(Mis)managed: A Novel TLB-based Covert Channel on GPUs

Ajay Nayak
Indian Institute of Science
Bangalore, India
ajaynayak@iisc.ac.in

Pratheek B.
Indian Institute of Science
Bangalore, India
pratheekb@iisc.ac.in

Vinod Ganapathy
Indian Institute of Science
Bangalore, India
vg@iisc.ac.in

Arkaprava Basu
Indian Institute of Science
Bangalore, India
arkapravab@iisc.ac.in

ABSTRACT

GPUs are now commonly available in most modern computing platforms. They are increasingly being adopted in cloud platforms and data centers due to their immense computing capability. In response to this growth in usage, manufacturers continuously try to improve GPU hardware by adding new features. However, this increase in usage and the addition of utility-improving features can create new, unexpected attack channels. In this paper, we show that two such features—unified virtual memory (UVM) and multi-process service (MPS)—primarily introduced to improve the programmability and efficiency of GPU kernels have an unexpected consequence—that of creating a novel covert-timing channel via the GPU's translation lookaside buffer (TLB) hierarchy. To enable this covert channel, we first perform experiments to understand the characteristics of TLBs present on a GPU. The use of UVM allows fine-grained management of translations, and helps us discover several idiosyncrasies of the TLB hierarchy, such as three-levels of TLB, coalesced entries. We use this newly-acquired understanding to demonstrate a novel covert channel via the shared TLB. We then leverage MPS to increase the bandwidth of this channel by 40×. Finally, we demonstrate the channel's utility by leaking data from a GPU-accelerated database application.

CCS CONCEPTS

• **Security and privacy** → **Hardware attacks and countermeasures**; **Hardware reverse engineering**; • **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

GPU; TLB; reverse-engineering; covert-timing channel; unified memory; multi-process service

ACM Reference Format:
Ajay Nayak, Pratheek B., Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)managed: A Novel TLB-based Covert Channel on GPUs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3433210.3453077

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8287-8/21/06…$15.00
https://doi.org/10.1145/3433210.3453077

1 INTRODUCTION

As GPUs continue to find favor among an increasing number of application developers, most major public cloud providers have added GPUs to their infrastructure [1, 7, 37]. On cloud platforms, resource consolidation through concurrent sharing of a single resource (*e.g.,* GPU) across multiple clients (tenants of the cloud) is of paramount importance. To cater to this emerging need, GPU vendors like Nvidia have enhanced multi-tenancy support through features like Multi-Process Service (MPS) [36]. MPS allows GPU kernels launched from different processes to execute concurrently on a GPU. Another recent feature, such as Unified Virtual Memory (UVM) [35], presents a unified view of CPU and GPU memory, thereby easing the development of GPU kernels. UVM also relieves the developers from manually managing the GPU's capacity-constrained on-board memory, thereby easing GPU programming.

These new features (*e.g.,* UVM, MPS) are undoubtedly helpful to ease programming and improve the utilization of GPUs. However, in this paper, we show that they also create a novel threat vector. We demonstrate that concurrent execution of kernels from different processes on a GPU can help create a covert timing channel via the GPU's translation lookaside buffer (TLB) hierarchy. UVM aids in the creation of such channels by allowing fine-grained inspection of microarchitectural details of a GPU's hardware resources.

There is much recent excitement on using microarchitectural features to enable timing attacks (both on CPUs, *e.g.,* [4, 5, 22, 40, 42] and GPUs [14, 15, 29]). There is also an active body of research on developing defenses against these timing channels (*e.g.,* [20, 46, 53, 55, 58]). This paper focuses on a relatively unexplored microarchitectural channel, *i.e.,* the TLB hierarchy. Although there has been prior work on TLB-based timing channels in CPU TLBs [8], our work focuses on the GPU TLB hierarchy. Unlike in CPUs, where the TLB hierarchy is private to a core, we discovered (crucially, via UVM) that the last-level TLB is shared in GPU. This enables broader TLB-based attacks on GPUs. Thus, the GPU TLB timing channel is a novel contribution and demonstrates that attackers can leverage the channel to leak secrets from GPU-accelerated applications.

The main challenge in devising the channel is the lack of publicly-available documentation of the TLB hierarchy on commercial GPUs. For example, in any hardware-based timing channel, a `Trojan` and a `Spy` need a shared hardware structure with known access and timing characteristics to communicate bits of information. Therefore, we need to identify which level in the TLB hierarchy is shared (if any), its structure (*e.g.,* size, associativity, indexing function), and the typical latencies of a TLB hit/miss.

Using UVM to discover a shared TLB level. We reverse-engineer the details of the TLB hierarchy of a commercially available GPU, the Nvidia 1080Ti (Pascal microarchitecture). A contribution of this paper is the use of UVM to discover previously unreported TLB details. UVM allows us to allocate a large, sparsely-allocated virtual

address space that exceeds the GPU's on-board memory capacity. UVM also ensures the use of relatively smaller page sizes (*i.e.,* 64KB vs. 2MB without UVM). As a result, we discover the existence of a shared L3 TLB, which has not been reported in prior work on this microarchitecture. This L3 TLB is key to our covert channel since it is the *only* TLB level shared across the GPU. We also reverse-engineer the details of all TLB levels, such as the number of entries and associativity, via careful microbenchmarking.

For the `Trojan` and the `Spy` to communicate by accessing and evicting entries from the shared TLB, it is crucial to develop a detailed understanding of the function used to index entries to the TLB. We devise an approach to accomplish this task by programmatically observing *eviction sets*—a group of virtual addresses that index to the same set of a TLB—via an intricate pointer-chasing microbenchmark. We were able to construct the indexing function via a careful combinatorial analysis of these eviction sets. We also discover evidence of dynamic coalescing of 16 contiguous virtual address pages into a single entry in both L2 and L3 TLBs. While AMD's CPUs employ such tricks in their TLBs [2], we are the first to report similar techniques being adopted in GPU TLBs publicly.

After gaining sufficient insight into the TLB hierarchy of the GPU, we create a covert-timing channel via the shared L3 TLB. The channel uses the popular prime+probe technique [31, 40, 42]. However, the key challenge is to identify a set of virtual addresses for the `Trojan` and `Spy` to access such that the addresses overflow the L1 TLB and (parts of) the L2 TLBs to the L3 TLB. Otherwise, the `Spy` will fail to monitor the activities of the `Trojan` in the L3 TLB. We leverage the insights of the structure of each TLB to carefully create minimal sets of virtual addresses that overflow to the shared L3 TLB. The `Trojan` then uses each such set of virtual addresses to transmit one bit of the secret to the `Spy` via the timing channel.

Using MPS to improve channel efficiency. In practice, we found the above channel to be functional, but with a high bit-error rate. In particular, the `Trojan` and the `Spy` interleave via context-switches to communicate, and we believe that these context-switches contribute to the large error rate, resulting in a noisy channel. We found that the channel can be improved using MPS. Since MPS enables concurrent execution of kernels from different processes within a GPU, a context-switch between the `Trojan` and `Spy` is avoidable. Consequently, the bit-error rate reduces with MPS enabled. Furthermore, MPS also avoids the latency of the intervening context-switch. This leads to a 40× increase in the bandwidth of the GPU's L3 TLB-based covert channel to 81Kbps.

We demonstrate the channel's utility by leaking data from a real-world application. Specifically, we modify a GPU-accelerated database library to leak rows of data using our channel while inserting rows into the database.

In summary, we make the following contributions.

- We discover the existence of a shared L3 TLB in Nvidia's Pascal microarchitecture-based GPU, using UVM.
- We reverse-engineer the indexing function for the TLBs via careful analysis with eviction sets.
- We show evidence of coalesced entries [43] in L2 and L3 TLBs.
- We use the shared L3 TLB to enable a novel covert timing channel on the GPU, improve the bandwidth by 40× using MPS and employ the channel to leak data from a GPU-accelerated application.

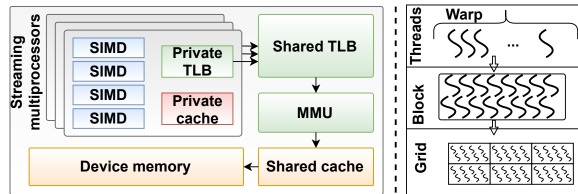


Figure 1: GPU architecture and programming model.

2 BACKGROUND

2.1 GPU Architecture

GPUs are massively data-parallel processors that employ thousands of concurrent threads of execution to provide high throughput. To keep this massive parallelism tractable, a GPU's hardware and software follow a hierarchical model.

Figure 1 depicts a typical GPU architecture on the left and a typical programming hierarchy on the right. The basic computing blocks of a GPU are streaming multiprocessors (SMs). A state-of-the-art GPU may contain up to 64 SMs. Each SM contains multiple single-instruction multiple-data (SIMD) units, and each SIMD has several *lanes* of execution (typically 16–32). A SIMD unit executes a single instruction over (possibly) different data points across all lanes in parallel. Each SM has a private L1 cache, and a scratchpad shared across SIMD units of that SM. All SMs share a larger L2 cache that is connected to the on-board memory. The GPU's on-board memory supports large bandwidth to service the needs of thousands of concurrent threads.

Like CPUs, GPUs also use TLBs to keep recently-used virtual-to-physical address translations. The total amount of virtual address mappable via a TLB is called the *reach* of that TLB. The reach can be calculated by multiplying the number of TLB entries with the size of virtual memory mapped by each entry. Modern GPUs support a hierarchical TLB structure. While the exact details of the TLB hierarchy are not publicly disclosed, some levels in the hierarchical TLB structure are private to an SM, while others are shared across SMs [19, 48]. Indeed, one of the core contributions of this paper is to reverse-engineer the TLB hierarchy of a modern GPU.

GPU programming languages, such as OpenCL and CUDA, expose a hierarchy of execution groups to the programmer that follows the hierarchy in the hardware (right side of Figure 1). We here focus on CUDA, used to program Nvidia's GPUs [33]. A CUDA program consists of portions that must execute on the CPU (Figure 2) and others that must execute on the GPU (Figure 3). A GPU code, called a *kernel*, is invoked with CUDA-specific syntax from the CPU code. In CUDA parlance, a thread is the smallest execution entity that runs on a single SIMD unit lane. A group of threads forms a warp, which is the smallest hardware-scheduled unit of work that executes in single-instruction multiple-thread (SIMT) fashion. Thread block, which is programmer-visible, is made up of several warps. All threads in a thread block are scheduled on the same SM. Finally, work on a GPU is dispatched at the granularity of a grid, which comprises of several thread blocks.

We now discuss two relatively new features in Nvidia GPUs that play a crucial role in this work.

• **Unified Virtual Memory (UVM) [35].** Not long ago, GPUs lacked the ability to share the virtual address space with a CPU process. It made writing GPU programs, especially those with pointers and shared data structures with CPU process, harder. Pointers on the CPU were invalid on the GPU and vice-versa. Programmers had to explicitly allocate memory on the GPU and copy data from the CPU to GPU's memory. This paradigm also limited the maximum allocatable memory in a GPU kernel to a relatively small on-board memory capacity in modern GPUs (*e.g.,* 11GB on Nvidia 1080Ti).

The UVM feature in Nvidia's GPUs addresses this shortcoming. Memory allocated using the UVM API (*e.g.,* `cudaMallocManaged`) is accessible on both the CPU and the GPU, with no additional programming effort. The UVM driver in the OS is responsible for migrating pages *transparently* across the CPU and the GPU. The UVM driver manages page tables on the GPU and the one on the CPU to provide an illusion of shared virtual address space and shared physical memory. A key additional benefit of UVM is that it allows allocating memory larger than GPU's capacity-constrained on-board memory. In contrast, the size of memory allocated using CUDA's default API, `cudaMalloc`, is limited to the GPU's on-board memory size. Furthermore, we discovered that the memory allocated using `cudaMallocManaged` is mapped using relatively smaller page sizes than the memory allocated using `cudaMalloc`.

• **Multi-Process Service (MPS) [36].** Earlier, a GPU could concurrently run kernels launched from the same application process only, via CUDA streams. However, two recent trends are necessitating the ability to execute kernels from independent applications concurrently. First, the GPUs have made their way into public cloud infrastructures [1, 7], where resource consolidation via concurrent execution of kernels from independent tenants (application) is important. Second, the number of SMs and the memory bandwidth continue to grow in modern GPUs. It may not always be possible to keep the entire GPU busy with kernels from a single application.

Nvidia thus introduced MPS to allow kernels from different processes to execute concurrently on a single GPU, spatially sharing GPU resources. The MPS's capabilities continue to flourish over generations of Nvidia GPUs. Until the Volta microarchitecture, MPS was largely a software-based solution. A daemon called the MPS-server would merge address spaces of CUDA contexts from multiple processes into a single address space. Volta onward, Nvidia enhanced hardware support to enable address space isolation across co-resident CUDA contexts from multiple applications.

2.2 TLB-based Timing Channel on CPUs

TLBs have been used as a vector in timing-channel attacks in the past. The attacks (relying on hit/miss information) demonstrated so far in the literature primarily apply on the CPU side. TLBleed [8] uses prime+probe and machine learning techniques to extract secrets (*e.g.,* key for RSA algorithm) from a process running on the same processor core. On CPUs, TLB is shared among hyper-threads, *i.e.,* processes running on the same physical core simultaneously. Thus, a cross-core channel using a TLB is impossible on CPUs. On GPUs, all the SMs share the last-level TLB. However, recent studies have demonstrated that overflowing the shared TLB is not possible within the GPU memory limits [12, 13, 19]. We challenge the memory limit notion in this work and leverage the hardware advancements meant for programmability, *i.e.,* UVM, to overflow the shared level within memory constraints. The shared nature provides a broader attack surface when kernels with malicious intent run on the GPU in parallel (using the last-level TLB to communicate covertly). Note that the host processes corresponding to these kernels can run on any CPU core, effectively making this channel a cross-core channel from a CPU's perspective.

3 ATTACK GOAL AND THREAT MODEL

Our goal is to demonstrate a TLB-based covert timing channel on a modern GPU. In a covert channel, two entities, a `Trojan` (sender) and a `Spy` (receiver), collaborate to pass information covertly. The `Trojan` and the `Spy` are GPU kernels launched by independent processes that do not otherwise have a direct communication channel (*e.g.,* via IPC). We assume that the `Trojan` is a GPU kernel with access to some secret data (*e.g.,* a database, inputs to a classifier) that it is not allowed to share with the `Spy`.

The covert channel leverages the shared last-level TLB and uses prime+probe [23, 31, 40] to communicate information covertly. The `Spy` *primes* the TLB by accessing memory locations and filling up entries in the TLB. The `Trojan` subsequently runs, perhaps accessing confidential data, and making control-flow decisions based on the value of that data. The `Spy` subsequently reruns the priming code while timing the accesses to the memory locations. Because the `Trojan`'s accesses evict certain TLB entries, some of `Spy`'s accesses take longer, thereby setting up a timing channel. The `Spy` uses this to determine the control-flow decision that the `Trojan` must have taken and thereby infer the secret.

The key novelty in our work is the use of the shared last-level (L3) TLB to enable the covert channel. While prior work has developed cache-based side and covert channel attacks on both CPUs [4, 5, 23, 59, 60] and GPUs [14, 15, 29], ours is the first to observe that the shared last-level TLB on GPUs can be a viable covert channel. A key challenge in our work is that the hierarchy of the TLB on modern GPUs is not publicly known. For example, details such as the structure of the TLB hierarchy, set-associative structures, and indexing function are not documented anywhere. Therefore, one of the contributions of this paper is the set of methods that we developed to reverse-engineer these details.

In our threat model, we consider two colluding GPU kernels, a `Trojan` and a `Spy`, that covertly exchange confidential data to which the `Trojan` has access. We assume that both the `Trojan` and the `Spy` run on a modern GPU having access to all the runtime features such as UVM. We assume that MPS is enabled on the platform as it improves GPU utilization. The GPU is connected to a CPU via a PCI-express card, and an unmodified device driver hosted on the CPU controls it. The CPU and GPU hardware, as well as the OS, are assumed to be trusted. The `Trojan` and the `Spy` kernels are launched on the GPU from unprivileged user-space processes.

In this paper, we demonstrate the feasibility of a covert channel on a single physical machine with an attached GPU. This covert channel can also be applied to jobs launched on the cloud. However, this would require both the `Trojan` and the `Spy` to be co-located on the same physical machine and the kernels to be launched on the same GPU. The problem of co-locating jobs on cloud platforms is orthogonal and has been studied in the past for CPUs [50, 51]

```
1   #define REPEAT 1
2   #define STEP   8
3   void generate (int min_size, int max_size, int stride) {
4     int *arr = cudaMallocManaged (REPEAT * max_size + 2);
5     for (j = 0; j < REPEAT; j++) {
6       start_idx = j * STEP * stride;
7       for (size = min_size; size <  max_size, size += stride) {
8         end_idx = start_idx + size;
9         /* Generate pointer-chase pattern */
10        for (i = start_idx; i < end_idx; i++) {
11          next_idx = i + stride;
12          arr[i] = next_idx > end_idx ? start_idx : next_idx;
13        }
14        p_chase<<<1, 1>>>(arr);    /* Launch Kernel */
15      }
16    }
17  }
```

Figure 2: Generating cyclic pointer-chase pattern.

```
1   #define PROBES 1024
2   __global__  void p_chase (int *arr) {
3     int j = 0, old_j, sum = 0, iter = 3;
4     /* Warmup the TLBs and caches before timing it */
5     /* Now track access time for each access */
6     for (i = 0; i < PROBES * iter; i++) {
7       old_j = j;
8       start = clock ();
9       j = arr[j];
10      /* A dependent instruction */
11      sum += j;
12      /* Store access time at immediate next index */
13      arr[old_j + 1] = clock () - start;
14    }
15    /* Make sure "sum" is not eliminated by compiler */
16    arr[2] = sum;
17  }
```

Figure 3: Pointer-chasing kernel that runs on a GPU.

and GPUs [29, 30]. Even in environments not pertaining to cloud settings, applications running on a single machine share the GPU. Since our goal is to establish the feasibility of this novel TLB-based covert channel on GPUs, we assume that the attacker has managed to co-locate the kernels on the same machine.

4 REVERSE ENGINEERING GPU'S TLB CONFIGURATION

The key requirement for creating a covert channel is to ensure that the `Trojan` and the `Spy` can communicate via timing measurements of a shared hardware structure. In this work, we focus on the GPU's TLB subsystem. Thus, we need to first understand the microarchitectural details of the GPU's TLB hierarchy to ensure that the `Trojan` and the `Spy` can communicate in a predictable manner. Unfortunately, unlike CPUs, details of the TLB hierarchy of commercial GPUs are not available publicly. Further, even (public) performance counters for TLB events are absent on GPUs such as Nvidia's Pascal that we use as the experimental platform.

Consequently, we need to reverse-engineer microarchitectural details of the GPU's TLB hierarchy. Specifically, we seek to answer the following questions:

- How many levels of TLB are present, and how big are they?
- Which levels of TLBs are private and which are shared?
- What are the indexing function for each level of TLB and the page sizes?

The answers to the first two questions are necessary to decide which TLB level should be used as the shared hardware structure for establishing the covert channel. The answer to the third question is necessary to ensure that `Trojan` and `Spy` can deterministically induce timing variations to communicate.

While we discuss our approach and results with respect to Nvidia's Pascal GPU microarchitecture, we believe a similar approach can be used to decipher the TLB details of any GPU. We make some standard assumptions in our approach: ① the replacement policy is LRU (or some approximation thereof, *e.g.,* Tree-LRU), ② the indexing function is static, *i.e.,* it does not vary across kernel runs.

At the core of our reverse-engineering effort is the pointer-chasing algorithm (Figure 2) that accurately measures the time for repeatedly looping over an array. The algorithm is called *pointer-chasing* since each array element stores the index of the next element in the array (*a.k.a.,* pointer) to be accessed. Since access to a given element needs to complete before the subsequent access can be initiated (data dependence), this access pattern ensures that hardware cannot start an access before the previous access completes. This enables accurate latency measurement for each access.

By varying the size of the array and the stride of each access and measuring the corresponding difference in access latencies, characteristics of caching structures such as TLB are inferred. Such pointer-chasing algorithms are commonly used in reverse-engineering data caches in CPUs and GPUs [47, 56]. Our core contribution here is to reverse-engineer new details of the GPU's TLB hierarchy that have never previously been reported publicly. We do so via careful use of the above-mentioned pointer-chase algorithm and extending it when required (summarized later).

A challenge in studying TLB is that latency measurements can get noisy due to hits/misses in the data cache for the accessed array element. This noise would lead to erroneous inferences about the TLB hierarchy. We address this challenge in two parts. First, we pass a flag to the compiler while compiling the pointer-chasing code to disable L1 data caching. Second, we ensured that accessed data fits inside the L2 data cache in all our experiments, always resulting in hits. Therefore, any observed variations in timing can be attributed to TLB behavior. This was necessary since, unlike the L1 cache, there is no publicly-available way to disable the L2 cache.

4.1 Levels of TLB and Their Reach

We set out to infer the number of levels of TLB and their respective reach. Previous studies have attempted the same before [12, 13, 19, 28, 56]. However, our use of UVM allowed us to discover many nuances that were not previously reported on Nvidia's Pascal GPUs.

Figure 4 shows the plot of how the average access latency changes with increasing array sizes. We observe *three* distinct steps in the plot (the smallest step is zoomed in). These steps signify that there are three levels of TLB in the GPU. Until the array's size is within the TLB reach at a given level, there is no increase in the average access latency since all the accesses will be hits in the TLB level. When the array size increases beyond the given TLB's reach, there will be misses at that level, increasing the average access time. The latency will stay at the elevated level as long as the array size is within reach of the next level of the TLB (say L2 TLB). Thus, there will be a plateau in the access latency until the array size increases beyond even the reach of that TLB level (here, L2). To the best of

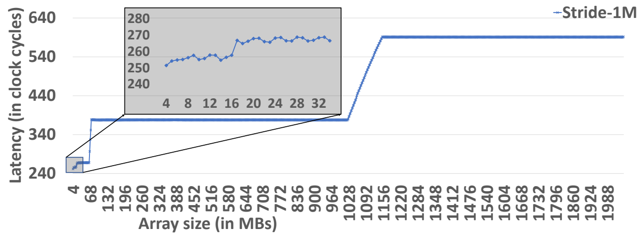


Figure 4: Observing multiple TLB levels.

our knowledge, we are the first to publicly identify that Nvidia's Pascal GPUs have three levels of TLBs.

Next, we infer the reach of each TLB level and the size of virtual memory translated by each TLB entry. We explore each level separately since the characteristics of each level could be different. We thus focus on the three steps in the latency plot of Figure 4 to decipher the details of each level (Figures 5a, 5b, and 5c).

We need to vary both the `size` of the array and the `stride` in the pointer-chase algorithm to infer reach. When the stride is smaller than or equal to the amount of memory that a TLB entry maps, we expect to observe a noticeable increase in the average access latency as soon as the array size overshoots the TLB's reach (*i.e.,* the start of a "knee"). This observation is expected because, until that point, we expect hits in the given level of TLB but capacity misses afterward. When the stride is twice the size of memory mapped by a TLB entry, the start of a knee will shift right on the x-axis (*i.e.,* array size) equal to the TLB reach. To overflow a given level of TLB, the number of entries to be accessed remain unchanged. However, as the stride is now twice of what a TLB entry maps, we need to double the array size to observe the knee in the latency plot.

Therefore, we look for patterns in the latency graph for each level where we observe the knee up to a given stride, and the knee shifts right in the next stride (twice that of the previous). The array size where the first knee appears and the shift on the x-axis between the first and the second knee denotes the TLB reach at the given level. The largest stride at which the first knee starts indicates the amount of memory mapped by a TLB entry.

In Figure 5a, the first knee starts around 1MB array size (x-axis) and the second knee around 2MB. Thus, the TLB reach for L1 TLB is 1MB. The second knee is observed with 128KB stride, while the first knee at 64KB stride. Thus, each entry in L1 TLB maps 64KB. Similarly, from Figure 5b and Figure 5c, we observe that the knee starts at 65MB and 1025MB, respectively (difficult to visually distinguish between 64MB and 65MB or between 1024MB and 1025MB in the plot, we verified it in the raw data). Thus, the L2 and the L3 TLB reach is 65MB and 1025MB, respectively. Further, individual TLB entries in L2 and L3 map 1MB of memory.

4.2 Indexing Function and Associativity

Having established that there are three levels in the TLB hierarchy, we now understand each TLB structure. Specifically, we answer two questions: ① what is the number of sets and number of entries per set? ② what is the function used to index virtual page numbers to TLB sets? The answers to both these questions go hand-in-hand because the indexing function must ensure that a virtual page number (VPN) is mapped uniquely to a set.

First, we note a few observations before answering our questions. Once the array size overflows the TLB reach for L2 and L3, the access latency increases in several steps with increasing array size before plateauing out again in the latency plots (Figures 5b and 5c). A set-associative structure observes such increase in several steps; after the TLB reach is overwhelmed, sets overflow one by one with increasing array size. Consequently, the number of misses increases in the same proportion until all sets overflow. The number of steps indicates the number of sets present in the structure [56]. In the latency plot of L1 TLB (Figure 5a), we observe 1 step, unlike in the plots for L2 (7 steps) and L3 (1023 steps). Therefore, we infer that L1 TLB is a fully-associative structure, while *the L2 and L3 TLBs are set-associative structures.* In the rest of this section, we present our analysis for the L2 TLB and elide similar details for the L3 TLB.

Next, we use the idea of *eviction sets* [52] to answer our questions, captured during each run of the pointer-chasing algorithm (Figure 3). Here, an eviction set is a group of virtual addresses that index to the same set of a TLB. Accessing a new address, which subsequently increases the access-time of a previously accessed address, suggests that the entry for the new address evicted the entry corresponding to the previously accessed address. Thus, the two addresses belong to the same eviction set. Note that the number of previously-accessed addresses that were evicted suggests the set's associativity (a consequence of a conflict/capacity miss).

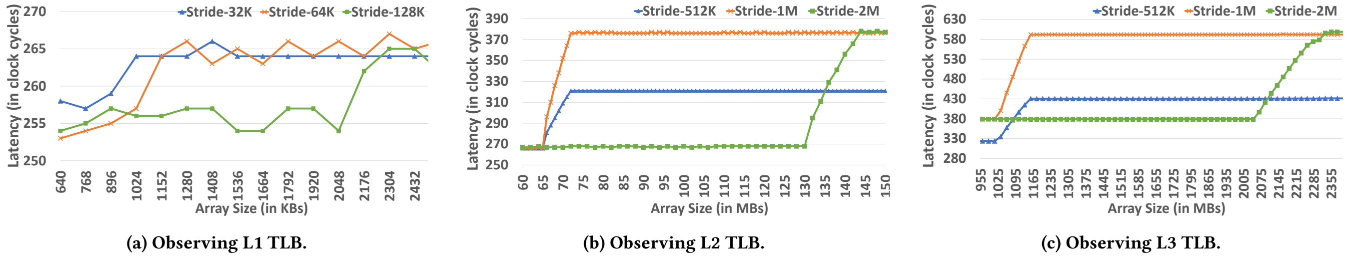
We observe eviction sets with uneven sizes; 1 set of size 17 and 6 sets of size 8, totaling 65 entries. This observation is consistent with previous studies [19, 28]. However, we believe that an odd number of sets (7 here, inferred from the number of steps) and unevenly-sized sets are unlikely. We instead hypothesize that *a victim cache* [32] *creates the illusion of an eviction set of size* 17 *that stores most recently-evicted TLB entries.*

We probe this hypothesis by running an *extended pointer-chasing* experiment. In this experiment, we first allocate a large virtual address space and run the pointer-chasing algorithm multiple times, albeit at different starting addresses in each iteration. We *merge* the eviction sets generated in each iteration based on common addresses present across them. While merging, we do not consider eviction sets of size 17 because we believe these are an illusion created by a victim cache. The number of eviction sets at the end of this iterative merging must indicate the number of sets in the TLB. We leave more details of the algorithm in Appendix A.

Our results showed eight such "merged" eviction sets, hinting that the L2 TLB consists of 8 sets. Since the TLB contains 65 entries, this suggests an *8-way set-associative structure with a single victim-entry* storing most-recently evicted TLB entry of any set.

We can further support this observation once we find the function that indexes VPNs to TLB sets. Our observations from Section 4.1 and Figure 5b help us eliminate the fact that L2 uses mod-based/linear indexing function. In mod-based indexing, if the `stride` value in Figure 3 is increased to two-times the amount of memory mapped by a TLB entry, the perceived coverage of the TLB must not change, contradictory to our observations.

We consider an indexing function that utilizes few bits in the VPN to index into the TLB. A recent study by Gras *et al.* [8] on TLBs of Intel CPUs suggested that the hardware combines a small number



(a) Observing L1 TLB. **(b) Observing L2 TLB.** **(c) Observing L3 TLB.**

Figure 5: Deciphering each level of the TLB hierarchy separately by varying `stride` and `size` parameters in the pointer-chase algorithm.

$$o_2 = i_{22} \oplus i_{25} \oplus i_{28} \oplus i_{31} \oplus i_{34} \oplus i_{37} \oplus i_{40}$$
$$o_1 = i_{21} \oplus i_{24} \oplus i_{27} \oplus i_{30} \oplus i_{33} \oplus i_{36} \oplus i_{39}$$
$$o_0 = i_{20} \oplus i_{23} \oplus i_{26} \oplus i_{29} \oplus i_{32} \oplus i_{35} \oplus i_{38}$$

Figure 6: The function used to index into the L2 TLB.

$$o_6 = i_{26} \oplus i_{33} \oplus i_{40} \qquad o_5 = i_{25} \oplus i_{32} \oplus i_{39}$$
$$o_4 = i_{24} \oplus i_{31} \oplus i_{38} \qquad o_3 = i_{23} \oplus i_{30} \oplus i_{37}$$
$$o_2 = i_{22} \oplus i_{29} \oplus i_{36} \qquad o_1 = i_{21} \oplus i_{28} \oplus i_{35}$$
$$o_0 = i_{20} \oplus i_{27} \oplus i_{34}$$

Figure 7: The function used to index into the L3 TLB.

of bits using $\oplus$ (exclusive or) for indexing purposes. Other studies have also suggested the presence of $\oplus$-based indexing functions for different hardware structures [11, 22, 27]. We assume that the indexing function used by our platform does the same.

To determine the indexing function, we appeal to the "merged" eviction sets formed earlier in the section. Note that all the addresses in a single eviction set index into the same TLB set. We obtain each bit of the TLB set by combining a few address bits using $\oplus$. We study the values at various bit positions in a group-wise comparison of eviction sets. By carefully selecting which eviction sets are grouped and studying the values at various bit positions, we determine which VPN bits decide set selection. We reconstruct the indexing function using this information. More details about the methodology can be found in Appendix B. The function in Figure 6 cleanly indexes the eviction sets into 8 sets (the set number is $o_2o_1o_0$, and i_n is the nth bit of the virtual address). We name this function `XOR-3`.

We now verify our hypothesis of a victim entry using the indexing function in Figure 6. We reran the pointer-chasing benchmark, using the indexing function to access exactly eight addresses that index to each of the 8 TLB sets and one additional address. If our hypothesis about the victim entry is correct, the access to this additional page must also appear to be an L2 TLB hit. Indeed, this is what we observed. If we increase the number of additional addresses accessed in other sets, then the single victim entry will no longer be sufficient. We conducted this experiment and observed that the accesses then take longer, indicating a miss in the L2 TLB.

We repeated the same exercise for the L3 TLB as well. Using a similar analysis, we found that the L3 TLB is an *8-way set-associative structure with 128 sets and* 1 *victim entry*. The indexing function that we inferred for the L3 TLB (called `XOR-7`) is shown in Figure 7.

4.3 Page size versus TLB entry size

In Section 4.1, we observed that each entry of the L1 TLB maps 64KB of contiguous virtual address space, while an entry in the L2 and the L3 maps 1MB. However, Nvidia's documentation [38] lists 4KB, 64KB and 2MB as the supported page sizes—1MB is *not* on the list. In this section, we investigate why each entry in the L2 and L3 TLB appears to map a 1MB virtual address region.

Understanding the 1MB observation will help construct eviction sets for the covert channel by defining the least gap between the entries in them. We hypothesize two possibilities while discarding the existence of a 1MB page size: ① the L2 and L3 TLB in Nvidia's GPUs deploy *coalesced large-reach TLBs (CoLT)* [43]. In CoLT, a single TLB entry can map multiple contiguous pages, as long as they map to a contiguous physical address range. For example, a single TLB entry can map 16 contiguous 64KB VPNs when the hardware detects that they map to contiguous physical page frames; ② a static prefetcher that always loads *next n* translations on a TLB miss (here, $n = 16$), as speculated in a previous study [19].

We design an experiment by again modifying the pointer-chase algorithm of Figure 2 to distinguish these two possibilities. We observe that the key difference between CoLT and static prefetching is that in CoLT, a single TLB entry can map 16 VPNs, unlike in static prefetching. Thus, the modified algorithm's main idea is to cyclically access some 64KB VPNs that are more than the number of TLB entries, as conceptually depicted in Figure 8. The facts (i) the access pattern is cyclic (*i.e.,* there is a back-edge) and (ii) the number of distinct 64KB VPNs accessed is more than the number of TLB entries, ensure that we would observe at least a few TLB misses if static prefetching was deployed. In contrast, we do not expect any TLB misses if CoLT is used, as long as the number of 64KB VPNs accessed is fewer than 16× of the number of TLB entries.

The experiment accesses few virtual addresses (say, ENTRIES) that are 1MB apart over a contiguous virtual address range in a cycle, starting from `start_idx`. Instead of returning to the starting point to create a circular pattern, the last entry in this chain returns to an offset less than 1MB (here, 512KB) from the original starting address. We repeat this pattern twice to access 2×ENTRIES virtual addresses that are at least 512KB apart in the virtual address region. Since the L2 TLB has 65 entries, we set ENTRIES=65.

As discussed earlier, since the L2 TLB has 65 entries and the access patterns have back-edges (Figure 8), we expect TLB misses if there was static prefetching. In our experiments, we observe *none*. Thus, we determine that L2 TLB uses CoLT by dynamically

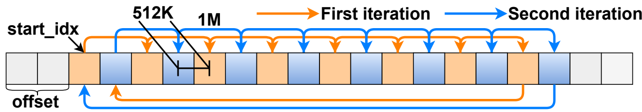


Figure 8: Pointer-chase pattern used to verify the use of CoLT.

coalescing 16 contiguous 64KB VPNs. Using the same experiment but with ENTRIES=1025, we verified that L3 TLB uses CoLT too.

These observations also explain why the L2 and L3 TLB indexing functions (Section 4.2) use address bits starting from the 21st position even though the page size is 64KB. In an implementation of CoLT, the indexing function should ignore the page offset bits (16) and an additional $log(n)$ bits where n is the number of mappings in each TLB entry (here, $n = 16$).

4.4 Sharing and Allocation Policy

We now set out to infer which level(s) of the three-level TLB hierarchy is/are shared across SMs and which level(s) is/are private to an SM. We then decipher the TLB entry allocation policies (*e.g.,* inclusive or exclusive) across the entire hierarchy. This understanding is essential to ensure that the `Trojan` and the `Spy` can communicate via a shared hardware structure deterministically. Additionally, knowledge of the allocation policy aids in forming minimally sized eviction sets for the covert channel.

To infer which level of TLB is shared (if any), we extend the basic pointer-chase algorithm in a couple of ways. First, we make it access a set of virtual addresses such that they cover the entire reach of a given TLB level. For instance, to inspect L3 TLB, the updated algorithm accesses virtual addresses covering 1025MB, while to inspect the L2 TLB, it accesses virtual addresses covering 65MB. Second, we run instances of the algorithm mentioned above in multiple thread blocks. Specifically, the number of thread blocks that we use is double the number of SMs in the GPU.

We then execute two instances of the above kernel in succession, but with different sets of virtual addresses noting the `smid` of SMs where each thread block executes. Finally, we execute the first kernel again with the original set of virtual addresses and measure the differences in access times. If the given TLB level is shared, when the first kernel executes again, all its accesses miss in that TLB level. The execution of the second kernel would evict all entries of the first kernel brought into the TLB. Using this methodology, we confirmed that L3 TLB is shared across all SMs in the GPU.

We are the first to discover a shared L3 TLB on Pascal. The L1 TLB is private to each SM, as interference is only among thread blocks executing on the same SM. Similarly, we found that the L2 TLB is shared across a *subset* of SMs. The observations about L1 and L2 TLBs are aligned with a previous study [19].

Next, we infer the allocation policy used in the TLB hierarchy, *i.e.,* we wish to know whether L3 TLB entries are inclusive, exclusive, or non-inclusive of entries in L1 and L2 TLBs. This is critical to ensure that we can create a covert channel with optimal bandwidth.

If the GPU had an inclusive allocation policy, then the copies of L1 and L2 TLB entries would also be present in the L3 TLB. We experimented to determine if that is indeed the case. For this experiment, we leveraged the indexing functions for the L2 and L3

```
1  barrier = 0 /* Initialize to zero */
2  {
3    /* Thread block 1 performs pointer-chase */
4    atomicAdd (barrier, 1);
5    /* Block 0 returns */
6  }
7  {
8    /* Thread blocks 2-N wait for barrier signal */
9    while (atomicAdd (barrier, 0) != 1);
10   /* Perform pointer-chase */
11 }
```

Figure 9: Pseudocode for synchronization among thread blocks.

TLBs (Section 4.2). We created a pointer-chase similar to Figure 2, but not with the strided access pattern. Instead, we choose addresses that index to the same set in the L3 TLB, but index into different sets of the L2 TLB. We created a pattern with 64 such entries and tracked the latency of each access (Figure 3). As this benchmark executes, it would evict entries from the L3 TLB since all the entries index into the same set of the L3 TLB.

If the L3 TLB were inclusive, then removing an entry from the L3 TLB would invalidate the corresponding entry from the L2 TLB. However, on measuring the access latencies for this experiment, we observed an L2 TLB hit for each access in the pattern. This result shows that entries are evicted from the L3 TLB but not from the L2 TLB, suggesting that *the L3 TLB is not inclusive.*

Suppose instead that the TLB hierarchy followed exclusive allocation, *i.e.,* an entry in any TLB level is the sole copy of it in the entire hierarchy. We experimented to determine whether that is the case. In this experiment, we use multiple thread blocks. Thread blocks of the same kernel get scheduled on different SMs. We launch a kernel with n thread blocks (where $n > 1$), with one warp in each. We used one block (called *block-1*) to access a set of virtual addresses in a pointer-chase fashion. Thread block-1 performs two iterations of the same set of accesses. After that, it signals other thread blocks to access the same set of virtual addresses, while thread block-1 itself completes. These other blocks (*i.e.,* $2 - n$) run concurrently. We use `atomic` primitives for synchronization among thread blocks (Figure 9 shows the pseudocode). We then measured the time that each block took to access the addresses. The number of entries in this pattern is such that they will fit in the L1 TLB.

We observed that accesses from thread block-1 in the first iteration miss in the entire TLB hierarchy (compulsory miss). In the second iteration, however, all accesses hit in the L1 TLB. This confirms that accesses in the first iteration populated the L1 TLB of the SM that ran thread block-1. For the remaining thread blocks ($2 - n$) that executed on different SMs, we observed that their accesses were L3 TLB hits. This is possible only if the first iteration of thread block-1 brought translations both in its L1 TLB *and* in the L3 TLB. Therefore, *L3 TLB entries are not exclusive of those in L1 TLB.* Observations from these experiments help us conclude that the TLB hierarchy uses a *non-inclusive, non-exclusive* (NINE) allocation.

Summary: Table 1 summarizes the results of the experiments reported in this section and presents the TLB configuration. We are the first to report L3 TLB shared across all SMs for Pascal microarchitecture. We discover idiosyncrasies such as the existence of victim entries, coalesced TLB entries in both L2, and L3 TLB. We also decipher the indexing function for L2 and L3 TLBs.

Table 1: Summary of Pascal TLB microarchitecture.

Caching Policy	Non-inclusive non-exclusive (NINE)		
Level	**L1**	**L2**	**L3**
Entries	16	64 + victim	1024 + victim
Organization	Fully associative	8-way Set associative	8-way Set associative
Indexing Function	NA	XOR-3	XOR-7

```
1  /* This function performs pointer-chase over a CovertVAs set.
2     The function "prime" has similar function body, except it
3     does not need to time the accesses. */
4  __device__ unsigned long probe (set) {
5    j = 0;
6    start = clock ();
7    for i in range (0, ITERATIONS)
8      j = set[j];
9    return clock () - start;
10 }
```

Figure 10: Probe function.

```
1  #define BITS_TO_SEND
2  #define LATENCY_THRESHOLD
3  __global__ void sender (int *arr, int *msg) {
4    for (j = 0; j < BITS_TO_SEND; j++) {
5      /* Based on message, probe the information set */
6      if (msg[j] == 0) {
7        prime (CovertVAs1);
8      } else {
9        /* Do nothing here */
10     }
11     /* Wake up receiver waiting for the signal */
12     prime (CovertVAs2);
13     /* Wait for receiver signal */
14     do {
15       time = probe (CovertVAs3);
16     } while (time < LATENCY_THRESHOLD);
17   }
18 }
```

Figure 11: Sender kernel (Trojan).

Most studies on recent microarchitectures either do not focus on TLBs [12, 13], or do not observe the L3 TLB [19]. These studies relied on analysis with `cudaMalloc` and yielded only partial information. Using `cudaMalloc` allocates 2MB pages, which is pinned on the GPU. Since there are 1025 entries in the L3 TLB, one would need to allocate more than 32GB of memory to discover it, compared to a little more than 1GB with UVM. In short, UVM plays a crucial role in reverse-engineering the GPU's TLB hierarchy and forming the channel. Appendix C elaborates more on these details.

5 GPU COVERT CHANNEL VIA TLB

We first create a minimal covert channel using the L3 TLB. We then increase the bandwidth of this channel by exploiting parallelism available in a GPU. Finally, we show how MPS can further increase the channel's bandwidth and reduce bit errors.

5.1 Creating a Minimal TLB Channel

The `Trojan` kernel (Figure 11) sends information to the Spy kernel (Figure 12) using the L3 TLB. As the L3 TLB is shared across all SMs of the GPU, the `Trojan` and the Spy are free to execute on any SM of that GPU.

We follow the popular prime+probe strategy [31, 40, 42] that is often used in cache-based covert and side-channel attacks to create the TLB channel. The Spy first *primes* the TLB by accessing

```
1  #define BITS_TO_RECEIVE
2  #define LATENCY_THRESHOLD
3  __global__ void receiver (int *arr) {
4    __shared__ short msg[BITS_TO_RECEIVE];
5    for (j = 0; j < BITS_TO_RECEIVE; j++) {
6      /* Wait for sender to signal about the bit */
7      do {
8        time = probe (CovertVAs2);
9      } while (time < LATENCY_THRESHOLD);
10     /* Probe information set and record the
11        decoded message in shared memory */
12     time = probe (CovertVAs1);
13     if (time > LATENCY_THRESHOLD)
14       msg[j] = 0;
15     else
16       msg[j] = 1;
17     /* Wake up sender waiting for the signal */
18     prime (CovertVAs3);
19   }
20 }
```

Figure 12: Receiver kernel (Spy).

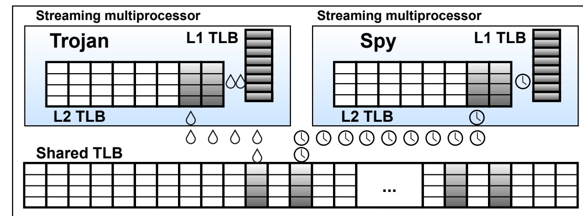


Figure 13: Covert channel using the shared L3 TLB.

a group of virtual addresses (VAs) that fall in chosen sets of the L3 TLB. The `Trojan` then executes and conditionally accesses VAs that index to the same L3 TLB sets, based on whether it wishes to communicate a 0 or a 1. By performing accesses, the `Trojan` evicts entries from the corresponding TLB set. The Spy then *probes* the L3 TLB by accessing the same VAs as it did for priming (see Figure 10). The access latency for VAs evicted from the L3 TLB will correspond to that of L3 TLB misses, allowing Spy to determine if `Trojan` accessed them, consequently inferring the bit that the `Trojan` tried to communicate.

A key challenge in using the L3 TLB for covert communication is that the L1 and the L2 TLBs may filter out the accesses, thereby leaving no footprint in the L3. To overcome this, we use the TLB microarchitecture knowledge from Section 4 and design the `Trojan` and `Spy` to ensure that their accesses always reach the L3 TLB (Figure 13). Specifically, we choose a group of VAs that are certain to overflow the entire L1 TLB and a few chosen sets of the L2. We call this group of virtual addresses *CovertVAs* because they enable covert communication between the `Trojan` and `Spy`. Note that CovertVAs allows the communication of a *single* bit of information, which can be iterated to communicate an entire message. As we shall see, `Trojan` and `Spy` use three CovertVAs (each with a disjoint group of VAs) to enable communication—one group of CovertVAs is used to communicate the secret bit and the other two for synchronization.

First, we determine the size of CovertVAs so that the memory accesses reach the L3 TLB. The size of CovertVA is coupled with the caching policy of the TLB hierarchy. From Section 4.4, we observed that the TLB hierarchy uses the non-inclusive, non-exclusive (NINE) policy. Thus, evicting Spy entries from L3 TLB does not evict entries from L1 and L2 levels. To form CovertVA in the NINE hierarchy,

we need to ensure that there are no L1 and L2 TLB hits. From Section 4, we know that L1 TLB is a fully-associative 16-entry structure. Therefore, to overflow the L1 TLB, CovertVAs should have at least 17 entries. Next, we need to ensure that CovertVAs entries overflow the L2 TLB as well. From Section 4.2, we know that both the L2 and L3 TLBs are 8-way set-associative structures and that the L2 TLB has 8 sets, while the L3 TLB has 128 sets. Since the L3 TLB is much larger than the L2 TLB, it is possible to choose entries for CovertVAs so that they are mapped to a smaller number of sets in the L2 TLB compared to that on L3 TLB.

We use the indexing functions in Figures 6 and 7 to choose elements of CovertVAs such that they map to *one* set of the L2 TLB but to *three* different sets in the L3 TLB. We also need to ensure that the victim entry in the L2 TLB is overflown (Section 4.2). Ultimately, a CovertVAs set used in our experiments contains 17 VAs. Accessing 17 elements ensures that an L1 TLB hit is never observed, avoiding the fully-associative structure. L2 being 8-way set associative, accessing 17 elements of *one* L2 set also ensures that L2 TLB hit is never observed (overcoming the victim entry along). For L3 TLB, we chose *three* sets (6-6-5 entries in *three* different sets). Since the L3 TLB's associativity is 8, if the `Trojan` and `Spy` both access these VAs, then some of the entries are bound to be evicted from the L3 TLB. Later, when `Spy` accesses the same VAs as in the prime phase, it will notice a measurable time difference based on whether `Trojan` accessed the entries (0) or not (1).

Finally, we decide how far apart each chosen VA is from one another. Each VA present across all the CovertVAs is at least 1MB apart. Note that 1MB is not the page-size allocated on using UVM. While forming CovertVAs, we need to ensure that the expected number of entries is occupied in the TLB (here, 17). As the GPU utilizes CoLT-like mechanisms (Section 4.3), to ensure that VAs do not fall in the same coalesced entry, the VAs should be at least 1MB apart (16 contiguous VAs of 64KB coalesce into one TLB entry of 1MB). For any stride smaller than 1MB, the VAs can fall in the same entry and not give expected results. In such cases, more than 17 addresses will have to be accessed, which can reduce the bandwidth of the channel. Additionally, for a stride of 64KB, *i.e.,* page-size, the GPU driver also performs page-promotion (to 2MB) [34], making the CovertVAs not usable as it's memory footprint increases. This increased memory footprint may not fit in the GPU memory.

Figures 11 and 12 show the (curated) pseudocode for the `Trojan` and `Spy`, respectively. Three groups of CovertVAs are required to communicate a single bit of information. CovertVAs_1 is used to communicate one bit of information while the other two (CovertVAs_2 and CovertVAs_3 in the figures) are needed to synchronize `Spy` and `Trojan`. Specifically, lines 6 – 10 in Figure 11 show how `Trojan` conditionally accesses the virtual addresses in CovertVAs_1 based on whether it wants to send 0 or 1. Line 12 shows how `Trojan` signals the `Spy` that it has sent the message using CovertVAs_2. `Trojan` then waits (lines 14 – 16) in a tight loop for `Spy` to signal when it is done reading the message (using CovertVAs_3). `Trojan` repeats these steps to send each bit of information to `Spy`.

`Spy` (Figure 12), on the other hand, waits for the `Trojan`'s signal via CovertVAs_2 (lines 7 – 9) before attempting to read the message sent. On receiving the signal from the `Trojan`, the `Spy` records the information passed by probing CovertVAs_1 in Line 12. Finally, `Spy` signals the `Trojan` that it is ready to receive the next bit (line 18).

The covert channel design above communicates one bit of information in each iteration from `Trojan` to `Spy`. However, the design underutilizes the GPU resources as most of the SMs on the GPU remain idle. To utilize the GPU resources better, we scale up the channel by using all the SMs for covert communication. The code discussed earlier (Figures 11 and 12) run in parallel across all the SMs. More parallelism increases the bandwidth of the covert channel. Specifically, in our experiments (shown later), we used up to 14 warps (each running on a different SM) that execute the code concurrently. We are limited to 14 warps since each warp needs three disjoint CovertVAs—one for communication and two more for synchronization. We carefully select a different set of three CovertVAs for each of the 14 warps. Recall that each CovertVAs consists of a group of VAs that index into three distinct sets in the L3 TLB. Thus, a single warp would need nine sets in the L3 TLB. Since there are only 128 sets in the L3 TLB, only 14 bits can be communicated concurrently in the given experimental setup.

5.2 Enhancing the channel with MPS

The covert channel discussed thus far works in principle, but we found that it was noisy in practice. The primary reason for this is that both `Trojan` and `Spy` cannot execute concurrently (without MPS). We observed that the GPU schedules the thread blocks belonging to the `Trojan` and the `Spy` on the same SMs. Consequently, there are context-switches involved during the communication of each bit between the `Trojan` and the `Spy`. Context-switches are slow, taking 100s of microseconds [57], hurting the channel's bandwidth. Context-switches may also pollute the observed L3 TLB signature left by the `Trojan` before the `Spy` accesses it by affecting timing measurements. As the channel relies on accurate timing measurements, context-switches increase the error rate, reducing the channel's effective bandwidth further.

MPS helps us address both the problems. As mentioned in Section 2, MPS allows kernels from different processes to execute on the same GPU concurrently. By executing the `Trojan` and `Spy` kernels as two separate MPS processes, they execute concurrently on different SMs of the same GPU. Context-switches are no longer necessary between executions of `Trojan` and `Spy`. Since they execute on different SMs they interfere only in the shared L3 TLB, as required. As a result, the error rate in data transmission reduces with the use of MPS. Furthermore, by avoiding the long latency context-switches, the bandwidth of the channel increases by 40×.

We utilize all the 28 SMs (14 each for `Trojan` and `Spy`) for covert communication on our GPU setup with MPS enabled. However, on GPUs with more number of SMs, there will still be idle SMs. These idle SMs can be the source of noise by other co-running applications. We discuss a method to avoid such noise in Section 7.

5.3 Channel Measurements

The `Spy` and the `Trojan` kernels are launched from two different CPU processes on the GPU. We exercise the channel by varying the number of thread blocks, a 20-bit long message per thread block, both with and without MPS. We present the maximum observed bandwidth of data transmission (with no bit-errors) by exercising the channel 100 times for each thread block count in Figure 14. Note the logarithmic y-axis in the figure.

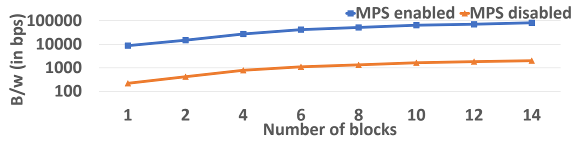


Figure 14: Maximum bandwidth of the channel.

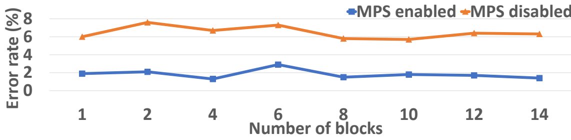


Figure 15: Average bit error rate in the channel.

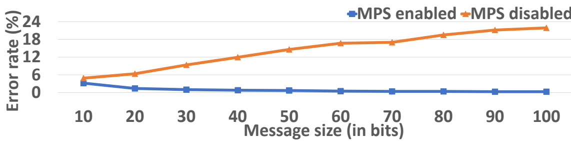


Figure 16: Average bit error rate for different message sizes.

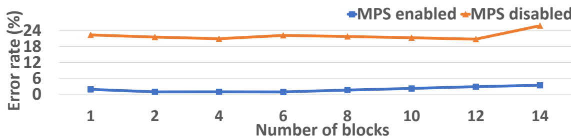


Figure 17: Average bit error rate in database application.

We observe that the bandwidth of the channel without MPS goes from a couple of hundred bits-per-second (bps) to over two thousand bps. Importantly, when we enable MPS, the bandwidth increases by 40× to 81Kbps since both the `Trojan` and the `Spy` simultaneously execute on the GPU.

We measured error rates in the channel by comparing the transmitted bits against the ones that `Trojan` intended to transmit. Figure 15 presents the measurements by varying the number of thread blocks, with and without MPS. We observe that the channel has a higher error rate without MPS compared to when MPS is enabled Thus, MPS reduces the error rate and boosts the bandwidth of the channel significantly.

Finally, for the channel to be useful, it should be able to communicate long messages with a low error rate. We compare the error rate of the channel, with and without MPS, by fixing the thread block size to 14 and varying the message size. Figure 16 shows the result we observed. It shows that the channel with MPS disabled has error rates that increase with the message size. We believe this is due to context-switches interfering with the timing measurements more often, increasing the possibility of misread bits. In contrast, MPS reduces the error rate for long messages as well, as there are no interfering context-switches.

6 LEAKING DATA FROM AN APPLICATION

To understand the utility of this channel, we show how a maliciously-modified GPU application can take advantage of this channel to leak data. We accomplish this using the Virginian Database [3], a GPU-accelerated database library. GPU-accelerated databases primarily take advantage of available parallelism to accelerate insert, update and search operations. We use an in-house modified version of the library that uses the GPU to perform parallel insert operations. It additionally has the `Trojan` (Figure 11) as another kernel.

The (malicious) library intends to leak a few rows worth of data while performing parallel insert into the database. We assume that the schema of the table is already known to the `Spy`. In an attack scenario, the `Spy` will be running on the GPU, waiting for the `Trojan`, to be invoked. When the `Trojan` runs, the synchronization primitives described in the previous section communicate data as needed.

The `Trojan` leaks one row of data per thread block. Each row constitutes of 24 bytes of data. Figure 17 shows the error rate we observed when the database library was used to leak information by varying the number of blocks while performing 1000 rows of insert operation into the database. Here, when MPS is enabled, the channel has low error rates as well.

To our knowledge, there are no known defenses for this novel GPU TLB covert channel. The reader may note that CPU-based covert channels such as flush+reload [60] or other GPU-based channels [29] offer higher bandwidth than our channel. However, many proposed mechanisms defend against these attacks well [49, 57]. Our channel being novel can still be exploited on systems that defend against the aforementioned faster channels.

7 DISCUSSION

- **Effect of co-running applications.** Covert-timing channels often suffer from noise added by other applications co-running on the hardware. In our channel, this noise occurs due to global memory accesses by these applications, changing the L3 TLB state. Such noise affects the accuracy and the bandwidth of the channel. Applications that run on SMs not occupied by `Spy` and `Trojan` can impact our channel's accuracy. MPS allows the launch of these applications, provided the GPU has sufficient resources (*e.g.,* SMs). To avoid noise, `Spy` ideally needs to occupy these remaining SMs (>28, 14 each for `Spy` and `Trojan`). The `Spy` can launch more dummy thread blocks than the number of remaining SMs. These thread blocks occupy SMs until the actual `Spy` and `Trojan` finish execution (without adding noise). Dummy thread blocks can reserve all of shared memory (48KB on our GPU), preventing other blocks from being scheduled on those SMs. To synchronize with actual `Spy` blocks, we can use synchronization mechanisms similar to Figure 9. No other application will co-run on the GPU once `Spy` and `Trojan` start, making the channel noise-free.

- **Existing defensive mechanisms.** Most prior work on defenses for timing channel attacks on CPUs are based on techniques built using physical addresses [49]. The GPU TLB channel operates on virtual addresses, and defenses from the literature do not directly apply to it. Prior work has also developed defenses, primarily focusing on GPU-based intra-SM channels [57]. It relies on performance-counter metrics for detecting resource contention. Such counters

for TLBs are not publicly available. Though their mechanism can be extended to detect our attack, they fall back to temporal sharing on detection of an inter-SM channel, which will hurt GPU utilization and performance. Recent architectural advancements such as Multi-instance GPUs (MIG) [39] provide hardware support for “isolated" sharing of GPU across applications by exposing up to 7 instances of *statically* partitioned GPU resources. Static partitioning often leads to resource under-utilization. Furthermore, MIG allows multiple processes to share resources within a single static instance that makes the proposed channel possible within an instance.

- **Possible mitigations.** The primary factors that enable the channel are the shared L3 TLB and spatial sharing of GPU by the applications. Spatial sharing is essential for GPU utilization. Thus, shared TLB should be dealt with efficiently. The L3 TLB can be partitioned dynamically between applications using different mechanisms [49]. However, it may increase runtime latency. Our channel relies on precise timing for measuring contention and synchronization to achieve higher bandwidths. Thus by adding disruptions to timing measurements [9, 26], *i.e.,* to `clock` measurements, the channel can be hampered significantly. However, this affects benign applications with precise timing requirements. Another approach is to generate random memory requests on a conflict miss in L3 TLB, adding noise to multiple data observing sets. However, the challenge here will be to ensure that these random requests do not cause preliminary *far-faults* [61], leading to further performance degradation.

8 RELATED WORK

Much recent attention has been devoted to timing-based covert channel and side-channel attacks. Most of these attacks focus on CPUs; we focus here on discrete GPUs and TLB-based channels.

Reverse-engineering GPU microarchitecture. One of the earliest works on reverse-engineering GPU microarchitecture was performed on GT200 GPU, specifically, the early Tesla microarchitecture [41, 56]. More recently, Mei *et al.* [28] studied the memory hierarchy of three generations (Fermi, Kepler, and Maxwell) of Nvidia GPUs. Both these works are on older architectures and are not directly relevant to our work.

Karnagel *et al.* [19] studied the TLB hierarchy of Nvidia Pascal and Kepler microarchitectures. In this paper, we showed that their findings are incomplete. In particular, we discovered an additional level of the TLB, and found the existence of a victim entry in the TLB. Jia *et al.* [12, 13] have conducted an extensive study of the Volta and Turing microarchitectures, but have not studied the TLB hierarchy in great detail.

GPU Covert and Side-Channels. Lee *et al.* [21] were one of the first to identify security concerns with GPUs. Their study exposed concerns regarding GPU memory management, such as allocation of non-zeroed pages and non-erasable memory regions. They also demonstrated that GPUs do not prevent kernels from reading memory written to by a recently-finished kernel. They exploited these vulnerabilities to extract web page information when GPUs were used to rendering them. Di Pietro *et al.* [44] demonstrated similar leaks targeting shared memory, global memory, and register state. Naghibijouybari *et al.* [29] established covert channels using the caches and functional units on Nvidia Fermi, Kepler, and Maxwell GPUs. However, they did not use the TLB as a covert channel.

Covert channels and side-channels are closely related, and the presence of a covert channel often suggests that one can similarly engineer a side-channel. Luo *et al.* [25] demonstrated a power side-channel attack by sampling and processing power traces on a GPU to extract secret keys of AES. Jiang *et al.* [14] demonstrated the recovery of an AES-128 key using a timing side-channel involving the co-relation between latency of data cache and memory coalescing efficiency. They demonstrated a similar attack on table-based AES encryption [15] by understanding the correlation between a table lookup and bank conflicts. Table-based AES was also compromised by a differential timing attack using shared memory bank conflicts [16]. Naghibijouybari *et al.* [30] demonstrated side channels on GPUs using coarse-grained metrics like memory utilization, performance counters, and timing. Wei *et al.* [54] trained multiple LSTM models to extract DNN model secrets on GPU. Luo *et al.* [24] constructed timing models of an RSA implementation on GPU and demonstrated a timing attack to extract the RSA private keys.

Recent studies have demonstrated multiple mechanisms to defend against these covert and side-channel attacks. To defend against side-channels targeting the memory coalescer, Kadam *et al.* [17] proposed `RCoal`, which introduces randomization to the coalescer. `Bcoal` [18] improves upon `RCoal` to handle additional cases where the coalescing can still happen at Miss Status Holding Registers. These defense mechanisms defend against memory coalescing attacks and, as such, do not impact our covert channel. Xu *et al.* [57], proposed `GPUGuard` to mitigate intra-SM contention-based covert and side-channels. For inter-SM resources (such as the L3 TLB), they fall back to temporal sharing, affecting GPU utilization. Hunt *et al.* [10] demonstrated a proof-of-concept side-channel attack on GPU TEEs by timing the communication of messages between the host and the GPU. They propose a defensive mechanism by making communication with the GPU data-oblivious, which does not defend against channels that use on-GPU resources.

TLB-based channel. Gras *et al.* [8] demonstrated a TLB-based side-channel on CPUs. It relies on the hyper-threading mechanism to share TLB between two processes. In contrast, our covert channel relies on a globally shared TLB available on modern GPUs by design. Deng *et al.* [6] proposed defensive mechanisms for the attack, such as static-partitioning of ways in the set-associative structure of per-core private TLBs. However, it brings in significant performance overheads, *e.g.,* MPKI increases significantly (3×). It also proposed another mechanism that relies on segregating process address space into a secure and a non-secure region. Such segregation is not present in processes that want to communicate covertly.

9 CONCLUSION

We demonstrated a novel covert timing channel on a GPU via shared L3 TLB. We discovered three TLB levels, victim entries, and coalescing of contiguous virtual pages in L2 and L3 TLB on Nvidia's GPUs. We constructed the indexing functions of L2 and L3 TLBs by careful analysis of eviction sets. This channel is possible due to UVM, a programmability feature. We improved the channel's bandwidth by 40× using MPS to execute the Trojan and the Spy concurrently. We showed the utility of the channel by leaking the contents of a database application.

ACKNOWLEDGMENTS

We thank Aditya Kamath, Shweta Pandey, and Ashish Panwar for providing helpful feedback on an earlier draft of this work. This work was partially supported by a Ramanujan Fellowship from the Government of India, Pratiksha Trust, Bangalore, a grant from the Robert Bosch Centre for Cyber-Physical Systems at IISc, and by research gifts from VMware and Intel.

REFERENCES

[1] Amazon. 2020. P3 instances with V100. https://aws.amazon.com/ec2/instance-types/p3/
[2] AMD. 2019. Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors. https://www.amd.com/system/files/TechDocs/56305.zip
[3] Peter Bakkum and Srimat Chakradhar. 2012. Efficient Data Management for GPU Databases. (2012).
[4] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report.
[5] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems.*
[6] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *Proceedings of the International Symposium on Computer Architecture.*
[7] Google. 2019. Cloud GPUs. https://cloud.google.com/gpu/
[8] 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 Symposium.*
[9] W. . Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy.*
[10] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation.*
[11] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design.*
[12] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *Computing Research Repository (CoRR), arXiv* (2019).
[13] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *Computing Research Repository (CoRR), arXiv* (2018).
[14] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *International Symposium on High Performance Computer Architecture.*
[15] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017.*
[16] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2019. Exploiting Bank Conflict-Based Side-Channel Timing Leakage of GPUs. *ACM Transactions on Architecture and Code Optimization* (2019).
[17] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. 2018. RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques. In *IEEE International Symposium on High Performance Computer Architecture.*
[18] G. Kadam, D. Zhang, and A. Jog. 2020. BCoal: Bucketing-Based Memory Coalescing for Efficient and Secure GPUs. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).*
[19] 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 *International Workshop on Data Management on New Hardware.*
[20] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. 2017. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *54th ACM/EDAC/IEEE Design Automation Conference.*
[21] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *IEEE Symposium on Security and Privacy.*
[22] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security.*
[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy.*
[24] Chao Luo, Yunsi Fei, and David Kaeli. 2019. Side-Channel Timing Attack of RSA on a GPU. *ACM Transactions on Architecture and Code Optimization* (2019).
[25] C. Luo, Y. Fei, P. Luo, S. Mukherjee, and D. Kaeli. 2015. Side-channel power analysis of a GPU AES implementation. In *IEEE International Conference on Computer Design.*
[26] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture.*
[27] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien" Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions and Defenses.*
[28] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU Memory Hierarchy Through Microbenchmarking. In *IEEE Transactions on Parallel and Distributed Systems.*
[29] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and Characterizing Covert Channels on GPGPUs. In *IEEE/ACM International Symposium on Microarchitecture.*
[30] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *ACM SIGSAC Conference on Computer and Communications Security.*
[31] Michael Neve, Seifert, and Jean-Pierre. 2007. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography.*
[32] Jouppi Norman. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *ACM SIGARCH Computer Architecture News.*
[33] Nvidia. [n.d.]. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Accessed: 2020-07-20.
[34] Nvidia. 2017. Maximizing Unified Memory Performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/
[35] Nvidia. 2017. Unified Memory for CUDA Beginners. https://devblogs.nvidia.com/unified-memory-cuda-beginners/
[36] Nvidia. 2019. Documentation for Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html
[37] Nvidia. 2019. GPUs everywhere. https://blogs.nvidia.com/blog/2017/05/08/microsoft-azure-gpu-instances/
[38] Nvidia. 2019. NVIDIA open-gpu-doc repository. https://github.com/NVIDIA/open-gpu-doc
[39] Nvidia. 2020. Nvidia Multi-instance GPUs. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
[40] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of "AES". In *RSA Conference on Topics in Cryptology.*
[41] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. 2009. *Microbenchmarking the GT200 GPU.* Technical Report. Computer Group, ECE, University of Toronto.
[42] Colin Percival. 2005. Cache Missing for Fun and Profit. In *Proc. of BSDCan 2005.*
[43] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *IEEE/ACM International Symposium on Microarchitecture.*
[44] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. In *ACM Transactions on Embedded Computing Systems.*
[45] W. V. Quine. 1952. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly* (1952).
[46] M. K. Qureshi. 2019. New Attacks and Defense for Encrypted-Address Cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA).*
[47] R. H. Saavedra and A. J. Smith. 1995. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.* (1995).
[48] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. 2018. Neighborhood-aware Address Translation for Irregular GPU Applications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture.*
[49] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (2019).
[50] Ristenpart Thomas, Tromer Eran, Shacham Hovav, and Savage Stefan. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM Conference on Computer and Communications Security.*
[51] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *USENIX Security Symposium.*
[52] Pepe Vila, Boris Köpf, and José F Morales. 2019. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy (SP).*
[53] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture.*
[54] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque. 2020. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*
[55] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization. In *Proceedings of the 28th USENIX Conference on*

Security Symposium.

[56] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software.*

[57] Q Xu, H Naghibijouybari, S Wang, N Abu-Ghazaleh, and M Annavaram. 2019. GPUGuard: Mitigating Contention Based Side and Covert Channel Attacks on GPUs. In *ACM International Conference on Supercomputing.*

[58] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).*

[59] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP).*

[60] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium.*

[61] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).*

A EXTENDED POINTER-CHASE

The extended pointer-chase experiment aims to overcome the odd observations made by the pointer-chase algorithm (Figure 2), possibly due to the victim cache [32], and find true *eviction sets.*

First, we understand how the pointer-chase algorithm reports eviction sets in the presence of victim cache. The victim cache keeps track of entries from TLB sets that last observed a conflict miss. Assume an 8-way set-associative TLB with 8 sets and a victim cache of size 1. For a given starting index (`start_idx`) in the pointer-chase pattern, set-1 of TLB first observes a conflict miss. When a new address is added to the pattern, for the same `start_idx`, set-2 observes a conflict miss. This conflict miss cannot be managed in a single entry victim cache, resulting in TLB overflow. This overflow results in the observation of 1 eviction set of size 17 (8 from set-1 and set-2 each, plus a conflict victim of set-1). Later additions of addresses to the pattern result in observing the remaining 6 eviction sets of size 8, observing a total of 7 eviction sets.

Next, we describe the experiment, which is divided into two steps: ① form many eviction sets using original pointer-chase algorithm multiple times, ② merge the eviction sets which are formed across iterations. In step 1, we aim to form the eviction set of size 17 with different TLB sets. We run the pointer-chase algorithm over a large virtual address region multiple times to achieve this, but at a different starting address in each iteration. In iteration 1, we run the algorithm, forming the pattern starting at `start_idx`. This pattern should lead to the scenario as detailed in the previous paragraph. In iteration 2, when we rerun the algorithm over a different `start_idx` (offset of 4MB compared to the previous iteration), two different sets become part of the eviction set of size 17 (say, set-3 and set-4), as the first set that sees a conflict miss is now set-3. Note that the sets that were part of the *larger* eviction set (size 17) in iteration 2 were part of smaller eviction sets in iteration 1. We continue these steps multiple times and collect eviction sets of all sizes.

In step 2, we "merge" the eviction sets based on shared address in them. While merging, we ignore the eviction sets of size 17. We can safely ignore the *larger* eviction set as they are formed from 2 sets of the TLB. Note that we only ignore *larger* eviction sets. The addresses that are ignored (say from iteration 2) are already part of the "merge" set from iteration 1. This step is safe as we assumed that the indexing function is static. This iterative ignore and merge operation captures addresses that truly lie in the same set of TLB. At the end of this process, the number of resultant eviction sets is the true count of the number sets present in a set-associative TLB. Indeed, we observed 8 "merged" eviction sets for the L2 TLB and 1024 sets for L3 TLB on our GPU (Section 4.2) using the above methodology, proving our victim cache hypothesis.

$Group_1(0,1,2,3)$ = 11111111111110100100--1--1--

$Group_2(4,5,6,7)$ = 11111111111110100100--1--0--

$Group_3(0,1,4,5)$ = 11111111111111010010-0--1--1-

$Group_4(0,2,4,6)$ = 11111111111111010010--0--0--0

Figure 18: Few minterms observed for different groupings in our analysis.

B INDEXING FUNCTION

Here, we describe how we constructed the indexing functions in Figures 6 and 7. Our indexing functions are inspired from a recent work, found using similar methodology [8] which suggests a $\oplus$-based indexing function for TLBs on Intel CPUs.

In $\oplus$-based indexing functions, the eviction sets show a peculiar pattern when *grouped* together. The pattern is made up of *don't care* bit positions in addresses from the eviction sets. These bits are identified when we construct a minimized Boolean function using the same addresses. We use an implementation of Quine-McCluskey [45] solver for our purpose. The solver returns a minimized Boolean function in *Sum-of-Products* (SOP) form, marking the *don't care* bits with the "–" symbol. Each product in the SOP form is a *minterm*, representing all the bit positions beyond page-offset (bits 20-48, inclusive). The *don't care* symbols indicate the bit positions whose values are irrelevant to the function. We form 2 distinct groups of eviction sets and feed the addresses from the group to the solver. We observe the differences in the output function, specifically the position of *don't care* symbols.

Before we group the eviction sets, we label them [0-7] for an 8-way set associative structure. Figure 18 lists a *minterm* from each group obtained from their SOP function. We form $Group_1$ from the union of all the addresses in eviction sets (0-3). Similarly, we form $Group_2$ from the remaining eviction sets. From the figure, we can observe that the groups follow a similar trend of "–", but the bits which are not marked are different. Specifically, the $\oplus$ of these bits differ, which gives us the values of bit-positions that decide if an address falls in $Group_1$ or $Group_2$. We uniquely identify each set by forming more distinct groups and finding values corresponding to the remaining bit-positions. In Figure 18, we can uniquely identify set-0 using $Group_1, Group_3$, and $Group_4$ along with $\oplus$-values at different bit positions. We constructed the indexing function in Figure 6 using the above methodology and empirically verified the functions against more than 100 samples collected with the extended pointer-chase experiment. Similarly, we constructed and empirically verified the indexing function for L3 TLB (Figure 7). The construction of the covert channel gives us further confidence in the accuracy of the function.

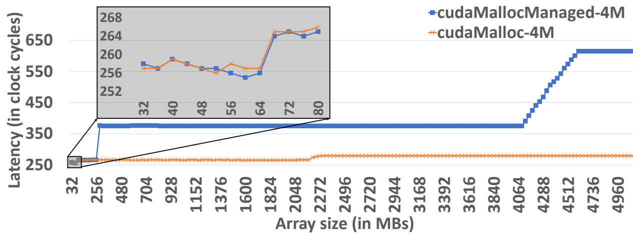


Figure 19: `cudaMallocManaged` vs. `cudaMalloc`.

C OBSERVATIONS WITH THE DEFAULT MEMORY ALLOCATOR

We elaborate on why UVM plays a crucial role in channel formation. We achieve this by comparing the results from Section 4 with experiments we ran using the default memory allocation method, *i.e.,* `cudaMalloc`. We use the pointer-chase algorithm in Figure 2, allocating memory with `cudaMalloc` instead, capturing access times with varying `size` values, and keeping `stride` constant.

Figure 19 shows the results when we set `stride` as 4MB, with the x-axis as `size` and the y-axis as access times (in clock cycles). The first steps for both curves are zoomed in. On using the UVM API, *i.e.,* `cudaMallocManaged`; we observe *three* steps at different `size` values. Specifically, the values are proportional to our findings in Section 4.1, *i.e.,* 1MB vs. 4MB. However, on using `cudaMalloc`, we observe only *two* steps (the second step around the 2100MB mark). We performed similar experiments to confirm the associativity, indexing function, and presence of CoLT-like mechanisms using `cudaMalloc`. We confirmed that 2MB pages are allocated on using `cudaMalloc`. The L2 TLB has 65 entries and supports CoLT-like mechanisms for 2MB pages as well, where 16 contiguous virtual addresses of 2MB are coalesced, creating an illusion of 32MB page size, as suspected in previous studies [12, 13, 19]. This explains why the second step is shallow and at a different `size` value. Every increase in `size` beyond the L2 TLB reach leads to 1 TLB miss and 7 TLB hits in that TLB level (a 32MB entry can keep track of 8 virtual addresses separated by 4MB `stride`). Whereas, in case of UVM, every access beyond the TLB reach is a TLB miss (the coalesced entry size is 1MB). By extension, we believe that the L3 TLB also supports CoLT-like mechanisms, and to overflow it using `cudaMalloc`, we need at least 32GB of physical memory (1025 entries $\times$ 32MB per-entry). However, with UVM, this memory requirement comes down to $\approx$ 1GB, making the covert channel possible even on GPUs with less on-board memory.