_2, \cdots, A_N\}$ be the base addresses of N data pages. To turn a page at A_i into a code page and link it to the next page at A_{i+1}, we overwrite its first 16 bytes with the encoding value given in Figure 3 and set the offset to $A_{i+1} - (A_i + 16)$. Because executing one instruction in a code page suffices for our purpose, we can simply fill the rest of the page with zeros. At address A_0, we have a `bra` instruction like the one shown on the left in Figure 3 placed in a kernel function, and we alter its offset to $A_1 - (A_0 + 16)$. To transfer control back to the instruction at $A_0 + 16$ from the faked ones, we set the offset of the instruction in the last code page to $(A_0 + 16) - (A_N + 16)$. How to find A_0 and alter the offset of `bra` at A_0 are described in Appendix E.

We observe that if using created code pages only, we have to let N be a large number to make the modified page table entry for the kernel's code page be used; however, if we also access a large number of data pages, N just needs to be as small as 16 on all tested

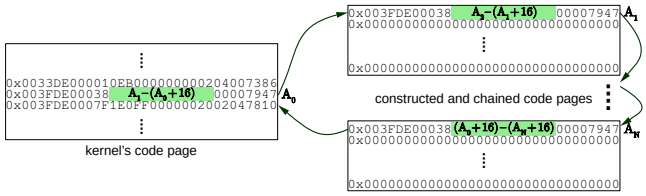


Figure 4: Fabricating GPU code pages out of data pages and integrating them into execution.

GPUs. We further repeat this for many times with created code pages at different virtual addresses, and the results are consistent. Thus, we can infer that the L1-iTLB of these GPUs has 16 entries and is fully-associative (otherwise, the smallest N evicting the target address translation should differ from 16 occasionally). Moreover, this observation signifies that there are TLBs at lower levels and they are unified (otherwise, accessing data page address translations should not affect N). Exchanging the above roles played by code and data pages, we can learn that the L1-dTLB of all these GPUs also has 16 entries and is fully-associative.

To check if the L1 TLB is shared, we run two threads of a kernel function on two SMs.[3] One thread uses a large number of code/data pages, while the other thread uses fewer than 16 code/data pages and will however experience TLB evictions if the two SMs share the L1 TLB. We find that the L1-iTLB and the L1-dTLB are private to each SM in Turing GPUs (e.g., RTX 2080), but they are shared between the two SMs of each TPC in Ampere GPUs (e.g., RTX 3080 and A100).

4.1.2 L2-uTLB. As inferred above, other than a separate L1 TLB, there should be one or more unified TLBs at lower levels. We notice that if there is only one active thread T_a, a very large number M_0 of pages (e.g., at least 3100 for RTX 3080) need to be referenced to evict a target address mapping from the entire TLB no matter which SM T_a is running on; however, if there is a second thread T_b running on an SM in another GPC[4] and T_b also accesses M_0 pages, T_a just needs to use M_1 pages to cause the page table walker to retrieve the modified translation, where M_1 is significantly less than M_0 but still much greater than 16. This observation indicates that at least two levels of unified TLBs, which we call L2-uTLB and L3-uTLB, exist below the first level (because if there is only one unified TLB level beneath L1, M_1 should be 16).

To gain insights into the L2-uTLB, we use the address mapping for an arbitrarily selected page as the target and try to shrink M_1 by taking out the pages one-by-one to check whether the target can still be evicted – if so, removing it for good; otherwise, returning it. As the TLBs at lower levels are unified, we use data pages for convenience. Due to the reasons discussed later, we use a sequence of 64KB pages managed by UVM and the virtual address of each data page is separated by `0x100000` from that of the next page in the sequence. These pages are linked together using pointer chasing. To force any related translations out of the L1-dTLB, we also keep the very first 16 pages of the ones removed from the sequence and iterate them after each page in the sequence is accessed.

[3]By requesting for a large amount of shared memory when launching a kernel function, we can force each SM to execute a single thread block. The `%smid` register gives the SM identifier on which the thread block is running.

[4]For Turing GPUs, T_b can run on an SM in the same GPC.

Interestingly, this process always has 8 pages left in the sequence to form a set for evicting the target no matter which GPU in Table 1 is used. We repeat this process with different targets, and we find that S_{L2} disjoint sets can be derived and any target can be evicted by exactly one of the S_{L2} sets, where S_{L2} is a power of 2 and also $S_{L2} \times 8 \approx M_1$. Therefore, we reach a conclusion that the L2-uTLB is 8-way set-associative in all the tested GPUs. Notice that the number S_{L2} of sets is microarchitecture- and/or class-dependent (see Table 1). We further accurately reverse-engineer how the L2-uTLB sets are indexed, whose details are described in Section 4.2.

To find if the L2-uTLB is shared between SMs, we adopt the method for testing L1 TLB sharing with only some slight changes. Rather than fewer than 16 pages, the thread checking TLB evictions uses more than 16 but fewer than $S_{L2} \times 8$ pages. We discover that the L2-uTLB is private to each SM in Turing GPUs, but it is shared between the SMs forming a GPC in all the Ampere GPUs.

4.1.3 L3-uTLB. To continue reverse-engineering the TLB at the next level, we just leverage one active thread running on an SM. We again start with choosing a data page and use its address translation as the target. However, instead of assembling a sequence of pages whose virtual addresses are consecutively separated by `0x100000`, we use pages whose address translations share the same L2-uTLB set with the target (using the hash function described in Section 4.2 for identifying them). We need to use as many such pages (which are also chained together) until the target gets evicted, and then we prune them by applying the above-mentioned removing-or-returning procedure. Note that we also pick 16 pages whose address translations are evenly distributed in four other L2-uTLB sets than the target's. They are iterated after each page in the sequence is accessed to help flush the L1-dTLB. Moreover, we collect the very first 8 pages that are removed from the sequence and iterate them as well after a page in the sequence is accessed to help flush the corresponding L2-uTLB set.

For any consumer-grade GPU being inspected, we observe that 8 pages are always left to form an eviction set. Eventually, we can derive S_{L3} such sets and any target can be evicted by exactly one of the S_{L3} sets from the L3-uTLB, where S_{L3} is a power of 2 and microarchitecture-dependent (see Table 1). As $S_{L3} \times 8 \approx M_0$, we can know that this 8-way set-associative L3-uTLB is the last level.

In contrast to consumer-grade GPUs, MIG-supported ones (which are all Ampere server-grade GPUs) seemingly have 8 more entries in an L3-uTLB set because there are always 16 pages left to form an eviction set. We can derive S_{L3} eviction sets where S_{L3} is a power of 2, and any given translation can be evicted by exactly one of the S_{L3} sets from the L3-uTLB. (Recall that the sequence for deriving an eviction set consists of pages whose address translations are cached in the same L2-uTLB set as the target.) However, there are two reasons making us believe that the associativity cannot be 16. First of all, we notice that $S_{L3} \times 16 \approx M_0 \times 2$. Second, given any target, if we just access 8 or more but not all elements in the corresponding eviction set, we surprisingly find that further using one of many other eviction sets can then evict the target from the L3-uTLB; but if fewer than 8 elements in the target's eviction set are referenced, no matter how we access the other $S_{L3} - 1$ eviction sets, the target cannot be forced out of the L3-uTLB. We also discover that the S_{L3} eviction sets can be evenly partitioned into two disjoint

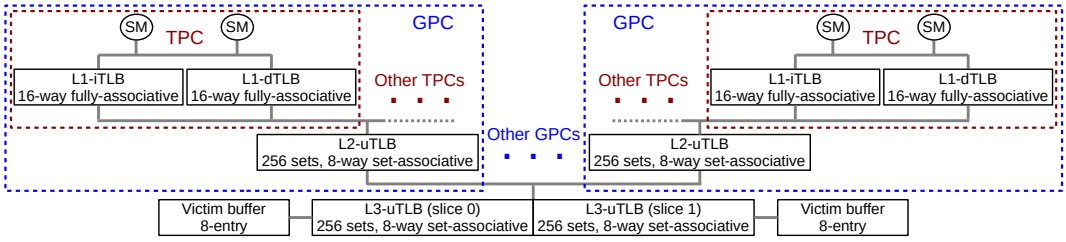


Figure 5: TLB structure of A30 and A100 GPUs.

groups and this "cross-set eviction" can be achieved if and only if two sets belong to the same group. These observations lead to the conclusion that the L3-uTLB in MIG-supported GPUs is still 8-way set-associative and it is (physically or just logically) split into two slices; and each slice has an 8-entry victim buffer shared by all the TLB sets in the slice.

Using the same method as for checking L1/L2 TLB sharing, we find that the L3-uTLB is *shared by all the SMs* in both the consumer-grade GPUs and MIG-supported ones. Figure 5 illustrates the reverse-engineered TLB structure of A30 and A100 that are two server-grade GPUs supporting MIG.

4.2 Set (and Slice) Selection Hash Functions

In spite of the fact that many efforts have been made to reverse-engineer CPU and GPU TLBs, we find most of the approaches to deriving the set selection functions either vague or ad hoc [9, 17, 26, 47]. Here we propose a systematic approach that can effectively and elegantly determine the GPU TLB set selection hash functions. Instead of acquiring eviction sets for each TLB set, our approach just needs one and only one eviction set for an arbitrarily chosen TLB set.

The approach consists of two primary steps. As we know, TLBs use virtual addresses to index their sets. The first step of our approach is to identify which bits in the virtual address are used in the set selection hash function. Without loss of generality, we assume that a TLB has S sets and a set has W ways. Given a page whose virtual address is A_0, we prune a sequence of pages to derive a minimal set $\{A_1, A_2, \cdots, A_W\}$ whose address translations evict that of A_0. From this eviction set, we select an address A_i arbitrarily and create a set of B new addresses $\{A_i^0, A_i^1, \cdots, A_i^{B-1}\}$, where B is the number of bits in virtual address and A_i^j is generated by flipping the bit at position j of A_i. For each A_i^j, we check if it can replace A_i in the original set to achieve eviction. If $\{A_1, \cdots, A_i^j, \cdots, A_W\}$ is still an eviction set, it simply means that the bit at position j is not involved in selecting TLB sets (because if it is, with all other bits unchanged, a different set number should be given by the function).

After the first step, we will acquire a set of H bit positions $\{p_0, p_1, \cdots p_{H-1}\}$ involved in the set selection hash function. The second step of our approach is to identify which of them should be XOR'ed. Since there are S sets in the TLB, the set selection function certainly consists of $\log_2 S$ XOR lines (e.g., the L2-uTLB set selection function of RTX 3080 has 7 XOR lines as shown in Figure 6). We find that if a bit is involved in a TLB set selection hash function of either CPU or GPU, it is normally associated with one XOR line only. This implies that if flipping two of the H bits at the same time does not change the selected TLB set, these two bits are in the same XOR line. We continue with A_i chosen in the first step and create a set of $\binom{H}{2}$ new addresses $\{A_i^{p_0p_1}, A_i^{p_0p_2}, \cdots, A_i^{p_{H-2}p_{H-1}}\}$, where A_i^{jk} is generated by flipping the bits of A_i at positions j and k together. For each A_i^{jk}, we examine if $\{A_1, \cdots, A_i^{jk}, \cdots, A_W\}$ can still achieve the desired eviction. If so, we know that the bit at position j is connected to the bit at position k by an XOR. After all such pairs are identified, we combine the mutually XOR'ed ones (e.g., $j \oplus k$, $k \oplus l$, and $l \oplus j$ are coalesced into $j \oplus k \oplus l$), and in the end there should be $\log_2 S$ combined groups, which essentially form the hash function.

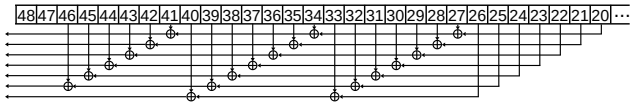
Notice that as we start flipping bits in these two steps, the more significant position a bit holds, the larger virtual address gap the flipping will produce. To make sure that all the created addresses can be legally accessed in the GPU context, we take advantage of the `cudaMallocManaged()` function to reserve as much address subspace for the GPU context as we can.[5]

4.2.1 Set Selection Functions w.r.t. *using 64KB Pages.* With the proposed approach, we can rapidly and accurately learn the hash functions used for selecting the L2-uTLB and L3-uTLB sets. Recall that the L3-uTLB of MIG-supported GPUs is sliced, and here the learned L3-uTLB set selection function for those GPUs is to choose a set within a slice. We will also show how slices are selected later.

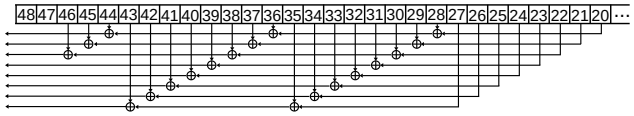
At first, we still use UVM-managed 64KB pages. Figure 6 shows the recovered functions for Ampere GPUs, from which we can see that the set selection hash functions XOR bits in a regular fashion from the 21st bit to the 47th bit of the virtual address. Extending the naming rules given in [9], we call the functions shown in Figure 6 XOR-7_{20}^{46} and XOR-8_{20}^{46} respectively, where 7 or 8 denotes the number of XOR lines, the subscript denotes the starting bit position, and the superscript denotes the ending bit position (inclusive). We discover that if 64KB pages are used, the 21st to 47th bits are consistently involved to build such functions for L2-uTLB and L3-uTLB in all the tested GPUs. Appendix F shows the functions used in Turing GPUs. Moreover, we find that this set of bits is used to build these functions in old Pascal GPUs as well, and hence the functions recovered in [26] are inaccurate.

It has to be mentioned that, from the perspective of a GPU, a virtual addresses has 49 bits, but in practice its 48th and 49th bits are always zero. This is because the virtual addresses used in a GPU context are allocated from the user address space of the host process, whose upper bound is `0x7FFFFFFFFFFF` in the commonly used systems. Therefore, we cannot flip these two high-order bits to

[5] On our system, we first leverage the `cudaMallocManaged()` function to "hoard" more than 69TB virtual address subspace from `0x100000000000` to `0x555554000000`, and then use the function again to "hoard" more than 41TB subspace from `0x560000000000` to `0x7fffc8000000`. The addresses in-between are used by the host process.



(a) XOR-7^{46}_{20} is used for indexing L2-uTLB in Ampere consumer-grade GPUs (e.g., RTX 3060 and RTX 3080).



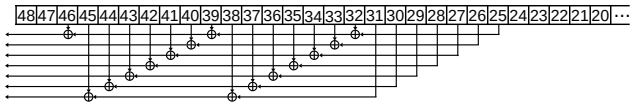
(b) XOR-8^{46}_{20} is used for indexing L2-uTLB and sliced L3-uTLB in Ampere server-grade GPUs (e.g., A30 and A100) as well as indexing L3-uTLB in consumer-grade ones.

Figure 6: TLB set selection hash functions in terms of Ampere GPUs when 64KB pages are used.

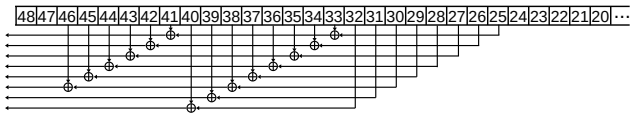
find out whether they are involved in the selection hash functions or not. Even if they were, it would not impose any impacts as they are constantly zero.

4.2.2 Set Selection Functions w.r.t. *using 2MB Pages.* In [17], Koschel *et al.* have revealed that the TLB set selection functions in Intel CPUs may change when pages of different sizes are used. We should examine if this is also true in terms of NVIDIA GPU TLBs. To this end, we use 2MB data pages to perform the proposed reverse-engineering approach.

Although the `cudaMalloc()` API function allocates GPU memory directly in the form of 2MB pages, it does not provide us with the control over sparsely distributing virtual addresses. To acquire the needed 2MB pages as well as the flexibility of manipulating virtual addresses, we take advantage of the page merging behavior of the UVM that is described in Section 3.2. Eventually, we create a sequence of UVM-managed 2MB data pages for deriving an eviction set, where the virtual address of each 2MB page is separated by `0x2000000` from that of the next page in the sequence. (The reason for this 32MB offset is described in Section 4.4.) From the derived eviction set, we can use our two-step method to recover the functions.



(a) XOR-7^{46}_{25} is used for indexing L2-uTLB in Ampere consumer-grade GPUs (e.g., RTX 3060 and RTX 3080).



(b) XOR-8^{46}_{25} is used for indexing L2-uTLB and sliced L3-uTLB in Ampere server-grade GPUs (e.g., A30 and A100) as well as indexing L3-uTLB in consumer-grade ones.

Figure 7: TLB set selection hash functions in terms of Ampere GPUs when 2MB pages are used.

Interestingly, similar to that on the CPU side, the TLB set selection functions on the GPU side also change when pages of different sizes are used. As an example, Figure 7 illustrates the functions recovered for Ampere GPUs. Compared with the ones shown in Figure 6, we can observe that the 21$^{\text{st}}$ to 25$^{\text{th}}$ bits of virtual addresses are no longer involved in the hash functions. (The number of TLB sets at L2 or L3 is not changed.) Hence, according to our naming convention, we have the XOR-7^{46}_{25} and XOR-8^{46}_{25} functions shown in Figure 7. We confirm that this set of bit positions is used to index TLB sets in other GPUs as well when 2MB pages are used.

4.2.3 Slice Selection Function. To learn the hash function used for choosing one of the two L3-uTLB slices in MIG-supported GPUs, we adapt our method for finding set selection hash functions. First, we choose two 16-element eviction sets E_0 and E_1 at random from one group derived by the specific procedure described in Section 4.1.3. From E_0, we arbitrarily extract 9 elements $\{A_0, A_1, \cdots, A_8\}$. Recall that the translation for A_0 can be evicted from the L3-uTLB after $\{A_1, \cdots, A_8\}$ and E_1 are accessed in order. Then, we start flipping bits. Here, instead of focusing on one address, we flip the same bit position in all the $\{A_0, A_1, \cdots, A_8\}$. Given a bit position j, the address translations for $\{A^j_0, A^j_1, \cdots, A^j_8\}$ are still mapped to one single L3-uTLB set, but whether the translation for A^j_0 can be evicted by accessing $\{A^j_1, \cdots, A^j_8\}$ and E_1 depends on if j is involved to select the slice. If j is not used in the slice selection hash function, the translation for A^j_0 is mapped to the same slice as before and it will still be evicted from the L3-uTLB after accessing $\{A^j_1, \cdots, A^j_8\}$ first and then E_1. Otherwise, the translation for A^j_0 is cached in the other slice and the eviction of interest cannot be achieved.

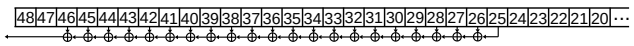


Figure 8: L3-uTLB slice selection hash function XOR-1^{46}_{25}.

Figure 8 gives the reverse-engineered function for choosing L3-uTLB slice in A30 and A100. Interestingly, we find that, unlike a set selection function, the slice selection function is always the same no matter if 64KB or 2MB pages are in use.

4.3 Replacement Policy

Following the analysis conducted in [47], we also leverage the *permutation vectors* introduced in [2] to learn the replacement policy implemented at each TLB level. Given a target TLB, we prime one of its sets (or the whole TLB in the case of the fully-associative L1-iTLB/dTLB) using a sequence of address mappings for $\{A_{W-1}, \cdots, A_1, A_0\}$. We manage to reference the address translation for A_0 again in the primed TLB set, and then reference new address translations mapped to the same TLB set to observe the ordering in which the translations in the primed set are evicted. This ordering is denoted as π_0 and is the permutation vector in terms of reusing the translation for A_0. The whole process is repeated to learn other permutation vectors $\pi_1, \cdots, \pi_{W-1}$ as well.

$$\pi_0 = \{0,1,2,3,4,5,6,7\} \quad \pi_1 = \{1,0,2,3,4,5,6,7\} \quad \pi_2 = \{2,0,1,3,4,5,6,7\} \quad \pi_3 = \{3,0,1,2,4,5,6,7\}$$
$$\pi_4 = \{4,0,1,2,3,5,6,7\} \quad \pi_5 = \{5,0,1,2,3,4,6,7\} \quad \pi_6 = \{6,0,1,2,3,4,5,7\} \quad \pi_7 = \{7,0,1,2,3,4,5,6\}$$

Figure 9: Permutation vectors for an L2-uTLB/L3-uTLB set.

Figure 9 presents the permutation vectors derived for the 8-way set-associative L2-uTLB and L3-uTLB with respect to their replacement policy in all the GPUs studied in this work. According to [2], this form of permutation vectors signifies the LRU replacement

policy. Similarly, the permutation vectors derived for the L1-iTLB and L1-dTLB also indicate that they use the LRU replacement policy in all the tested GPUs.

Regarding the victim buffer of each L3-uTLB slice in MIG-supported GPUs, the policy is still LRU. Yet, we also observe a special behavior of the victim buffer that address translations evicted from different L3-uTLB sets actually cannot be held in the buffer at the same time. In other words, if a translation is evicted from a set into the buffer while the buffer presently contains translations evicted from another set, the buffer will be flushed first. This special behavior means that 9 elements of E_1 suffice to achieve the desired eviction in Section 4.2.3.

4.4 TLB Sub-Entries

Since applications running on GPUs usually utilize very large working sets, it will hurt performance badly if the TLB reach is too small. TLB coalescing has been proposed to expand the translation coverage of the TLBs on both the CPU and GPU sides [35, 37–39]. In [26], Nayak *et al.* claim that NVIDIA GPUs enforce TLB coalescing, which combines 16 address translations to occupy just one TLB entry if the corresponding virtual page numbers are consecutive and the mapped physical page frames are also contiguous. However, the results of our experiments do not agree with this claim.

In our experiments, we selectively introduce and/or remove contiguity in virtual and/or physical addresses[6], and we notice that address translations reside in one L2-uTLB or L3-uTLB entry *as long as* the virtual base addresses of the corresponding pages are (1) within the same 1MB-aligned address range if the pages are 64KB, or (2) within the same 32MB-aligned range if the pages are 2MB. This observation disproves the existence of dynamically coalescing TLB entries and explains why we separate the base addresses by `0x100000` (i.e., 1MB) and `0x2000000` (i.e., 32MB) when using sequences of 64KB and 2MB pages respectively to perform the above-mentioned tasks. Moreover, it casts light on why the starting bit positions in the recovered TLB set selection hash functions are 20 and 25 when using 64KB and 2MB pages.

Instead of TLB coalescing, we conjecture that there are 16 sub-entries in one L2-uTLB or L3-uTLB entry, and they have a one-to-one mapping relationship with the address translations for 16 pages of size 64KB or 2MB located in the same 1MB- or 32MB-aligned range. If any sub-entry encounters an eviction, the rest of them are also invalidated. Interestingly, we find that the entries of L1-iTLB and L1-dTLB do not have such sub-entries.

4.5 Other Properties and Summary

We also investigate several other important properties of GPU TLBs and give the results here. We leave out the detailed steps as they are similar to the corresponding ones specified in [47]. Table 1 summarizes the inferred TLB properties for the GPUs studied in this work.

Inclusivity and Exclusivity. We find that the L2-uTLB is neither inclusive nor exclusive in all the inspected GPUs. The same is also true for the L3-uTLB.

Reinsertion. We find that an L2-uTLB hit is reinserted into the L1 and an L3-uTLB hit is also reinserted into the L2 and L1. We do not find that an L1 TLB hit reinserts the translation into the L2 or L3 after either is flushed. We also do not notice that the translations evicted from an upper level are reinserted into its lower level(s).

Table 1: Summary of reverse-engineered TLB properties.

TLB Property	GTX 1650 (Turing)	RTX 2080 (Turing)	RTX 3060 (Ampere)	RTX 3080 (Ampere)	A30 (Ampere)	A100 (Ampere)
L1-iTLB						
No. of sets	1	1	1	1	1	1
No. of ways	16	16	16	16	16	16
Subs per entry	0	0	0	0	0	0
Replacement	LRU	LRU	LRU	LRU	LRU	LRU
Shared	✗	✗	TPC	TPC	TPC	TPC
Hit ↦ L2/L3†	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗
L1-dTLB						
No. of sets	1	1	1	1	1	1
No. of ways	16	16	16	16	16	16
Subs per entry	0	0	0	0	0	0
Replacement	LRU	LRU	LRU	LRU	LRU	LRU
Shared	✗	✗	TPC	TPC	TPC	TPC
Hit ↦ L2/L3†	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗
L2-uTLB						
No. of sets	32	32	128	128	256	256
No. of ways	8	8	8	8	8	8
Subs per entry	16	16	16	16	16	16
Set selection*	XOR-$5^{46}_{20/25}$	XOR-$5^{46}_{20/25}$	XOR-$7^{46}_{20/25}$	XOR-$7^{46}_{20/25}$	XOR-$8^{46}_{20/25}$	XOR-$8^{46}_{20/25}$
Replacement	LRU	LRU	LRU	LRU	LRU	LRU
Shared	✗	✗	GPC	GPC	GPC	GPC
Inclu/Exclu	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗
Hit ↦ L1/L3†	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗
L3-uTLB						
No. of sets	128	128	256	256	256+256	256+256
No. of ways	8	8	8	8	8	8
Subs per entry	16	16	16	16	16	16
Set selection*	XOR-$7^{46}_{20/25}$	XOR-$7^{46}_{20/25}$	XOR-$8^{46}_{20/25}$	XOR-$8^{46}_{20/25}$	XOR-$8^{46}_{20/25}$	XOR-$8^{46}_{20/25}$
Replacement	LRU	LRU	LRU	LRU	LRU	LRU
Shared	All SMs	All SMs	All SMs	All SMs	All SMs	All SMs
Inclu/Exclu	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗	✗/✗
Hit ↦ L1/L2†	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓
Sliced	✗	✗	✗	✗	✓	✓
No. of slices	-	-	-	-	2	2
Slice selection	-	-	-	-	XOR-1^{46}_{25}	XOR-1^{46}_{25}
Victim buffers	0	0	0	0	2	2
Buf entries	-	-	-	-	8	8

† We use ↦ to denote the reinsertion behavior. For example, "Hit ↦ L1/L3" in terms of the L2-uTLB asks whether an L2-uTLB hit is reinserted into the L1 or reinserted into the L3.
* In the subscript of each selection XOR function, the first number is the starting bit position when using 64KB pages and the second number is the starting bit position when using 2MB pages.

5 DATA EXFILTRATION AGAINST MIG

As aforementioned, NVIDIA has introduced the MIG feature in its more recent server-grade GPUs (e.g., A100 and A30), and advocates the use of this feature in GPU cloud computing for quality of service and security. According to NVIDIA [31], MIG partitions *all* the components of the memory system in such a GPU to provide users with extremely strong and secure isolation in the multi-tenancy scenarios. In this section, we challenge this confinement guarantee by showing that data exfiltration across MIG-created GPU instances can in fact be easily achieved.

5.1 Threat Model

We assume that there are two communicating entities, a sender and a receiver, capable of using two GPU instances created by MIG on a server-grade GPU. The sender has access to some sensitive data, and attempts to transmit this piece of data to the receiver through a covert channel. For example, the sender may be a trojan embedded in a ML framework for stealing trained DNN models. The receiver is on the other end of the covert channel for collecting the transmitted data. The sender and the receiver are both unprivileged. Given the isolation promise of MIG, data exfiltration between GPU instances created by MIG is theoretically unattainable (at least from the perspective of manipulating logical resources).

[6]To introduce or remove contiguity in physical addresses, we use our PTE modification primitive to change the mapped page frames.

5.2 Cross-GPU-Instance Covert Channel

In Section 4, we have reverse-engineered the TLB properties of server-grade GPUs supporting MIG (e.g., A30 and A100). At the beginning, the reverse-engineering process was carried out without MIG being involved. However, when performing the process on GPU instances created by MIG, we surprisingly retrieved the same properties as before at each level. Although those of the L1 and L2 TLBs are supposed not to change (since a GPC belongs to one GPU instance as a whole), the unchanged L3-uTLB properties indicate that this last-level TLB is not partitioned in spite of NVIDIA claiming that the entire memory system is partitioned by MIG. Further eviction experiments with separate GPU instances verify that the L3-uTLB is indeed always shared.

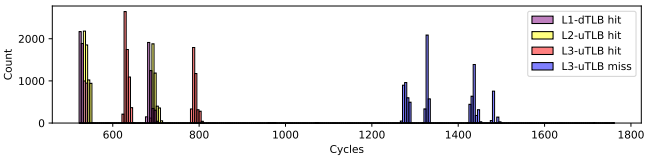


Figure 10: A30 GPU memory access times in the cases of TLB hits and misses. The histogram for A100 is almost the same.

Essentially, contention on the shared L3-uTLB can be exploited to construct a covert channel for data exfiltration that MIG intends to eradicate. The transmitted data can be inferred via measuring GPU memory access times. Figure 10 shows the timing distribution of GPU memory accesses on L1-dTLB hits, L2-uTLB hits, L3-uTLB hits, and L3-uTLB misses. For each case, 10,000 GPU memory accesses are measured and we also ensure that the accesses do not incur data cache misses. We can observe that almost all the memory accesses suffering L3-uTLB misses need more than 1250 cycles to finish while the memory accesses whose address translations can be found in the TLB hierarchy just need less than 850 cycles. It is also interesting to see that we may not be able to easily use timing to distinguish L1 hits from L2 hits (even L3 hits) and L2 hits from L3 hits on such MIG-supported GPUs, which highlights the advantages of relying on TLB incoherence for accurately reverse-engineering the TLB hierarchy.

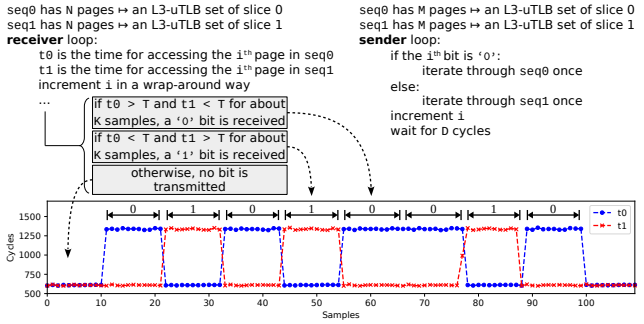


Figure 11: Protocol for cross-GPU-instance covert communication.

We create a protocol illustrated in Figure 11 to achieve the desired cross-GPU-instance covert communication. The receiver leverage the reverse-engineered slice and set selection hash functions to derive two sequences, `seq0` and `seq1`, of N pages, where $8 < N < 16$. For `seq0`, the virtual-to-physical address translations of its N pages will use a single arbitrarily chosen L3-uTLB set of slice 0. Because the set selection function for an L3-uTLB slice is exactly the same as that for an L2-uTLB (see Table 1), these N translations will use a single L2-uTLB set as well. The `seq1` is constructed similarly with respect to an L3-uTLB set of slice 1. The receiver constantly loops through its `seq0` and `seq1`. In a loop iteration, it first measures the time `t0` for accessing the page in `seq0` and then measures the time `t1` for accessing the page in `seq1`. Since the LRU replacement policy is used at each TLB level, we can expect that when a page in `seq0` or `seq1` is accessed (after the TLBs have been warmed up), its address translation cannot be in the L1 and L2 TLBs but can be found in the L3 slice's victim buffer if there is no one currently flushing that buffer. In this case, `t0` or `t1` shall be in the range of 600 to 850 cycles. Otherwise, if someone else has evicted the translation from the victim buffer, `t0` or `t1` shall be more than 1250 cycles.

Accordingly, the sender can create contention on the shared L3-uTLB's victim buffers to manipulate the receiver's `t0` and `t1`. To this end, the sender also derive two sequences, `seq0` and `seq1`, of M pages, where $M > 8$, following the same rules as described above for the receiver. When a bit '0' needs to be sent, the sender sequentially accesses each page in its `seq0` for one time, which can regularly force the translations referenced by the receiver out of the slice 0's victim buffer while not affecting the slice 1's at all. Thus, given a reasonable threshold T, say 1000, the receiver will observe `t0` $> T$ and `t1` $< T$ for around K samples during a '0' being sent, where K depends on the number of pages in sender's `seq0`. Likewise, when a bit '1' needs to be transmitted, the sender iterates over its `seq1` for one time and the receiver will observe `t0` $< T$ and `t1` $> T$ for around K samples. Furthermore, we may let the sender wait for D cycles between transmitting two bits to help synchronization, especially when many bits are continuously sent. Notice that the sender's `seq0` and the receiver's `seq0` do *not* need to target an identical L3-uTLB set in slice 0. If the same set is chosen by both, the contents in the victim buffer will be routinely evicted due to the translations introduced by the sender; otherwise, the victim buffer will be continually flushed due to the special behavior stated in Section 4.3. The same is also true for `seq1` of both sides.

An example is presented at the bottom of Figure 11, where the sender and receiver use two separate GPU instances created by MIG on an A30, and the sender has $M = 11$ and $D = 0$. From the given trace of <`t0`, `t1`> samples in this example, it is not hard to find that the received bits are "01010010".

5.3 Evaluation

Before evaluating our cross-GPU-instance covert channel in a commercial cloud environment, we examine its performance limits in a lab environment that is set up on a Dell PowerEdge R740 server (with two Xeon Silver 4114 processors and 64GB DDR4 2400 memory). We installed an A30 GPU in the server and let MIG create two instances, each of which has 28 SMs and 12GB GPU memory. The sender and receiver directly use the MIG-created GPU instances.

We read `/dev/random` to generate a piece of data having 65,536 bits (i.e., 8KB). We design the sender to transmit 8192 bits (i.e., 1KB) each time with a 16-bit header. We adjust the bandwidth of the covert channel and send the piece of data 10 times. We evaluate the error rate using the Levenshtein edit distance between the transmitted and received bits as the metric [21, 57]. The bandwidth of the

channel is mainly determined by how long each loop iteration of the sender takes. According to our protocol, the fastest bit-sending iteration we can obtain is to use the smallest M and D (i.e., 9 and 0), but we notice that a slightly larger M will not affect the sending speed too much, and thus we will stick to $M = 11$ for simplicity. On the receiver side, we also use $N = 11$.

Table 2: Channel bandwidths and error rates in a lab environment.

Bandwidth	Delay D	Max. Err. Rate	Min. Err. Rate	Avg. Err. Rate
31.47 kbps	0	0.24%	0.17%	0.21%
25.04 kbps	12,000	0.22%	0.09%	0.17%
20.71 kbps	24,000	0.20%	0.10%	0.16%
17.52 kbps	36,000	0%	0%	0%
15.39 kbps	48,000	0%	0%	0%

Starting from 0, we gradually increase the delay cycles D inserted between transmitting two bits. As each GPU memory access having a miss in the TLB hierarchy needs more than 1250 cycles, each time we add 12,000 to D (i.e., about 8~10 such memory accesses). The evaluation results are illustrated in Table 2. We can observe that the bandwidth of this channel can reach up to 31.47 kbps with a 0.21% error rate on average. We have identified two causes for the errors. The main cause is that when many 0's or 1's are consecutively sent, the receiver may estimate the number of these bits inaccurately. The other one is that when two opposite bits are sent, a long transition with `t0` $> T$ and `t1` $> T$ may appear sometimes, which makes the signal of our interest too short to be recognized as a valid bit. Inserting delay between sending bits reduces the speed of the channel, but it can help mitigate the two problems causing errors. Particularly, when D becomes large enough, there will be multiple samples with `t0` $< T$ and `t1` $< T$ separating a bit's signal from its neighbors', and such samples can be leveraged to attain a synchronization purpose. As shown in the last two rows of Table 2, after knowing when a bit's signal starts and ends, the receiver can retrieve the messages without any error. We find that the fastest speed such error-free data exfiltration can reach is 18.04 kbps when D is 34,000 cycles.

To demonstrate the practicality, we perform the evaluation on a commercial cloud platform, Puzl Cloud [40], which relies on MIG to virtualize GPUs and provide secure GPU sharing to its users. We registered multiple accounts, and managed to make three tenants, `A`, `B`, and `C`, co-resident on a single A100, where each of `A` and `B` occupies a quarter of the GPU and `C` uses a half of the GPU. We assume that `A` is an attacker who runs the receiver and `B` is a victim whose GPU program has been implanted with the sender. We re-evaluate the bandwidth and error rate in very realistic scenarios where the rent GPUs are used to train DNN models.

Even though an A30 is used in our lab environment while an A100 is used in this cloud environment, we do not find any need for changing the channel's configuration, which signifies the flexibility of this covert channel. As a baseline, we first perform the evaluation without `C` and `B` running any programs. From the top part of Table 3, we can observe that the results in two representative setups (that are the fastest one with $D = 0$ and an error-free one with $D = 36,000$) does not show much difference compared with their counterparts in Table 2.

Next, we let the tenant `C` train a ResNet-50 DNN model on the CIFAR-100 dataset using PyTorch. This training process constantly uses 4GB GPU memory. The middle part of Table 3 gives the evaluation results under this circumstance. Although training a large DNN model is very memory-intensive, we can find that the performance of our covert channel is not affected. The reason for such is that the reach of each GPC's L2-uTLB in MIG-supported GPUs is up to 64GB[7] and when executing a normal GPU program, the need for L3-uTLB accesses is extremely rare after the warm-up phase.

Table 3: Channel bandwidths and error rates in a commercial cloud.

Baseline – `C` and `B` stay idle				
Bandwidth	Delay D	Max. Err. Rate	Min. Err. Rate	Avg. Err. Rate
31.36 kbps	0	0.34%	0.12%	0.23%
17.48 kbps	36,000	0%	0%	0%
`C` trains a ResNet-50 model on the CIFAR-100 dataset				
Bandwidth	Delay D	Max. Err. Rate	Min. Err. Rate	Avg. Err. Rate
31.30 kbps	0	0.33%	0.14%	0.22%
17.35 kbps	36,000	0%	0%	0%
`B` trains a ResNet-50 model on the CIFAR-100 dataset				
Bandwidth	Delay D	Max. Err. Rate	Min. Err. Rate	Avg. Err. Rate
25.42 kbps	0	0.89%	0.21%	0.47%
14.39 kbps	36,000	0.76%	0%	0.28%

In reality, it is possible that the GPU instance of the victim is being actively used for some legitimate workload during the malicious covert communication. Thus, we need to evaluate the performance of our covert channel in such a situation. To this end, we let `B` train a ResNet-50 model on the CIFAR-100 dataset as well. The evaluation results are given in the bottom part of Table 3, from which we can see that the performance is negatively impacted. Nevertheless, the average error rate is still very low – less than 0.5% if no delay is inserted and less than 0.3% if 36,000 delay cycles are added. Furthermore, in terms of a setup, the total clock cycles needed for sending a certain amount of bits are increased, which leads to a reduced bandwidth (e.g., when $D = 0$, there is about 6 kbps reduction).

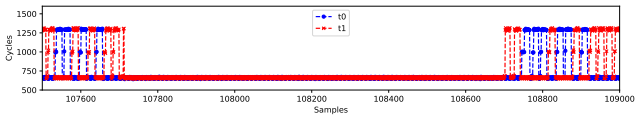


Figure 12: A snippet of a <t0, t1> trace when training a DNN and the sender concurrently run on the victim's GPU instance.

A major reason for such decreased bandwidth and increased error rate is GPU context switching governed by the NVIDIA time-sliced scheduler [51]. Figure 12 illustrates this case, from which we can observe that the sender's context is switched out between samples 107711 and 108702. Even though training a large DNN model and sending data through our covert channel run concurrently on the same GPU instance, the signal is not noisy at all and we can easily retrieve the transmitted bits. Yet, at the sample 107710, we can find that GPU context switching occurs in the middle of sending a '1' bit, and such events may result in losing bits (if the split does not generate clear signals) or duplicating bits (if clear signals are generated on both sides of the split).

5.4 Mitigation

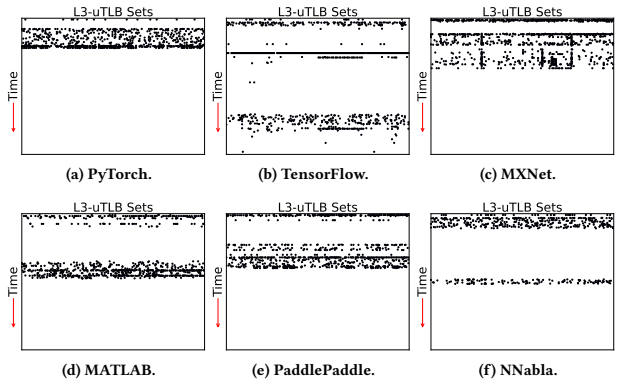
To thwart this data exfiltration attack, the most straightforward and effective approach is to let MIG partition the L3-uTLB as well. However, this requires NVIDIA to modify its hardware implementation, which takes time and effort. Moreover, it cannot be directly used to protect the MIG-supported GPUs in the current clouds.

[7] An L2-uTLB has 2048 entries, and each entry has 16 sub-entries. Each sub-entry can store a 2MB page's address translation. Thus, the 64GB reach is given by 2048×16×2MB. Note that any MIG-created GPU instance has no more than 40GB memory.

Hence, we formulate a software-based mitigation approach that is deployable in practice. The basic idea is to monitor the L3-uTLB victim buffers in a similar way to the receiver and create a large amount of noise through intense evictions when anomalous behavior is detected. As shown by the evaluations, a benign GPU program hardly has the need for accessing the L3-uTLB in its execution, and thus when the monitoring finds that the number of its L3-uTLB misses within a time window is above a threshold, we consider there is an ongoing attack. Subsequently, the defense flushes the victim buffers for some time. Note that this countermeasure needs to be deployed on a GPU instance that is not allocated to any tenant. As multiple GPU instances of different sizes can be created by MIG on a server-grade GPU, we can use the smallest possible instance for this defensive purpose. We have implemented a prototype and verified that it can effectively disrupt the covert channel (i.e., the receiver is made to continually observe `t0` $> T$ and `t1` $> T$) while imposing no performance overhead on other tenants' running GPU programs (e.g., regardless of whether our prototype is deployed, it takes the tenant `C` ~79 minutes to train the model in the previous example). This approach surely has drawbacks. As aforementioned, it needs to occupy a single GPU instance, and the other drawback is higher power consumption due to constant monitoring.

6 BEYOND DATA EXFILTRATION

In addition to the covert channel shown above, there are several other potential attacks exploiting the unpartitioned L3-uTLB. Here, we briefly illustrate a side-channel attack that infers which machine learning (ML) framework is used in another GPU instance. (We also discuss performance degradation possibilities in Appendix G.) Note that, in practice, people tend to directly use ML frameworks provided by the pre-built container images of the GPU cloud, and thus the identification of the ML framework may be leveraged to deduce the underlying image. We report some preliminary results and leave deeper exploration for future work.

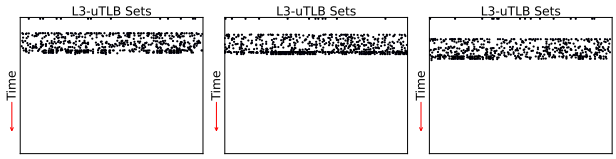


(a) PyTorch. (b) TensorFlow. (c) MXNet.

(d) MATLAB. (e) PaddlePaddle. (f) NNabla.

Figure 13: L3-uTLB access patterns during the first 30 seconds of training a ResNet-50 model on the CIFAR-100 dataset in six different ML frameworks. The Y-axis displays time (top-down order) and X-axis displays L3-uTLB sets.

Although ML frameworks often use the same libraries to operate on GPUs (e.g., cuDNN [30]), they each employ unique strategies for managing models and data within GPU memory. More importantly, we observe that during the startup phases of ML frameworks, they exhibit distinct patterns and timings when loading models and



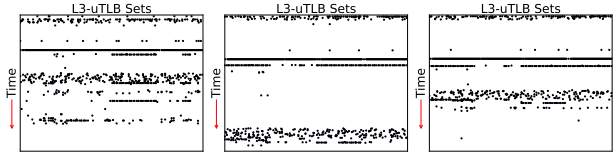
(a) VGG-16 + CIFAR-100. (b) ResNet-50 + SVHN. (c) VGG-16 + SVHN.

Figure 14: L3-uTLB access patterns when using PyTorch to train VGG-16 or ResNet-50 on either CIFAR-100 or SVHN.

data. Consequently, we posit that the early L3-uTLB access patterns can be exploited to reveal the identities of the ML frameworks in use. To validate this hypothesis, we conduct experiments on A30's MIG-created GPU instances, testing six different ML frameworks: PyTorch, TensorFlow, MXNet, MATLAB, PaddlePaddle, and Neural Network Libraries (NNabla). To ensure standardized environments, we retrieve their latest official container images from Docker Hub and NVIDIA NGC Hub.

We first compare the initial L3-uTLB access patterns obtained via Prime+Probe in one GPU instance when using each of the ML frameworks to train a ResNet-50 model on the CIFAR-100 dataset in another GPU instance. Despite training the same model on the same dataset with the same settings, as illustrated in Figure 13, we can easily discern distinct patterns associated with each framework. Notably, we find that these patterns are highly consistent, indicating a strong correlation between the observed L3-uTLB access patterns and the ML frameworks in use. To evaluate this observation, we collect 100 examples for each framework and train a ResNet-18 model. We achieve *100% accuracy* using 5-fold cross-validation.

We further investigate how consistent such patterns are across different models and datasets. To this end, we perform experiments using the VGG-16 model and SVHN dataset apart from the previously tested ResNet-50 and CIFAR-100. Interestingly, we notice that, for some frameworks (e.g., PyTorch, as shown in Figure 14), the impacts of using different models and datasets on the initial L3-uTLB access patterns are not significant, while for others (e.g., TensorFlow, as shown in Figure 15), the impacts are more pronounced. However, even in the latter case where the initial access patterns change to some extent, many key features that help recognize the corresponding frameworks remain. For example, although the patterns in Figure 15 have apparent differences compared to the one shown in Figure 13b, we can still easily notice discernible characteristics that allow us to conclude they are from TensorFlow. We directly use the model trained on data from the ResNet-50+CIFAR-100 case for evaluation. Regardless of whether the test access patterns are obtained when we have VGG-16+CIFAR-100, ResNet-50+SVHN, or VGG-16+SVHN running on the victim GPU instance, we can still *100%* correctly infer the ML framework in use.



(a) VGG-16 + CIFAR-100. (b) ResNet-50 + SVHN. (c) VGG-16 + SVHN.

Figure 15: L3-uTLB access patterns when using TensorFlow to train VGG-16 or ResNet-50 on either CIFAR-100 or SVHN.

7 RELATED WORK

Many efforts have been made to reveal the microarchitectural details of modern GPUs over the past decade [14–16, 23, 26, 42, 49, 52]. These works rely on timing side-channel information to infer the structures of components like TLBs. Even though early studies can roughly find how TLBs are organized in very old pre-Pascal GPUs, as demonstrated in [26], many works have failed to discover an L3 TLB existing in Pascal GPUs. Our work shows that prior studies have also failed to identify the existence of an L3 TLB in post-Pascal GPUs. In addition, our work is the first one that successfully find the existence of L1-iTLB in modern GPUs.

Compared with GPU TLBs, CPU TLBs have already been fully reverse-engineered. In [47], Tatar *et al.* introduce using TLB incoherence to accurately deduce the properties of TLBs in CPU cores. As illustrated in [47], the awareness of the exact TLB properties can help accelerate the TLB-based attacks proposed in [9, 10, 17, 48, 56].

Data exfiltration via microarchitectural covert channels has been studied extensively on the CPU side [8, 11, 12, 22, 46, 53], and lately this problem on the GPU side started drawing attention. Similar to our work, Nayak *et al.* exploit the last-level TLB in Pascal GPUs to construct a covert channel in [26]. While their work represents the first attempt to reverse-engineer the TLB set selection hash functions, we discover that their results may not be fully accurate. Moreover, they consider only the simplest MPS model of sharing but not MIG that is advocated for use in cloud computing. In [24], Naghibijouybari *et al.* use contention on caches, functional units, or memory to construct several covert channels between concurrent kernels. However, all those covert channels proposed in [24] cannot work across GPU instances of MIG, as the resources on which contention is created are completely partitioned. In [3], Ahn *et al.* propose another GPU covert channel that exploits contention on shared interconnect in TPC and GPC. Because each GPC belongs to only one GPU instance as a whole, this covert channel cannot achieve cross-GPU-instance either. In [54], Xu *et al.* propose a countermeasure that detects anomalous contention events and use partitioning to guard against contention-based side/covert attacks on GPUs. Yet, this defense becomes unnecessary as MIG imposes much stronger partitioning.

In [45], Side *et al.* exploit contention on the host-GPU PCIe bus to create a covert channel between VMs using virtualized GPUs. In [7], Dutta *et al.* achieve data exfiltration from shared GPUs connected by NVLink. In [6], Dutta *et al.* also examine how to manipulate the shared resources between the CPU and the integrated GPU such as the last-level cache and the ring bus to construct covert channels. Unlike our work scrutinizing MIG that is specifically designed to prevent resources from being shared for security, these works focus on situations in which no isolation is guaranteed in the first place.

Aside from data exfiltration, in [5], Di *et al.* investigate how to build a defense against potential performance degradation caused by contention on the shared TLB with respect to Pascal GPUs. The defense is based on a software-enforced isolation. However, due to the inaccuracy in the previous reverse-engineering studies, they treated the L2 TLB as the last shared level without knowing the existence of the L3 TLB shared by all the SMs. With the advent of MIG, such a defense becomes unnecessary.

8 CONCLUSION

In this paper, we have formulated an approach to fully reverse-engineering the TLB hierarchies of modern GPUs. Using the approach, we are able to depict a more complete and accurate picture of GPU TLBs, which facilitates us to discover a design flaw of NVIDIA MIG that it does not partition the last-level TLB. Exploiting this vulnerability, we have constructed a very reliable covert channel that enables data exfiltration between MIG-created GPU instances in real clouds. As MIG isolation has proven imperfect, more investigations are certainly needed for checking if any other design vulnerabilities exist.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation (CNS-2147217, CNS-2054657, CNS-2008339, and CNS-1942182). The authors thank the anonymous reviewers for their comments and suggestions that help us improve the quality of the paper.

REFERENCES

[1] [n. d.]. envytools. https://github.com/envytools/envytools.
[2] Andreas Abel and Jan Reineke. 2013. Measurement-Based Modeling of the Cache Replacement Policy. In *RTAS*.
[3] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. 2021. Network-on-Chip Microarchitecture-Based Covert Channel in GPUs. In *MICRO*.
[4] cnvrg.io. [n. d.]. Multi-Instance GPU Support for ML Workloads with cnvrg.io on NVIDIA A100. https://cnvrg.io/solutions/multi-instance-gpu/.
[5] Bang Di, Daokun Hu, Zhen Xie, Jianhua Sun, Hao Chen, Jinkui Ren, and Dong Li. 2021. TLB-Pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling. *ACM TACO* (2021).
[6] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky Buddies: Cross-Component Covert Channels on Integrated CPU-GPU Systems. In *ISCA*.
[7] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2022. Spy in the GPU-box: Covert and Side Channel Attacks on Multi-GPU Systems. *arXiv preprint arXiv:2203.15981* (2022).
[8] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *CCS*.
[9] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.
[10] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
[11] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[12] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. 2022. Leaky Way: A Conflict-Based Cache Covert Channel Bypassing Set Associativity. In *MICRO*.
[13] Mark Harris. 2013. Unified Memory in CUDA 6. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.
[14] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *CoRR* abs/1903.07486 (2019). arXiv:1903.07486 http://arxiv.org/abs/1903.07486
[15] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826
[16] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big Data Causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU. In *DaMoN*.
[17] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *EuroS&P*.
[18] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *S&P*.
[19] Tobias Mann. 2022. Fractional GPUs Empower New Wave Of Accelerated Software Development. https://www.nextplatform.com/2022/06/03/fractional-gpus-empower-new-wave-of-accelerated-software-development/.
[20] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2014. Confidentiality Issues on a GPU in a Virtualized Environment. In *FC*.
[21] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.

[22] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.
[23] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE TPDS* (2017).
[24] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and Characterizing Covert Channels on GPGPUs. In *MICRO*.
[25] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *CCS*.
[26] Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)Managed: A Novel TLB-Based Covert Channel on GPUs. In *ASIA CCS*.
[27] NVIDIA. . CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[28] NVIDIA. . Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.
[29] NVIDIA. . NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.
[30] NVIDIA. . NVIDIA cuDNN Developer Guide. https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.
[31] NVIDIA. . NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf.
[32] NVIDIA. . NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.
[33] NVIDIA. . NVIDIA Virtual GPU Software Documentation. https://docs.nvidia.com/grid/latest/.
[34] NVIDIA. . Pascal MMU Format Changes. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.
[35] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *ISCA*.
[36] Josh Patterson. 2021. A Cloud GPU Value Model for NVIDIA Multi-Instance GPUs (MIG). http://www.pattersonconsultingtn.com/blog/cloud_gpu_value_model_for_nvidia_mig.html.
[37] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *HPCA*.
[38] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *MICRO*.
[39] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*.
[40] Puzl.cloud. [n. d.]. Kubernetes based GPU cloud. https://puzl.cloud/gpu-cloud.
[41] Run:AI. 2020. Run:AI creates first fractional GPU sharing for Kubernetes deep learning workloads. https://www.run.ai/blog/run-ai-creates-first-fractional-gpu-sharing-for-kubernetes-deep-learning-workloads.
[42] Peter Van Sandt and Zhe Jia. 2021. Dissecting the Ampere GPU Architecture through Microbenchmarking. https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/.
[43] Tim C. Schroeder. 2011. Peer-to-Peer & Unified Virtual Addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.
[44] Anton Shilov. 2021. Nvidia Increases Market Share as GPU Sales Explode: JPR. https://www.tomshardware.com/news/jpr-gpu-shipments-in-q1-2021-hit-119-million-units.
[45] Mert Side, Fan Yao, and Zhenkai Zhang. 2022. LockedDown: Exploiting Contention on Host-GPU PCIe Bus for Fun and Profit. In *EuroS&P*.
[46] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. 2018. Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in IaaS Clouds. In *NDSS*.
[47] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *USENIX Security 22*.
[48] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*.
[49] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to Tune Dense Linear Algebra. In *SC*.
[50] Vultr. 2022. Introducing Vultr Talon: Affordable Cloud VMs Accelerated with NVIDIA GPUs. https://www.vultr.com/news/Affordable-Cloud-VMs-Accelerated-with-NVIDIA-GPUs/.
[51] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In *DSN*.
[52] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*.
[53] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security*.
[54] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. 2019. GPUGuard: Mitigating Contention Based Side and Covert Channel Attacks on GPUs. In *ICS*.
[55] Zihao Zhan, Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xenofon Koutsoukos. 2022. Graphics Peeping Unit: Exploiting EM Side-Channel Information of GPUs to Eavesdrop on Your Neighbors. In *S&P*.
[56] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. 2020. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *MICRO*.
[57] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. 2021. Red Alert for Power Leakage: Exploiting Intel RAPL-Induced Side Channels. In *ASIA CCS*.
[58] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. 2017. Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU. *PETS* (2017).

A ISOLATION CLAIMS OF NVIDIA MIG

We list some excerpts from NVIDIA documentations to show the strong isolation claims of MIG. The following excerpt is from [32]:

> *With MIG, each instance's processors have* ***separate and isolated paths through the entire memory system - the on-chip crossbar ports, L2 cache banks, memory controllers, and DRAM address buses are all assigned uniquely to an individual instance*** *...*

The following is an excerpt from A100 GPU's white paper [29]:

> *MIG is* ***especially beneficial for Cloud Service Providers*** *who have multi-tenant use cases, and it ensures one client cannot impact the work or scheduling of other clients, in addition to* ***providing enhanced security*** *and allowing GPU utilization guarantees for customers.*

B GPU PAGE TABLE DETAILS

As shown in Figure 1, a modern GPU supports multiple page sizes and a virtual address has 49 bits divided to select a walking path through a 5-level page table hierarchy. The bits $[48:47]$ of a virtual address are used to index the topmost table of this 5-level hierarchy called page directory 3 (PD3). A PD3 occupies a normal 4KB page, and each PD3 entry's size is 1KB. The tables at the second level are called PD2, each of which occupies a 4KB page as well. The bits $[46:38]$ are used to index a PD2 to retrieve a PD2 entry whose size is 8B. The tables at the third level are called PD1, each of which is also 4KB. The bits $[37:29]$ are used to index a PD1 to retrieve a PD1 entry whose size is 8B. An entry in either PD3, PD2, or PD1 is valid if its last three least significant bits are '010'. The bits $[35:8]$ of a valid entry give the number of 4KB page frame where the table at the next level resides.

The tables to which PD1 entries point are referred to as PD0 and they are indexed by the bits $[28:21]$ of the virtual address. A PD0 resides in a 4KB page and each PD0 entry has a size of 16B. PD0 entries point to either 2MB huge pages or last-level page tables (PT). If the least significant bit of a PD0 entry is '1', this entry points to a 2MB page and its bits $[35:8]$ give the 4KB page frame number to which the 2MB page's first 4KB part is mapped. Otherwise, a PD0 entry serves as a dual pointer with its bits $[35:4]$ pointing to a 64KB page PT (valid if its $[2:0]$ are '010') and its bits $[99:72]$ pointing to a 4KB page PT (valid if its $[66:64]$ are '010'). A 64KB page PT has a size of only 256B and its physical address is derived via left-shifting the value given by the PD0 entry's $[35:4]$ by 4 bits, while a 4KB page PT has a size of 4KB and the value given by the entry's $[99:72]$ is the number of 4KB page frame where this PT resides. A 64KB page PT is indexed by the bits $[20:16]$ of the virtual address, and a 4KB page PT is indexed by the bits $[20:12]$ of the address. An entry in either type of PT has a size of 8B and is valid if its least significant bit is '1'. The bits $[35:8]$ of a valid PT

entry give the number of 4KB page frame to which the 4KB page or the 64KB page's first 4KB part is mapped. Apparently, with respect to 2MB, 64KB, and 4KB pages, the bits $[20:0]$, $[15:0]$, and $[11:0]$ of the virtual address form the page offset, respectively.

C DUMPING GPU MEMORY

To achieve significantly faster GPU memory dumping, we need to take advantage of the DMA mechanism. GPUs are equipped with DMA copy engines for transferring large amounts of data over the PCIe. To operate a copy engine for the purpose of dumping the entire GPU memory, we have added a function to the official NVIDIA GPU driver. Due to the fact that the driver has an internal hardware abstraction layer (HAL) constructed, we can implement our GPU memory dumping function in a very generic way without the need for considering the differences in the copy engines of different microarchitecture generations.

Specifically, for each generation, the NVIDIA GPU driver has an implementation of the `uvm_ce_hal_struct` that has a function named `memcopy()` capable of moving data between the host and GPU memory via DMA. We find that to use the `memcopy()` function for our purpose, we first need to allocate some GPU memory appropriately using the `uvm_mem_alloc_vidmem()` function and then map it using the `uvm_mem_map_gpu_kernel()` function. After this operation, we are allowed to use the `memcopy()` function to access arbitrary GPU physical address range without being limited to the one allocated and mapped before.

The UVM kernel module creates a device file `/dev/nvidia-uvm`. We rely on the `ioctl()` system call to pass the dumping command and a physical address range to the modified driver through this file. The contents sent back to the host memory via DMA will be written into a file.

D UVM-MANAGED PAGES

A module in the NVIDIA driver is in charge of allocating UVM-managed pages. The `cudaMallocManaged()` function registers a virtual address subspace for UVM use. The UVM module allocates pages when the GPU accesses addresses in the registered address space. Although three page sizes are supported (see Figure 1), we find that UVM actually only allocates 64KB and 2MB pages.

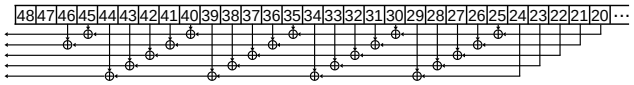
UVM starts with allocating 64KB pages, but it will merge the 64KB pages within a 2MB page into the 2MB page if the residency reaches certain conditions. For example, if the first 17 or more 64KB pages in a 2MB page are present on GPU, the page table entries for these 64KB pages will be purged and replaced with a 2MB entry; but if just the first 16 64KB pages are used, the merging operation will not be triggered. We find that some other residency patterns with less than 17 pages can also trigger the merging. For instance, if every other 64KB page is used (i.e., 16 ones as there are 32 64KB pages in a 2MB page), the merging will also happen.

E MORE REVERSE-ENGINEERING DETAILS

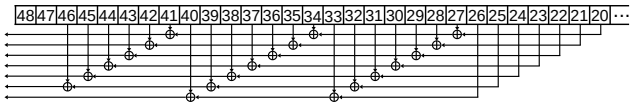
To check if the TLB hierarchy is fully filled due to accessing a set of pages, we modify the address mappings for the first a few pages in the set. The newly mapped physical page frames have a predefined value. If the predefined value is read out, it means that the stale translations for these pages are evicted and new ones are loaded by the page table walker.

As GPU code and data pages are not distinguishable from the perspective of their page table entries, we are allowed to directly write anything on a code page as long as we know the corresponding virtual address. Yet, unlike those data pages allocated by CUDA APIs whose virtual addresses are given, there is no straightforward way to learn the virtual address of a kernel function used in the GPU context. We have formulated a solution to the problem that is to identify the code page frame holding the kernel function in the dumped GPU memory and deduce the corresponding virtual address from the extracted page table. After finding the virtual address A_0 of the branch instruction, we can make A_1 be its target by writing its offset part with $A_1 - (A_0 + 16)$.

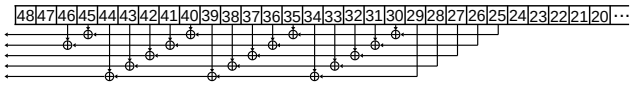
F SET SELECTION FUNCTIONS OF TURING



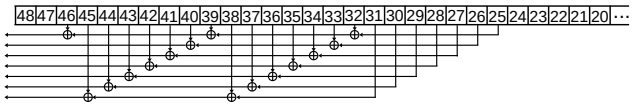
(a) XOR-5^{46}_{20} is used for indexing L2-uTLB when using 64KB pages.



(b) XOR-7^{46}_{20} is used for indexing L3-uTLB when using 64KB pages.



(c) XOR-5^{46}_{25} is used for indexing L2-uTLB when using 2MB pages.



(d) XOR-7^{46}_{25} is used for indexing L3-uTLB when using 2MB pages.

Figure 16: TLB set selection hash functions used by Turing GPUs.

G PERFORMANCE DEGRADATION

Given the large L2-uTLB reach, performance degradation attacks may not be easy to achieve under normal circumstances. Yet, if 64KB pages are primarily used, it can trigger numerous L3-uTLB accesses, and attacker on another GPU instance can repeatedly evict the L3-uTLB to conduct performance degradation attacks. A common case where 64KB pages are utilized without merging is when GPUs are used for graphics rendering. However, graphics rendering APIs (e.g., Vulkan and OpenGL) are currently not supported in MIG-created instances (but supported in vGPU-created instances). Even so, there are still cases where 64KB pages remain unmerged in MIG-created GPU instances, and a simple one is when a program sparsely uses UVM address space on the GPU side. For example, we create an array of 65536 objects in UVM on the CPU side, but only reference 3700 of them on the GPU side, with the addresses of each referenced object pair separated by at least `0x100000`. When we traverse these objects 2000 times in the absence of a malicious program sweeping the L3-uTLB, it takes 6.34 seconds. However, in the presence of such continuous sweeping, it takes 8.11 seconds, which is 28% slower than before. This slowdown is very consistent when the objects are traversed for a varying number of iterations.