Design, Implementation, and Analysis of a TLB-based Covert Channel on GPUs

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Nayak Ajay Ashok

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2021

Declaration of Originality

I, **Nayak Ajay Ashok**, with SR No. **04-04-00-10-22-18-1-15694** hereby declare that the material presented in the thesis titled

Design, Implementation, and Analysis of a TLB-based Covert Channel on GPUs

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-2020**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Wednesday 9th June, 2021 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name(s):

① Dr. Vinod Ganapathy Advisor Signature

② Dr. Arkaprava Basu Advisor Signature

DEDICATED TO

My Parents

and all those who supported me through this journey

Acknowledgements

First and foremost, I would like to thank Dr. Vinod Ganapathy, and Dr. Arkaprava Basu, for agreeing to be my mentors. Working under their wing was an immense pleasure, continuously learning something new regularly. They have inspired me greatly by their different approaches towards pursuing research and striving for excellence.

I want to thank my close collaborators Pratheek and Swaroop, with whom I spent most of the time discussing projects. Without them, working on this thesis would have been a lot less interesting. I want to thank my fellow lab-mates from CSSL, Subhendu, Aditya, Kripa, Rounak, Arun, Rakesh, Nikita, Abhinivesh, Chinmay, Akash, for making the lab a lively place. Anytime I wanted to discuss architecture, the members from CSL, mainly Vinay, Venkat, Aditya, Ravi, Shweta, Akash, Neha, Rajat, were always there for a fun debate, even when off-topic. For that, I am grateful. Talking with members from both the labs and getting to know their exciting work was a great experience.

I want to extend my gratitude to the staff at the CSA office, Mrs. Padmavathi, Mrs. Meenakshi, Mrs. Kushael, and Mrs. Nishitha, for their help with administrative tasks. I would also like to thank all the technical and non-administrative staff at CSA for making the department a great place to be in.

I want to thank my parents, elder brother for guiding me throughout my journey. Finally, I want to thank all my relatives for their immense support throughout the years, especially Vivek, Sangeetha, and cousins Saanvi and Saathvik, for whenever I missed home, I was always welcome at theirs'.

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 are continuously trying 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 thesis, 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.

Publication based on this Thesis

(Mis)managed: A Novel TLB-based Covert Channel on GPUs. Ajay Nayak, B. Pratheek, Vinod Ganapathy, and Arkaprava Basu. *To appear in the proceedings of the 16th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2021).*

Contents

List of Figures

LIST OF FIGURES

List of Tables

List of Code Snippets

Chapter 1

Introduction

As GPUs continue to find favor among an increasing number of application developers, most major public cloud providers have added GPUs in their infrastructure [9, 1, 38]. 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) [39]. MPS allows GPU kernels launched from different processes to execute concurrently on a GPU. Another recent feature such as Unified Virtual Memory (UVM) [36], presents a unified view of CPU and GPU memory, thereby easing the development of GPU kernels. Before UVM, developers had to manually copy data structures between onboard GPU memory and CPU memory. 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 useful to ease programming and improve the utilization of GPUs. However, in this thesis, 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.,* [44, 42, 4, 5, 24] and GPUs [31, 16, 17]). There is also an active body of research on developing defenses against these timing channels (*e.g.,* [48, 60, 55, 22, 57]). This thesis 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 [10], 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 that there is little 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 TLB hits and misses.

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 thesis 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 onboard 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 [42, 33, 44]. 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.

GPUs are primarily used for applications that exhibit high-degree of parallelism. The covert channel can benefit from this parallelism and improve the bandwidth. The `Trojan` and the `Spy` can

have multiple instances of their said programs on the GPU, thereby leveraging the entire available hardware for covert communication. Each instance of `Trojan` has a corresponding `Spy` instance, where the `Trojan`-`Spy` pair communicate using separate sets of virtual addresses to avoid interference caused by other pairs.

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 the `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.

1.1 Contributions

In this thesis, 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 [45] 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.

1.2 Outline

The rest of the thesis is organized as follows:

- **Chapter 2:** We provide the background necessary to understand the thesis. We describe the GPU architecture and two new features available on Nvidia GPUs. We also briefly discuss about covert channels, and TLB-based attacks demonstrated in the literature.
- **Chapter 3:** We define the goal of the thesis while providing the threat model we consider in forming our channel.

- **Chapter 4:** We reverse-engineer the details of the TLB hierarchy. We describe the conducted experiments and provide the microarchitectural details concluded such as the reach, the associativity, and the caching policy of the TLB hierarchy.
- **Chapter 5:** We perform analysis using `cudaMalloc`, thereby understanding shortcomings of previous studies and solidifying the importance of UVM in our channel.
- **Chapter 6:** We evaluate the TLB-based covert channel by constructing a minimal channel, use parallelism on the GPU to increase bandwidth, and finally exploit spatial sharing to improve the bandwidth 40×.
- **Chapter 7:** We leak data of a modified application using our novel channel. This demonstrates the merit of the channel and it's applicability.
- **Chapter 8:** We mainly focus on works that focus on GPU-based side and covert channels. We also discuss about defensive mechanisms proposed in the literature, and briefly provide overview of state of TLB-based attacks on CPUs.
- **Chapter 9:** We provide concluding remarks on the thesis with interesting future research aspects.
- **Appendices**
  - **Appendix A** We provide an analysis of a XOR-based indexing function and study the interaction of values that belong to different sets formed by the indexing function using a Quine-McCluskey solver.
  - **Appendix B** We delve into the details of the methodology and provide an algorithm we used to verify the indexing function of TLBs on Nvidia Pascal-based GPUs.

Chapter 2

Background

In this chapter, we first discuss GPUs, their architecture, programming paradigm, and two new features that are of primary focus in this thesis. We also describe covert channels and TLB-based channels demonstrated in the literature so far, motivating the reason to explore such channels on GPUs.

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 2.1 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 that are shared across SIMD units of that SM. All SMs share a larger L2 cache that is connected to the onboard memory. The GPU's onboard 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 amount of address 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 [21, 50]. Indeed, one of the core contributions of this thesis is to reverse-engineer the TLB hierarchy of a modern GPU.

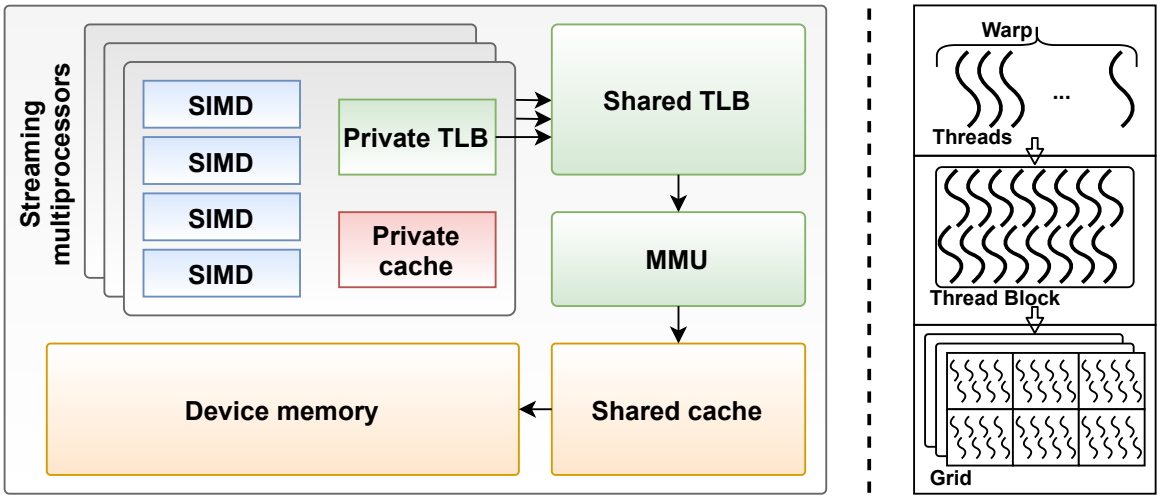


Figure 2.1: GPU architecture and programming model.

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 2.1). We here focus on CUDA, used to program Nvidia's GPUs [35]. A CUDA program consists of portions that must execute on the CPU (Snippet 1) and others that must execute on the GPU (Snippet 2). 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) [36].** 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 CPU to GPU's memory. This paradigm also limited the maximum allocatable memory in a GPU kernel to a relatively small onboard 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 onboard memory. In contrast, the size of memory allocated using CUDA's default API, *i.e.,* `cudaMalloc`, is limited to the GPU's onboard memory size. Furthermore, we discovered that the memory allocated using `cudaMallocManaged` is typically mapped using relatively smaller page sizes (64KB, Chapter 4) than the memory allocated using `cudaMalloc` (2MB, Chapter 5).

• **Multi-Process Service (MPS) [39].** Earlier, a GPU could concurrently run kernels launched from the same application process only, via CUDA Streams. Jobs from different CPU processes or applications time-share the GPU resources. 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, 9], 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. We expect this trend to continue in the future.

2.2 Covert Channels and TLBs

A covert channel is an ability through which two processes (a sender and a receiver) collude and pass information among each other. It is a special attack as the underlying security policies forbid the processes from communicating with each other. However, the policy enforcer (for example, the operating system) is unable to detect the said communication channel, thus the covert nature (Figure 2.2, where `Trojan` is the sender and `Spy` is the receiver). A variant of the covert channel, called the timing channel, has been studied extensively in the past. Timing channels use a global resource shared between the two processes to create timing variances. Microarchitectural resources, such as caches [51], directories [61], way-predictors [24], have been used for covert communication. In timing channels, the receiver observes timing variations on certain operations (say, memory accesses) to discern information the sender tried to communicate. Data is communicated in the form of stream-of-bits, which later is decoded by the `Spy` to form meaningful information.

TLBs have also been used as a vector in timing-channel attacks in the past. The attacks (relying

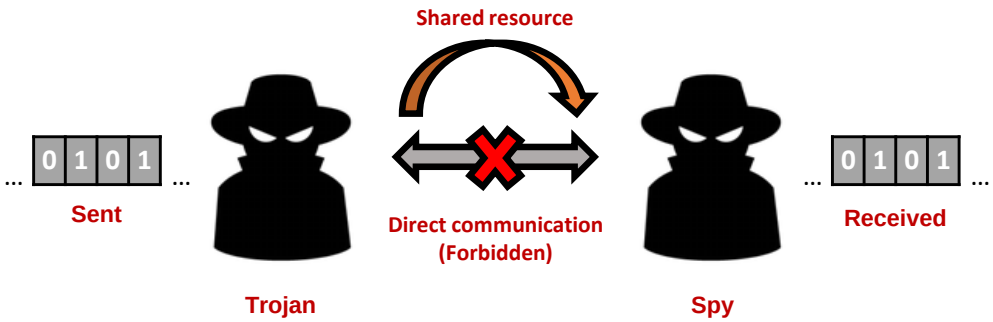


Figure 2.2: A covert channel.

on hit/miss information) demonstrated so far in the literature primarily apply on the CPU side. TLBleed [10] 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 hyperthreads, *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 [14, 15, 21]. 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.

Chapter 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 an explicit 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 [42, 33, 25] 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 the `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 TLB (L3) to enable the covert channel. While prior work has developed cache-based side-channel and covert channel attacks on both CPUs [25, 61, 62, 5, 4] and GPUs [31, 16, 17], 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 thesis 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 with 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 GPU from unprivileged user-space processes.

In this thesis, 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 [52, 53] and GPUs [31, 32]. 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 simply assume that the attacker has managed to co-locate the kernels on the same machine.

Chapter 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 into 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 TLB details of other commercial GPUs too. We make some standard assumptions in our approach: ① the TLB uses Least Recently Used (LRU) replacement policy (or some approximation thereof, such as Tree-LRU), ② the indexing function is static, *i.e.,* it does not vary across kernel runs.

```
#define REPEAT 1
#define STEP   8
void generate (int min_size, int max_size, int stride) {
    /* Allocate a large address space. */
    int *arr = cudaMallocManaged (REPEAT * max_size + 2);
    for size in range (min_size, max_size, stride) {
        for j in range (0, REPEAT) {
            start_idx = j * STEP * stride;
            end_idx = start_idx + size;
            /* Generate pointer chase pattern */
            for i in range (start_idx, end_idx) {
                next_idx = i + stride
                arr[i] = next_idx > end_idx ? start_idx : next_idx;
            }
            /* Launch Kernel */
            p_chase<<1, 1>>(arr);
        }
    }
}
```

Snippet 1: Generating a pointer chase.

```
#define PROBES 1024
__global__  void p_chase (int *arr) {
    int j = 0, old_j, sum = 0, iter = 1;
    /* Warmup the TLBs and caches before timing it */
    /* Now track access time for each access */
    for i in range (0, PROBES * iter) {
        old_j = j;
        start = clock ();
        j = arr[j];
        /* A dependent instruction */
        sum += j;
        end = clock ();
        /* Store access time at immediate next index */
        arr[old_j + 1] = end - start;
    }
    /* To make sure sum is not optimized out, use it */
    arr[2] = sum;
}
```

Snippet 2: Pointer chasing kernel.

At the core of our reverse engineering effort is the pointer chasing algorithm (Snippet 1) that accurately measures the time for repeatedly looping over an array `arr`. 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 next access can be initiated (data dependence), this access pattern ensures that hardware cannot parallelize or start an access before

the previous access is complete. The compiler may additionally perform code optimizations such as loop unrolling and code movement. Therefore, we check the SASS generated for the code to see if code-section being timed is memory access only.[1] This enables accurate latency measurement for each memory 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 algorithm are commonly used in reverse-engineering data caches in CPUs and GPUs [49, 58]. 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 as and when required (summarized later).

A challenge in studying TLB is that latency measurements can get noisy due to hits and the misses in the data cache for the array element being accessed. This 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.[2] Second, in all our experiments, we ensured that accessed data fits inside the L2 data cache and thus always result in hits. Therefore, any observed variations in timing can directly be attributed to the TLB behavior only. This was necessary since, unlike L1 cache, there is no publicly-available way to disable the L2 cache. Note that all our memory accesses fit in the L2 cache, and never use the L1 cache. Therefore, cache prefetchers do not interfere with our latency measurements. Table 4.1 summarises our experimental setup, specifically the GPU runtime used for our reverse-engineering experiments.

GPU model	Nvidia 1080Ti
Microarchitecture	Pascal
Cuda toolkit version	10.0
Cuda driver version	410.1

Table 4.1: Experimental Setup.

4.1 Levels of TLB and their reach

We set out to infer the number of levels of TLB and their respective reach. However, we are not the first to attempt reverse-engineering of the GPU's TLB hierarchy. Wong *et al.* [58] have attempted the same before. However, our use of `cudaMallocManaged`, instead of `cudaMalloc`, allowed us to discover many nuances (*e.g.,* three levels of TLB, coalesced TLB entries) that were not previously reported.

[1] SASS can be generated using `cuobjdump --dump-sass <cuda_executable>`

[2] The specific flag is `-Xptxas -dlcm=cg -Xptxas -dscm=wt`. We verified its validity using hardware performance counters.

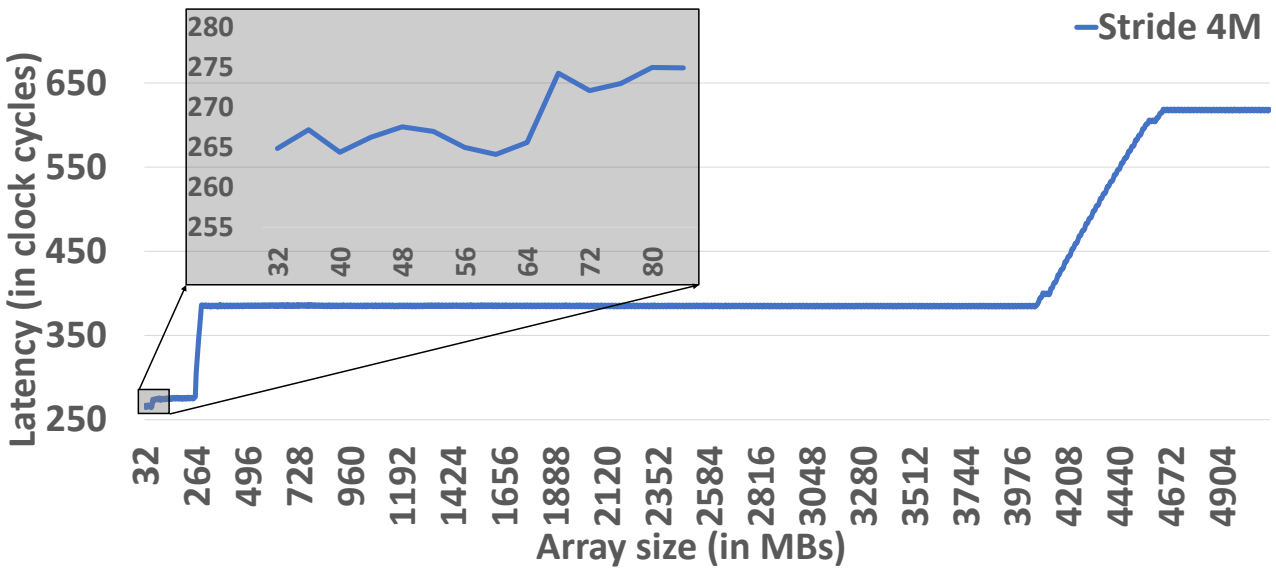


Figure 4.1: Observing multiple TLB levels using `cudaMallocManaged`. This graph shows our results from the pointer-chasing experiment with increasing `size`. We conclude that there are 3 levels in the TLB hierarchy due to 3 "knees" being observed.

Figure 4.1 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. As long as 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 hit 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). Note that, to the best of 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 amount of virtual memory mapped (translated) by each entry in a TLB. We need to explore each level separately since the characteristics of each level are different. We thus focus on the three steps in the latency plot of Figure 4.1 to decipher the details of each level. To infer reach and entry size, we need to vary both the `size` of the array and the `stride` (in powers of 2) in the pointer-chase algorithm. The reach of structures such as TLBs can be calculated as a product of virtual memory region mapped by each entry, and the number of such entries. By varying the `stride`, we try to occupy different number of entries in the TLB. For example, if the `stride` is smaller than the virtual memory region mapped by a single entry, two neighbouring access in our pattern can be served by only one TLB entry. When the `stride` is larger, two different

TLB entries are required. Thus, by varying the `stride` and the number of addresses accessed, *i.e.,* the `size` of the array, we can infer the TLB reach and entry size.

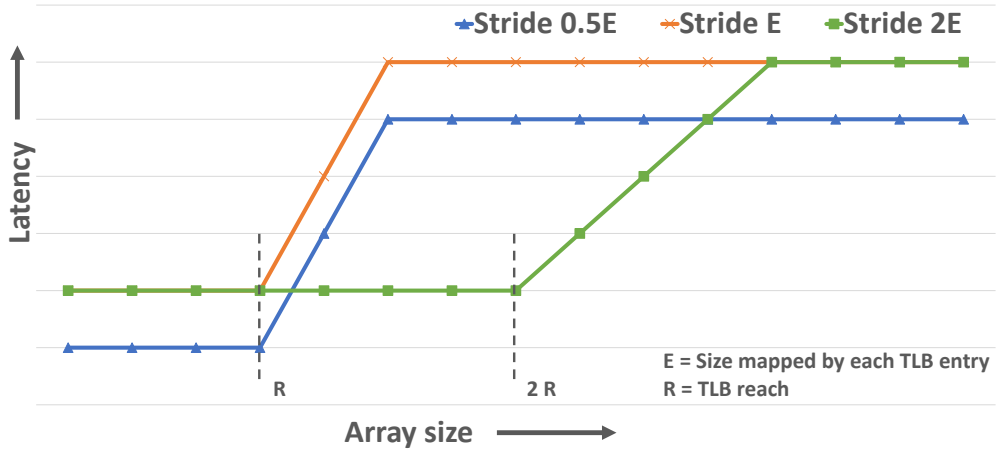
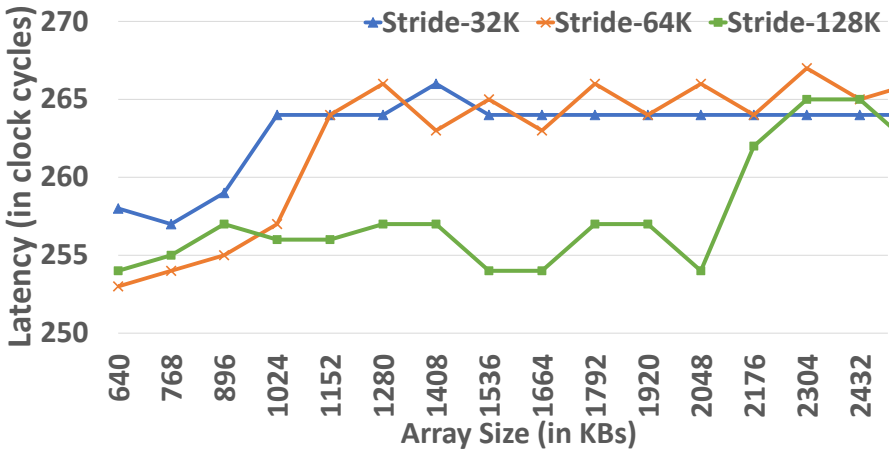


Figure 4.2: Inferring TLB reach and memory mapped by each TLB entry. This graph illustrates the pattern we need to obtain to decipher the memory mapped by each entry in the TLB and the corresponding reach.

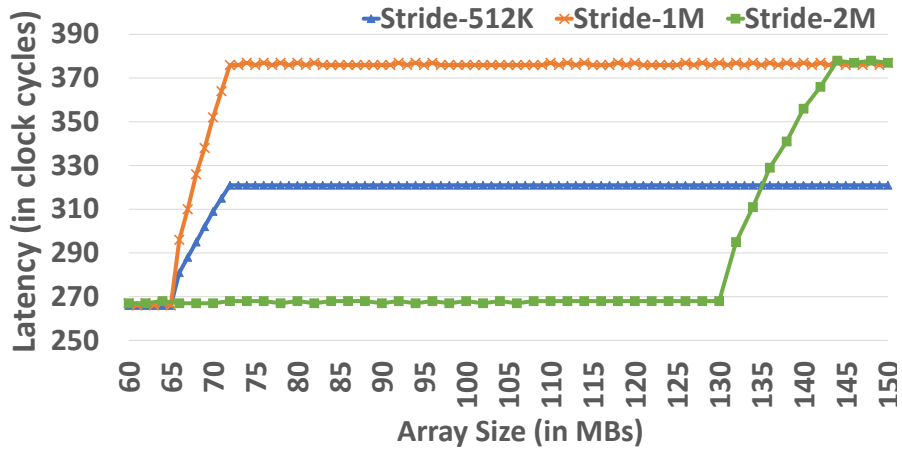
When the stride is smaller than or equal to the amount of memory that a TLB entry maps, we expect to observe a noticeable increase in the average access latency as soon as the array size overshoots the TLB reach (*i.e.,* the start of a "knee"). This is because, until that point, we expect hits in the 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 that a TLB entry maps, we need to double the array size to observe the knee in the latency plot.

Figure 4.2 illustrates the pattern we need to look out for with changing `stride` and `size` values to infer the memory mapped by each entry. The entry size is closely related to the `stride` argument, as mentioned before. We look for the pattern in the latency graph for each level where we observe a sharp rise in latency (knee) up to a given stride (E), and the knee shifts right in the next stride (twice that of the previous) *i.e.,* $2E$. 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 (R) at the given level. The largest stride at which the first knee starts indicates the amount of memory mapped by a TLB entry.

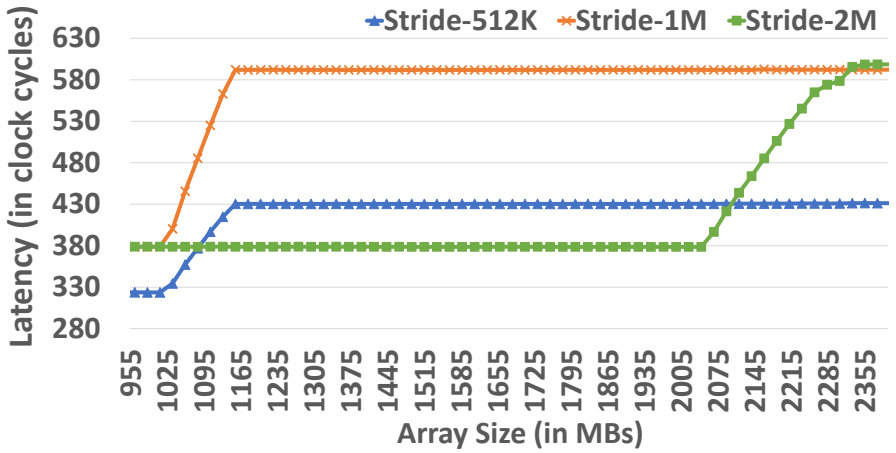
In Figure 4.3a, the first knee starts around 1MB array size (x-axis) and second knee around 2MB. Thus, the TLB reach for L1 TLB is 1MB. The second knee is observed with 128KB stride, while the first knee ends at the 64KB stride. Thus, each entry in L1 TLB maps 64KB. Similarly, from



(a) Observing L1 TLB.



(b) Observing L2 TLB.



(c) Observing L3 TLB.

Figure 4.3: Discerning each level of the TLB hierarchy with the pointer-chase algorithm using `cudaMallocManaged`. These graphs show how we deciphered the memory mapped by an entry in each level of the TLB hierarchy and their respective reaches.

Figure 4.3b and Figure 4.3c, we observe that the knee starts at 65MB and 1025MB, respectively.[1] Thus, the reach of the L2 and the L3 TLB are 65MB and 1025MB, respectively. Further, individual TLB entries in L2 and L3 map 1MB of memory, as the first knee disappears after 1MB stride.

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 4.3b and 4.3c). 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 as each set observes a capacity miss. 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 [58]. In the latency plot of L1 TLB (Figure 4.3a), 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, to answer our questions, we use the idea of eviction sets [54] captured during each run of the pointer-chasing algorithm (Snippet 2). An eviction set is a group of virtual addresses that index to the same set (row) of a TLB. We gather eviction sets during each run of our pointer-chasing benchmark. Suppose that the benchmark accesses a new address, and we subsequently observe that the access-time of a previously-accessed address has increased. This means that the entry for the new address evicted the entry corresponding to the previously-accessed address, and the two addresses are from the same eviction set. As the benchmark runs, the number of these previously-accessed addresses that are evicted also suggests the associativity of the set, as these are a consequence of a conflict or a capacity miss.

Previous studies have suggested a set-associative structure for the L2 TLB with uneven set sizes, *i.e.,* 1 set with 17 entries and 6 sets with 8 entries each [30, 21]. Our observations are consistent with these prior studies, and we observe one eviction set of size 17, and 6 eviction sets of size 8. This is further supported by observations in Section 4.1 that the L2 TLB's reach is 65MB with each

[1] While it may be hard to visually distinguish between 64MB and 65MB or between 1024MB and 1025MB in the plot; we verified it in the raw data.

entry mapping 1MB memory, *i.e.,* there are 65 entries. 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 *the illusion of an eviction set of size 17 is created by a victim entry that stores most recently-evicted TLB entries.* In spirit, this is reminiscent of the concept of a victim cache [34].

4.2.1 Extended pointer chase

We probe the hypothesis of the existence of victim cache by running an *extended pointer-chasing* experiment. The extended pointer chase experiment aims to overcome the odd observations made by the pointer-chase algorithm (Snippet 1), possibly due to the victim cache [34], and find true *eviction sets*. In this experiment, we first allocate a large virtual address space, and run the pointer-chasing algorithm from Snippet 2 multiple times, albeit at different starting addresses in each iteration of the algorithm. Specifically, we set the value of `REPEAT` in Snippet 1 to a value greater than 1.

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.

4.2.2 XOR-based function

We can further support the observation of victim caching once we find the function that indexes VPNs to TLB sets. Suppose that the TLB uses `mod`-based indexing. When the pointer-chasing microbenchmark fills the TLB up to capacity, and then subsequently accesses new virtual address, this access will evict entries from the TLB set to which it indexes. When we iterate the microbenchmark by increasing the array's size further, it will access more new addresses, in turn evicting existing entries from all the sets, one by one. In `mod`-based indexing, with `stride` equal to the TLB entry size, the sets overflow in the order in which they were populated. Therefore, we should observe a uniform pattern in the increase of access latency, *i.e.,* the entries populated in the 'oldest' set observe longer access latency first, followed by the entries in the next oldest set. However, we observed that, in many cases, the sets which were populated later (by entries added later to the pattern) were overflowing first. This observation leads us to conclude that the TLB does *not use `mod`-based indexing*.

Our conclusion is further supported by a second observation. In `mod`-based indexing, consecutive page numbers index into adjacent sets. Thus, if the `stride` parameter is increased to two-times the amount of memory mapped by a TLB entry in a `mod`-based indexed TLB, the perceived coverage of the TLB must not change (although the TLB itself may be underutilized and not filled to capacity). However, recall from Section 4.1 and Figure 4.3b that we in fact observe *twice* the coverage when the stride is twice the memory mapped by a single TLB entry.

Suppose we consider a page size of 1MB, given that each TLB entry maps 1MB of memory. The VPN then consists of 29-bits, given the 49-bit virtual addresses in Pascal. One possibility is that the hardware somehow utilizes all 29 bits in a hash function to determine the TLB set to which the page number indexes. However, we believe that it is very unlikely for the hardware to use such a generic hash function. A recent study by Gras *et al.* [10] on Intel CPUs suggested that the hardware combines a small number of bits using $\oplus$ (exclusive or). Other studies have also suggested the presence of $\oplus$-based indexing functions for different hardware structures [29, 13, 24]. We assume that the indexing function used by our platform does the same.

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 [47] 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.

```
Group1 (0, 1, 2, 3) = 11111111111010010 0--1--1--
Group2 (4, 5, 6, 7) = 11111111111010010 0--1--0--
Group3 (0, 1, 4, 5) = 11111111111010010 -0--1--1-
Group4 (0, 2, 4, 6) = 11111111111010010 --0--0--0
```

Before we group the eviction sets, we label them $[0-7]$ for an 8-way set associative structure (the "merged" eviction sets formed earlier in Section 4.2.1). Above we list a *minterm* from each group obtained from their SOP function. We form $\texttt{Group}_1$ from the union of all the addresses in eviction sets $(0-3)$. Similarly, we form $\texttt{Group}_2$ from the remaining eviction sets. From the listing, 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 $\texttt{Group}_1$ or $\texttt{Group}_2$. We uniquely identify each set by forming more distinct groups and finding values corresponding to the remaining bit-positions. In the above listing, we can uniquely identify set-0 using $\texttt{Group}_1$, $\texttt{Group}_3$, and $\texttt{Group}_4$ along with $\oplus$-values at different bit positions.

We use the above observations for constructing the TLB indexing function. 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 used and the algorithm developed can be found in Appendix B. The function in Figure 4.4a cleanly indexes the eviction sets into 8 sets (the set number is $o_2o_1o_0$, and i_n is the n^{th} bit of the virtual address). We name this function XOR-3.

We used the indexing function in Figure 4.4a to verify our hypothesis on the presence of a victim entry. We reran the pointer-chasing benchmark, using the knowledge of the indexing function to access exactly eight addresses that index to each of the 8 TLB sets and one additional address. The prior accesses must have filled up the TLB sets, and the access to the additional address must, therefore, evict an entry. Suppose our hypothesis about the existence of the victim entry is correct. In that case,

the access to this page must also appear to be an L2 TLB hit. Indeed, this is what we observed. When 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 4.4b. We leave the details of analyzing $\oplus$-based indexing function in Appendix A.

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

(a) The function used to index into the L2 TLB.

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

(b) The function used to index into the L3 TLB.

Figure 4.4: Functions to index into the TLB on using `cudaMallocManaged`. These figures show the virtual address bits and their corresponding use in computing the indexing functions of the L2 and the L3 TLB. Here, o_n is the n^{th} bit in binary representation of set number, and i_n is the n^{th} bit of the virtual address.

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 [40] 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 in 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)* [45]. 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 [21].

We design an experiment by again modifying the pointer-chase algorithm of Snippet 1 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 4.5. 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`. The last entry in this chain, instead of returning to the starting point to create a circular pattern, 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 4.5), 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 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 21^{st} 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, 4).

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 polices (*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. The 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

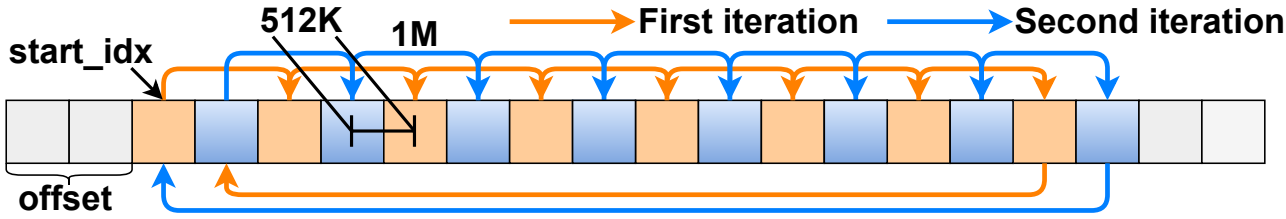


Figure 4.5: Pointer chase used to verify the presence of CoLT-like mechanisms. This figure illustrates the pointer-chasing pattern we need to create for distinguishing between a prefetch-based TLB and a TLB with coalesced entries.

```
# define ENTRIES 65
# define OFFSET 256KB
{
    /* start pointer chase at an offset, not
       the 0th index. */
    start_idx = 0 + OFFSET
    i = 1, temp = start_idx;
    /* create ENTRIES number of accesses */
    while (i < ENTRIES) {
        arr[temp] = temp + stride;
        temp = temp + stride;
        i++;
    }
    /* Point back to start_idx + OFFSET */
    arr[temp] = start_idx + OFFSET;
    /* Move at some offset */
    i = 1, temp = start_idx + OFFSET;
    while (i < ENTRIES) {
        arr[temp] = temp + stride;
        temp = temp + stride;
        i++;
    }
    arr[temp] = start_idx;
}
```

Snippet 3: Generating a pointer chase to verify CoLT.

couple of ways. First, we make it access a set of virtual addresses such that they cover the entire reach of 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 [21].

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 L1 and L2 entries. This is critical to ensure that we can create a covert channel with optimal bandwidth.

If the GPU had an inclusive allocation policy, then copies of the L1 TLB 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 TLBs (Section 4.2). We created a pointer-chase similar to Snippet 1, 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 (Snippet 2). 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 the removal of an entry from the L3 TLB would invalidate the corresponding entry from L2 as well. 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 (Snippet 4 shows the pseudocode). We then measured the time that each block took to access the addresses. The number of entries in this pattern are 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

```
barrier = 0 /* Initialize to zero */
{
  /* Thread block 1 performs pointer-chase */
  atomicAdd (barrier, 1);
  /* Block 0 returns */
}
{
  /* Thread blocks 2-N wait for barrier signal */
  while (atomicAdd (barrier, 0) != 1);
  /* Perform pointer-chase */
}
```

Snippet 4: Synchronization among blocks.

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.

Impact of TLB replacement policy on the reverse-engineering methodology. At the beginning of this chapter, we assumed that the replacement policy used in the TLB hierarchy is LRU or an approximation of LRU (say, Tree-LRU). Such policies ensure *determinism* in the observations made across all the runs of our experiments, making reverse-engineering feasible. The determinism here lies in the choice of the eviction victim (the replacement policy) from the TLB (L1) or the TLB sets (L2 and L3). Randomized choice of eviction victims can severely hurt the observations made in consecutive runs of our experiments, preventing us from making conclusive remarks about the TLB hierarchy. Though such randomization can hurt our methodology, the evidence from our experiments being consistent across all the runs suggests that such randomization is not used.

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` (Figure 4.4a)	`XOR-7` (Figure 4.4b)
Sharing Nature	Per-SM	Among few SMs	Across all SMs

Table 4.2: Summary of Pascal TLB microarchitecture.

4. REVERSE ENGINEERING GPU'S TLB CONFIGURATION

Summary: Table 4.2 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. We discover idiosyncrasies such as the existence of victim entries and coalesced TLB entries in both L2, and L3 TLB. We also decipher the indexing function for L2 and L3 TLBs.

We also performed similar experiments using default memory allocation (Chapter 5). However, it yielded only partial information. For example, it was not possible to note the existence of the L3 TLB. Memory allocated using `cudaMalloc` typically uses 2MB and each L3 TLB entry coalesces 16 entries. Since we now know that there are 1025 entries in L3, one would need to allocate more than 32GB of memory for conducting experiments to discover L3. It is not possible to allocate more than 32GB using `cudaMalloc` since it is beyond the physical memory capacity of even high-end GPUs. In short, *UVM plays a crucial role in reverse-engineering the GPU's TLB hierarchy*.

Chapter 5

Observations with the cudaMalloc API

In this chapter, we show our observations for the experiments described in Chapter 4, using the original memory allocation API, *i.e.,* `cudaMalloc`. These observations justify the importance of UVM in the formation of the channel. We use `cudaMalloc` for memory allocation in this chapter unless specified otherwise. Specifically, we replace `cudaMallocManaged` in line 5 of Snippet 1 with `cudaMalloc`. We operate under the same assumptions to achieve the goals from Chapter 4.

5.1 TLB Organization

We run the pointer-chasing kernel varying `stride` and `size` arguments. Figure 5.1 shows the change in average access latency with increasing `size` arguments. One can note, there are two distinct steps in the plot. On comparing Figure 5.1 with Figure 4.1, we observe that the steps are at different places, specifically the second step. This observation leads us to believe that the usage of `cudaMalloc` increases the reach of the TLBs, possibly due to the use of a different page size. To verify our claim, we investigate the structures by focusing on each level separately.

Figure 5.2a shows that the first knee starts around 32MB array size (x-axis), and the second knee starts at 64MB. Thus, the reach for L1 TLB is 32MB. The second knee is observed with 4MB stride, while the first knee at 2MB stride. Thus, each entry in L1 TLB maps 2MB. Similarly, from Figure 5.2b, we observe that the knee starts at 2080MB. Therefore, the L2 TLB reach is 2080MB. Further, each entry in the L2 TLB maps 32MB of memory, as the first knee disappears after 32MB stride.

We observe that L1 TLB characteristics are similar to those observed while using UVM, *i.e.,* it is a 16 entry fully-associative structure. Similarly, L2 TLB seems to be a 65 entry set-associative structure with victim cache. Each entry maps a 32MB virtual memory region and accounts for a total of 2080MB (32MB $\times$ 65). As L1 and L2 TLB structures are similar to ones observed in Chapter 4, we believe that L3 TLB will follow suit. For L3 TLB to be visibly apparent, *i.e.,* the third "knee" in

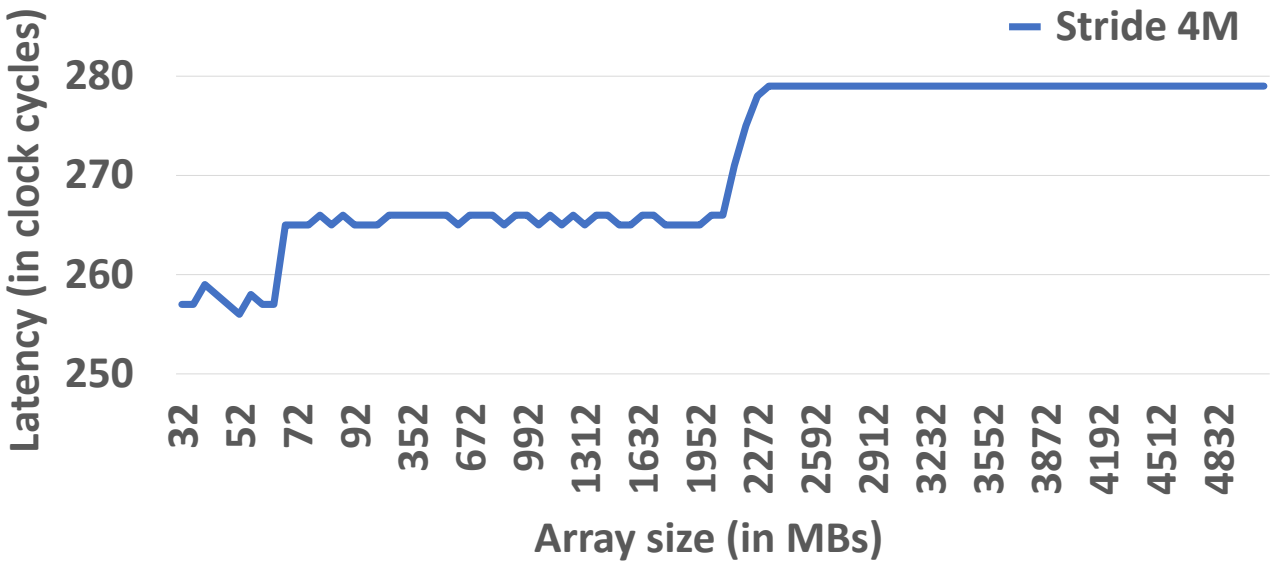
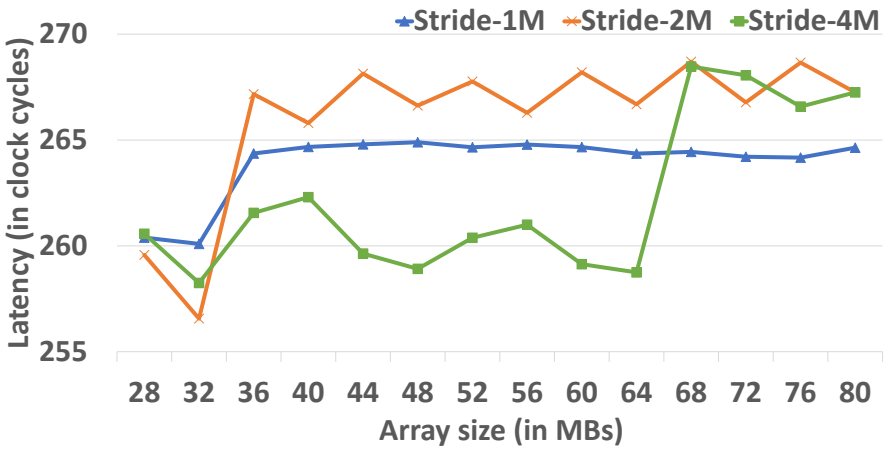


Figure 5.1: Observing multiple TLB levels using `cudaMalloc`. This graph shows our results from the pointer-chasing experiment with increasing `size`. We conclude that there are 2 levels in the TLB hierarchy due to 2 "knees" being observed. This graph illustrates the incompleteness of conclusions made in previous studies.

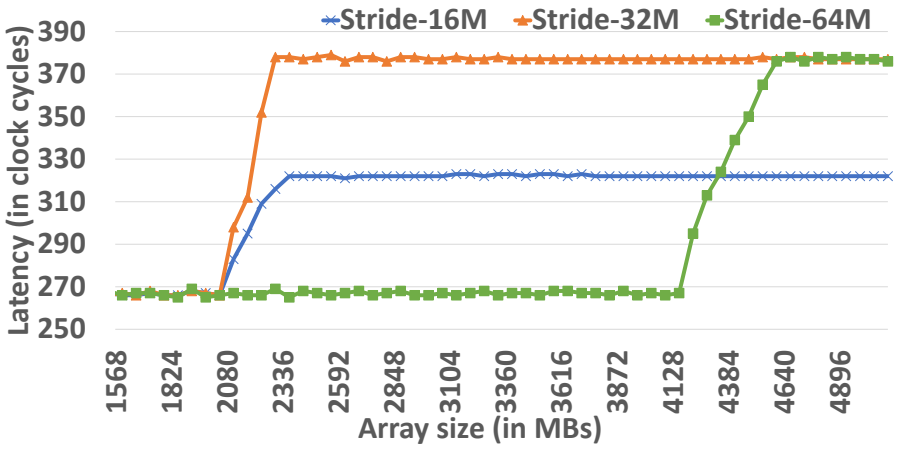
Figure 5.1, we need to access more than 1025 entries (Section 4.1). Each entry maps 32MB and 1025 such entries leads to a memory requirement of ~32GB (32MB $\times$ 1025).

Nvidia GPUs support three different page sizes, 4KB, 64KB and 2MB [40]. We believe that, on using `cudaMalloc`, the page size allocated by the memory allocator is 2MB as each entry in L1 TLB maps 2MB (Figure 5.2a). However, this does not answer why the observed entry size is 32MB in L2 TLB. By extending our analysis from Section 4.3 to `cudaMalloc`, we concluded that the illusion of 32MB memory mapped by a single TLB entry is due to CoLT-like mechanisms. Specifically, in L2, there are 16 contiguous address translations of 2MB each, mapped into a single TLB entry causing an illusion of 32MB entry. Thus, our analysis suggests the presence of CoLT-like mechanisms across the two supported page sizes. Supporting different page sizes requires different indexing functions into the TLB as the page-offsets are of different lengths. Using our knowledge from Section 4.2, we constructed the L2 TLB indexing function for 2MB page. Figure 5.3 cleanly indexes into the L2 TLB. Notice that the first bit present in the indexing function is the 25^{th} bit, which indicates that CoLT-like mechanisms are indeed present for 2MB page size.

With `cudaMalloc`, the driver pins the allocated memory pages on the device memory, *i.e.,* GPU DRAM. We have 11GB of physical memory on our GPU, with only high-end GPUs having more than 32GB of physical memory. As we cannot allocate more than available onboard memory with `cudaMalloc`, the availability of UVM is crucial to the channel. Specifically, the usage of UVM makes



(a) Observing L1 TLB.



(b) Observing L2 TLB.

Figure 5.2: Discerning each level of the TLB hierarchy with the pointer-chase algorithm using `cudaMalloc`. These graphs show how we deciphered the memory mapped by an entry in each level of the TLB hierarchy and their respective reaches.

the visibility of the L3 TLB at a much smaller memory footprint possible. As such, an attacker can exercise the channel even on GPUs with onboard physical memory as low as 3GB.

5.2 Supporting Multiple Page sizes

In this section, we perform experiments to understand how each level of TLB, discovered so far, support different page sizes, *i.e.,* the support of 64KB and 2MB page sizes (Sections 4.3 and 5.1).

CPU architectures that support different page sizes either have a unified or split TLB design. In the unified TLB approach, all the supported page sizes are present in the same hardware structure. A translation can happen sequentially (starting from the smallest page size) or in parallel before declaring a TLB miss. In the split TLB approach, there are separate hardware structures to cache

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

Figure 5.3: Function to index into the TLB on using `cudaMalloc`. This figure show the virtual address bits and their corresponding use in computing the indexing function of the L2 TLB. Here, o_n is the n^{th} bit in the binary representation of set number, and i_n is the n^{th} bit of the virtual address.

translations of different page sizes. These structures are checked for translation in parallel. Intel CPUs follow a hybrid approach, where lower levels (L1) follow split design, *i.e.,* separate structures for 4KB, 2MB, and 1GB pages. The last level, *i.e.,* L2 has a unified structure for 4KB and 2MB pages; 1GB still being a separate structure.

```
for i in range (0, PROBES * iter) {
    old_j = j;
    start = clock ();
    /* managed is allocated with cudaMallocManaged */
    j = managed[j];
    sum += j;
    managed[old_j + 1] = clock () - start;

    old_k = k;
    start = clock ();
    /* malloc is allocated with cudaMalloc */
    k = malloc[j];
    sum += k;
    malloc[old_k + 1] = clock () - start;
}
```

Snippet 5: Test multiple page size support.

We conduct another experiment to understand the structure of TLB for different page sizes. The experiment consists of the same pointer-chasing pattern. However, it accesses memory addresses backed by different page sizes alternately. Snippet 5 shows the pseudocode for the experiment. Note that we perform memory accesses from two different arrays in the `for` loop compared to only one in Snippet 2. The array `managed` is allocated using the `cudaMallocManaged` API and the array `malloc` is allocated using the `cudaMalloc` API. From previous sections (Sections 4.3 and 5.1), we concluded that `cudaMallocManaged` uses 64KB page size and `cudaMalloc` uses 2MB page size. We vary both the arrays' size and observe changes in average access time while maintaining the same number of elements in the pointer chase pattern.

If the TLB entries for both the page sizes are cached in the same hardware structure, the effective

reach observed for both the arrays will be halved. For example, in an 8-way set-associative structure, 64KB and 2MB page sizes will occupy 4 entries each (assuming the TLB is populated one set after another) with our access pattern. A capacity/conflict miss can manifest due to access to a 2MB or a 64KB page (due to the victim entry). Say, `managed` array causes the miss, and it replaces an entry in the set that was occupied by `malloc` array. Being a cyclic access pattern, consequent access from the `malloc` array will also cause a miss. These misses are observed at half the reach for the corresponding page sizes as the entries share the TLB structure *equally*. On tracking the average access time for increasing `size` values, we should observe the steps in Figure 5.1, albeit at smaller `size` values.

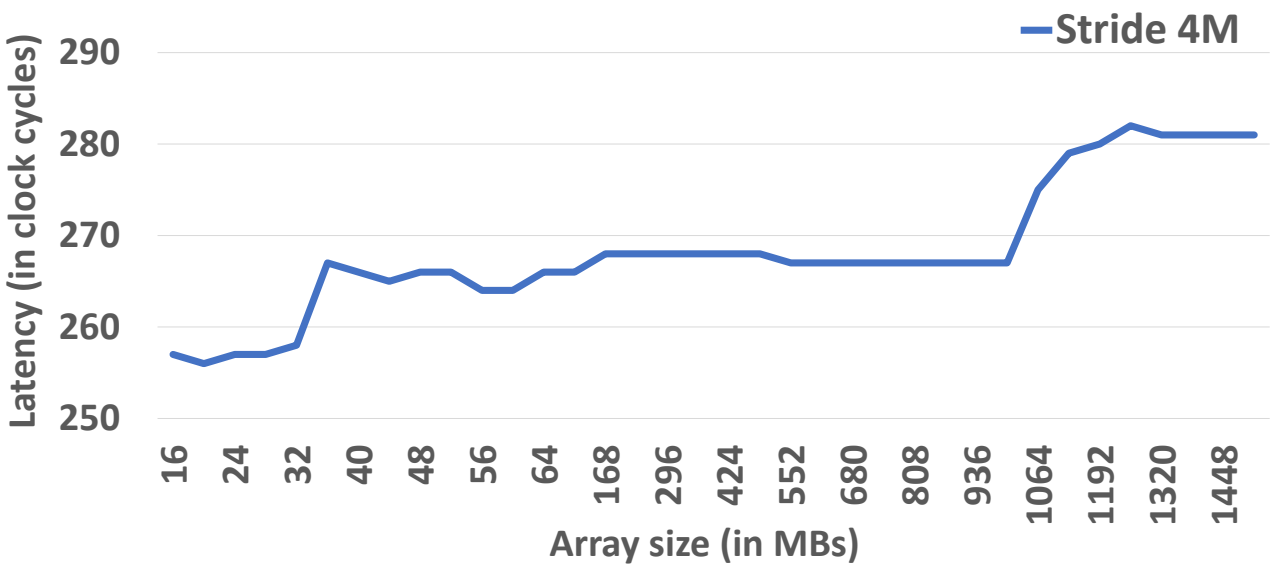


Figure 5.4: Levels observed at different `size` values for `cudaMalloc`. This graph illustrates the sharing nature of the TLB hardware for two different page sizes. A shift towards left is observed (compared to Figure 5.1), suggesting that a single hardware structure is shared by the TLB entries of different page sizes.

Indeed, we observed as described above. Figure 5.4 shows our observation for the experiment with varying `size` keeping the `stride` at 4MB. To keep the number of elements in the pattern the same, we traversed the array allocated with `cudaMallocManaged` with a `stride` of 128KB. In the figure, we observe two steps; however, the `size` where we observe the "knees" is 32MB and 1040MB for L1 and L2 TLB, respectively. Though not shown here, the `managed` array also reports a reduction in the reach, *i.e.,* half of the `size` reported in Figure 4.1. These observations suggest that a single hardware structure caches entries of both 64KB and 2MB page sizes at each level in the TLB hierarchy.

Chapter 6

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 channel's bandwidth and reduce bit errors. While forming the channel, we make the same assumptions described in Chapter 4. To reiterate: ① the TLB uses Least Recently Used (LRU) replacement policy (or some approximation thereof, such as Tree-LRU); ② the indexing function is static, *i.e.,* it does not vary across kernel runs.

6.1 A Minimal Channel

The `Trojan` kernel (Snippet 7) sends information to the `Spy` kernel (Snippet 8) using the shared 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.

```
/* The function performs pointer chase on a set. The function
   "prime" has similar function body, except it does  not need
   to time the accesses. */
__device__ unsigned long probe (set) {
    j = 0;
    start = clock ()
    for i in range (0, ITERATIONS):
        j = set[j]
    return clock () - start;
}
```

Snippet 6: Probe function.

We follow the popular prime+probe strategy [42, 33, 44] 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 a group of virtual addresses (VAs) that fall in chosen sets of the L3 TLB. The `Trojan` then executes

```
#define BITS_TO_SEND
#define LATENCY_THRESHOLD
__global__ void sender (int *arr, int *msg) {
    for j in range (0, BITS_TO_SEND) {
        /* Based on message, probe the information set */
        if (msg[j] == 0) {
            prime (CovertVAs_1)
        } else {
            /* Do nothing here */
        }
        /* Wake up receiver waiting for the signal */
        prime (CovertVAs_2)
        /* Wait for receiver signal */
        do {
            time = probe (CovertVAs_3)
        } while (time < LATENCY_THRESHOLD);
    }
}
```

Snippet 7: Sender kernel (`Trojan`).

```
#define BITS_TO_RECEIVE
#define LATENCY_THRESHOLD
__global__ void receiver (int *arr) {
    __shared__ short msg[BITS_TO_RECEIVE]
    for j in range (0, BITS_TO_RECEIVE) {
        /* Wait for sender to signal about the bit */
        do {
            time = probe (CovertVAs_2)
        } while (time < LATENCY_THRESHOLD);
        /* Probe information set and record the
           decoded message in shared memory */
        time = probe (CovertVAs_1)
        if (time > LATENCY_THRESHOLD)
            msg[j] = 1
        else
            msg[j] = 0
        /* Wake up sender waiting for the signal */
        prime (CovertVAs_3)
    }
}
```

Snippet 8: Receiver kernel (`Spy`).

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 6.1). The access latency for VAs evicted from the L3 TLB will correspond to that of L3 TLB

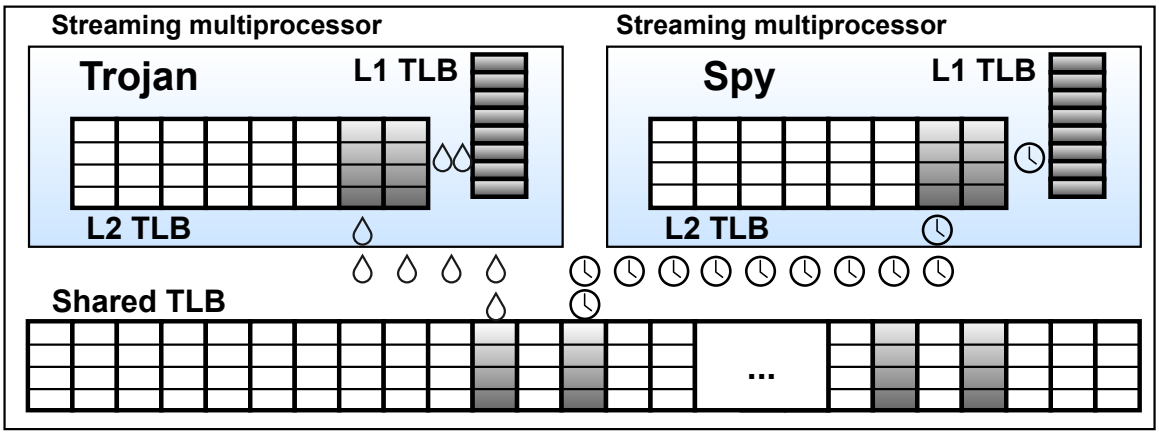


Figure 6.1: Covert channel using the shared L3 TLB.

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 Chapter 4 and design the `Trojan` and `Spy` to ensure that their accesses always reach the L3 TLB (Figure 6.1). 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 Chapter 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.

6. GPU COVERT CHANNEL VIA TLB

We use the indexing functions in Figure 4.4a and Figure 4.4b 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 are 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 are occupied in the TLB (here, 17). As GPUs utilize CoLT-like mechanisms; 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 as observed in Section 4.3). 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) [37], making the CovertVAs not usable as it's memory footprint increases. This increased memory footprint may not fit in the GPU memory.

Snippets 7 and 8 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 Snippet 7 shows 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` (Snippet 8), 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).

6.2 Parallelizing the Channel

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 (Snippets 7 and 8) run in parallel across all the SMs. More parallelism increases the bandwidth of the covert channel.

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.

6.3 Exploiting MPS to Enhance the Channel

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 [59], 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.

Forcing the `Trojan` and the `Spy` kernels to run on separate SMs, with MPS disabled does not improve the bandwidth. We can achieve this by launching dummy thread blocks that return immediately. For example, both the `Trojan` and the `Spy` launch two thread blocks (say block-0 and block-1). Block-0 of both the `Trojan` and the `Spy` run on SM-0 and block-1 on SM-1. block-0 of `Spy` and block-1 of `Trojan` return immediately, leaving one thread block each for the `Trojan` and `Spy` on separate SMs. However, we observed that this does not improve bandwidth. This observation leads us to believe that Pascal GPUs perform coarse-grained scheduling of thread blocks based on GPU context, irrespective of whether they are the only ones resident on the SM or not.

The problem of low bandwidth (a consequence of time-sharing) can be addressed using MPS. MPS allows kernels from different processes to execute on the same GPU concurrently (Section 2).

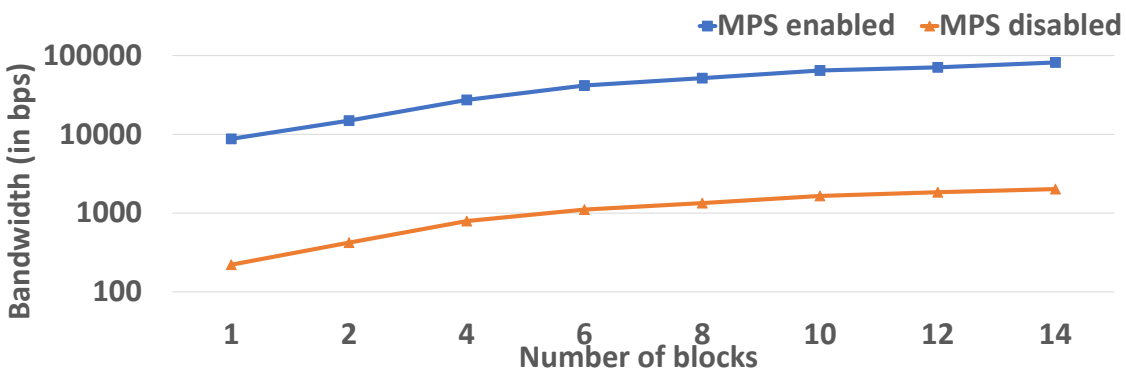


Figure 6.2: Maximum bandwidth of the covert channel. This graphs shows how the bandwidth of the channel benefits from parallelism, *i.e.,* occupying more SMs and spatial sharing, *i.e.,* MPS.

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×.

Noise from co-running applications? 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, a problem common in most of the covert-timing channels. In our channel, this noise occurs due to global memory accesses by these applications, possibly changing the L3 TLB state. These noise-inducing applications can run on SMs not occupied by the `Spy` and the `Trojan`. Such noise affects the accuracy and the bandwidth of the channel. MPS allows the launch of these applications, provided the GPU has sufficient resources (*e.g.,* SMs). To avoid noise, the `Spy` ideally needs to occupy these remaining SMs. The `Spy` can launch more dummy thread blocks than the number of remaining SMs. These thread blocks occupy SMs until the actual the `Spy` and the `Trojan` thread blocks finish execution (without adding noise). Dummy blocks can reserve all of shared memory (48KB on our GPU), preventing other thread blocks from being scheduled on those SMs. To synchronize with the actual `Spy` blocks, we can use synchronization mechanisms similar to one used in Section 4.4 (Snippet 4). No other application will co-run on the GPU once `Spy` and `Trojan` start, making the channel noise-free. Similar techniques have been suggested in the past [31].

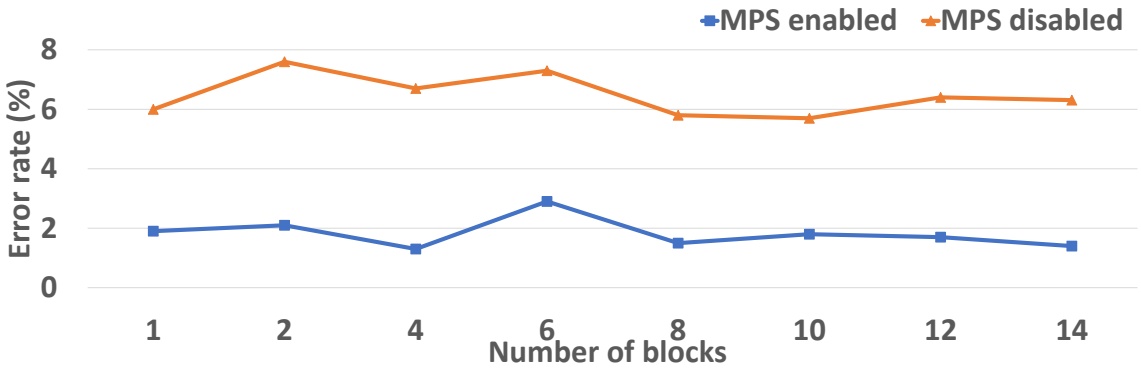


Figure 6.3: Average bit error rate in the covert channel. This graph demonstrates how the error rate of the channel is low due to spatial sharing of GPU between the `Trojan` and the `Spy`.

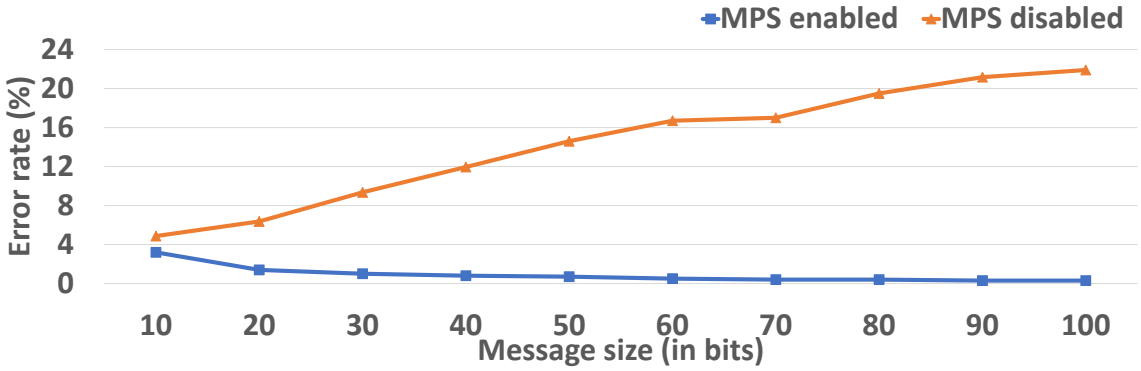


Figure 6.4: Average bit error rate in the covert channel for different message sizes. This graph shows how the channel benefits with MPS, maintaining low error rate across longer message sizes, compared to MPS disabled which shows increase in error rate with message size.

6.4 Channel Measurements

We exercise the channel with varying number of thread blocks, 20-bit long message per block, both with and without MPS. We present the maximum observed bandwidth of data transmission (excluding any bit errors) by exercising the channel 100 times for each thread block count in Figure 6.2. Please note the logarithmic y-axis in the figure.

We observe that the bandwidth of the channel without MPS goes from a couple of hundred bits-per-second to over two thousand bits-per-second. Importantly, when we enable MPS, the bandwidth increases by 40× to 81Kbps since both `Trojan` and `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 6.3 presents the measurements with varying number of thread blocks, with and without MPS. We observe that without MPS, the channel has a higher error rate. However, using MPS reduces the error rate. In other words, without MPS, the channel is noisy to be

useful. Thus, MPS reduces error rate and boosts the bandwidth of the channel, significantly.

For the channel to be useful, it should be able to communicate long messages with 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 6.4 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 error rate for long messages as well, an expected result.

Impact of TLB replacement policy on the covert channel. In this chapter, we formed a covert channel by constructing minimal eviction sets, *i.e.,* CovertVAs. The usability of CovertVAs and the observed error rate in the channel highly depend on the replacement policy used in the TLB. If the TLB replacement policy was not deterministic (Chapter 4), the channel will become highly unreliable. For example, a random replacement policy will deeply hurt the formation of the channel. To form the channel in NINE caching policy, every memory access needs to reach the level where the channel is formed, here L3. In the presence of a random replacement policy, this requirement cannot be ensured. However, the empirical evidence from this chapter and Chapter 4 suggests that such a random replacement policy is not implemented on the GPU we evaluated on.

Chapter 7

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], which is 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` (Snippet 7) as another kernel.

The goal of the (malicious) library is 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 `Trojan` *i.e.,* the database library, to be invoked. When the `Trojan` runs, the synchronization primitives presented 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.

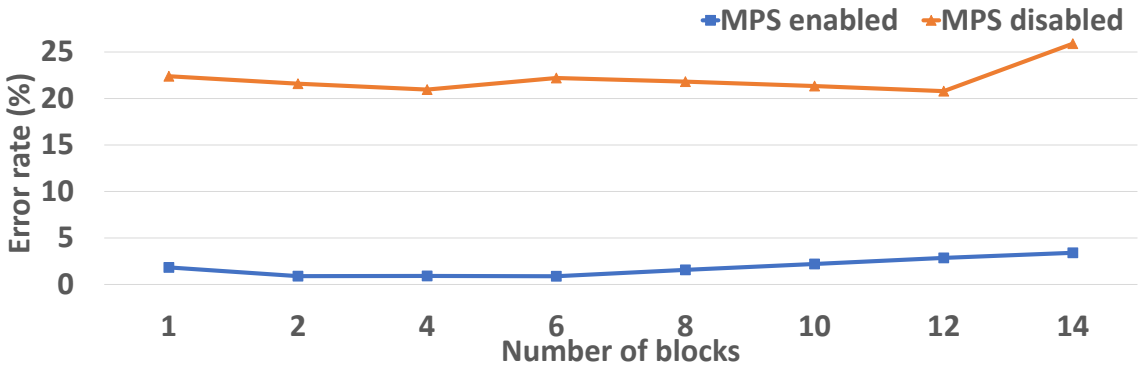


Figure 7.1: Average bit error rate in the covert channel used in a GPU accelerated database application.

Figure 7.1 shows the error rate we observed when the database library was used to leak information with varying number of blocks while performing 1000 rows of insert operation into the database. Here, when MPS is enabled, the channel has low error rate as well.

Why is the channel important? To our knowledge, there are no known defenses for this novel GPU TLB covert channel. The reader should note that CPU-based covert channels such as flush+reload [62] or other GPU-based channels [31] offer higher bandwidth than our channel. However, many proposed mechanisms defend against these attacks well [51, 59]. Our channel being novel, can still be exploited on systems that defend against the aforementioned faster channels. Most prior work on defenses for timing channel attacks on CPUs are based on techniques built using physical addresses [51]. 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 [59]. 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, on detection of an inter-SM channel, they fall back to temporal sharing, which will hurt GPU utilization and performance (by upto 2×). Recent architectural advancements such as Multi-instance GPUs (MIG) [41], provide hardware support for "isolated" sharing of GPU across applications by exposing upto 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.

Attackers would be interested in using a channel that is easier to mount. Another inter-SM channel that can be mounted on a GPU is an L2 cache channel, as demonstrated by Naghibijouybari *et al.* [31]. However, the reliability of such a channel depends on the knowledge of the physical address scheme, which user programs cannot control. In contrast, our channel relies on virtual addresses, which user programs can control.

Chapter 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 microarchitecure was performed on GT200 GPU, specifically, the early Tesla microarchitecture [58, 43]. More recently, Mei *et al.* [30] 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.* [21] studied the TLB hierarchy of Nvidia Pascal and Kepler microarchitectures. In this thesis, 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.* [14, 15] 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.* [23] were one of the first to identify security concerns with GPUs. Their study exposed concerns regarding GPU memory management, such 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.* [46] demonstrated similar leaks targeting shared memory, global memory, and register state. Naghibijouybari *et al.* [31] 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. Dutta *et al.* [7] demonstrated covert channels using resources shared between the CPU and an integrated GPU.

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.* [26] demonstrated a power side-channel attack by sampling and processing power traces on a GPU to extract secret keys of AES.

8. RELATED WORK

Jiang *et al.* [16] 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 [17] 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 [18]. Naghibijouybari *et al.* [32] demonstrated side-channels on GPUs using coarse grained metrics like memory utilization, performance counters, and timing. Wei *et al.* [56] trained multiple LSTM models to extract DNN model secrets on GPU. Luo *et al.* [27] constructed timing models of an RSA implementation on GPU and demonstrated a timing attack to extract the RSA private keys. Frigo *et al.* [8] compromised a browser running on a mobile phone by leveraging an integrated GPU.

Recent studies have demonstrated multiple mechanisms to defend against these side and covert channel attacks. To defend against side-channels targeting the memory coalescer, Kadam *et al.* [20] proposed `RCoal`, which introduces randomization to the coalescer. `Bcoal` [19] 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.* [59], propose `GPUGuard` to primarily mitigate intra-SM contention-based covert and side-channels. For inter-SM resources (such as L3 TLB), they fall back to temporal sharing, affecting GPU utilization. Hunt *et al.* [12] demonstrated a proof-of-concept side-channel attack on GPU TEEs by timing the communication of messages between 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.* [10] 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.

Chapter 9

Conclusion and Future Directions

In this thesis, we demonstrated the presence of a new covert channel that utilized GPU TLBs. To construct the covert channel we performed a detailed study of the TLB hierarchy present on a GPU, specifically understanding the organization as well as the indexing function used by the TLBs. We discovered the last level of TLB (L3) on a GPU within physical memory constraints which was never observed on latest GPU models (For example, Pascal microarchitecture). This was made possible due to a new functionality added to GPUs, meant to improve programmability, *i.e.,* UVM. We demonstrated a naive covert channel with the understanding of the TLB hierarchy and significantly improved it with the help parallelism, and another functionality meant to improve GPU efficiency *i.e.,* MPS. We showed the utility of the channel by leaking the contents of a database application.

Often, covert channels are an antecedent to a side-channel, given the two are closely related. Thus, the presence of a side-channel due to UVM characteristics is an interesting prospect. Especially, the interplay of `cudaMalloc` and UVM pages together in the system, on a victim kernel that uses only UVM pages needs to be further investigated. In a side-channel, the victim kernel is an unmodified program. Most of the kernels use shared memory, constant memory only accessing the global memory at the beginning and towards the later end of the kernel. Such diverse memory usage leaves little global memory trace. One of the key challenges here is to find kernels that leave such a trace which can be exploited to get meaningful information at page size granularity.

Covert channels are inherently more difficult to defend against completely as compared to a side-channel. One of the key goals while defending against covert channels is to drastically reduce the bandwidth of the channel. The primary factors that enable the channel are the shared L3 TLB and spatial sharing of GPU by the applications. Spatial sharing is critical for GPU utilization, suggesting that MPS cannot be disabled. Thus, the shared L3 TLB has to be dealt with efficiently. The L3 TLB can be partitioned dynamically between applications using different mechanisms [51]. 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 [11, 28], *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 request 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* [63], leading to further performance degradation. We believe that finding an efficient and scalable way to defend against channels that use last-level microarchitectural structures is an interesting research aspect worth pursuing. A randomized replacement policy can be used at higher levels of the TLB hierarchy. Such randomized policy can not only deter reverse-engineering efforts, but also make the channel unreliable as discussed in Chapters 4, 6.

Appendices

A Analysis with a toy indexing function

To formulate the steps we need to take for constructing the indexing functions, we first understand the intricacies of a XOR-based indexing function similar in nature to the one proposed in a recent work by Gras *et al.* [10]. We look towards the eviction sets for this purpose. These sets are generated using the extended pointer-chasing algorithm presented in Section 4.2.1. However, in this section, we rely on a toy indexing function to generate eviction sets.

$$o_2 = i_2 \oplus i_5 \oplus i_8$$
$$o_1 = i_1 \oplus i_4 \oplus i_7$$
$$o_0 = i_0 \oplus i_3 \oplus i_6$$

Figure 10.1: A toy indexing function. Here, o_n is the n^{th} bit in the binary representation of set number, and i_n is the n^{th} bit of the virtual address.

We use Figure 10.1 as our indexing function. The function uses *three* adjacent tuples of size *three* (in bits). These tuples are $i_0 i_1 i_2$, $i_3 i_4 i_5$ and $i_6 i_7 i_8$, where i_n is the n^{th} bit in an address. The output of the function is $o_2 o_1 o_0$. We assume there are no page-offset bits in a generated address. This assumption does not impact the analysis, as the indexing function does not use page-offset bits. It uses 9 bits, and as such there are 512 possible values, indexing into 8 sets. For example, using the indexing function, an address, say $A = 100010001$, is indexed as 111 and belongs to set-7.

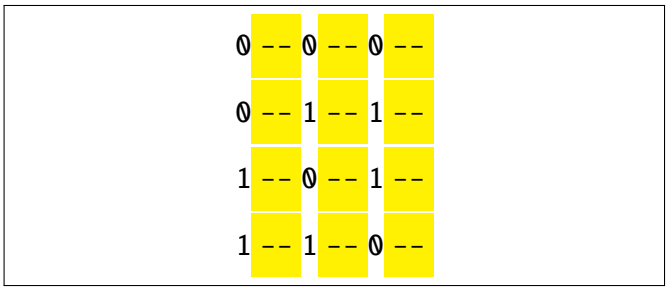
We use the eviction sets generated with the indexing function to understand how each bit contributes to the decision of an address in getting indexed to a specific TLB set. We feed these eviction sets to a Quine-McCluskey (QM) solver. The QM solver is used to create a minimized Boolean function for a given input. Previous studies have also used this method to approximate the indexing function. The solver provides a minimized Boolean expression in *Sum of Products (SOP)* form.

First, we determine the bits used to generate the indexing function. We feed the entries present in

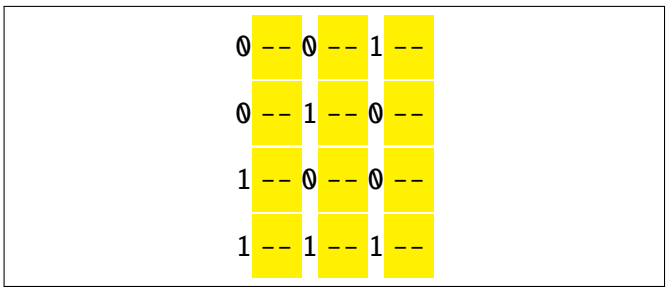
set-0 to the solver and check if the solver can form a minimized Boolean expression. This exercise helps us check if there are any bits that do not contribute to the expression, marked by '−' symbol by the solver, *i.e.,* the *don't-care* bits. The solver could not minimize it, *i.e.,* there were no *don't-care* bits in the expression. This observation suggests two possibilities, ① all the bits are getting utilized in the expression; ② The addresses for eliminating bits that do not contribute to the function are present in other sets. The observations made further in this section suggests that ② is the case.

To further investigate ②, we define a new function. We call this function *Merge(X)*, where *X* consists of a group of eviction sets. If *X* is formed from the union of sets labeled 0 and 1, we denote it as $X = \cup\{0, 1\}$. In this function, we take all the addresses present in the sets used to form *X* and feed it to the QM solver.

We divide the eviction sets into two disjoint sets of size 4. This kind of grouping helps to observe features that distinguish each group from a bit-value point of view. We observe *Merge(X)* where, $X = \cup\{0, 1, 2, 3\}$. The solver returns *minterms* in SOP form. The solver marks the bits which do not contribute in deciding if an address goes in sets used to form *X* with '−' (irrespective of the value of these bits, the address will be in *X*). We observe a common pattern in '−' across all *minterms*, as shown below (highlighted in yellow). The value of bits in non *don't-care* bit positions should determine if an address falls in *X* or not. This observation made us further investigate the scenario with the remaining sets.



Next, we compute *Merge(Y)* from the remaining sets, *i.e.,* $Y = \cup\{4, 5, 6, 7\}$. The *don't-care* bit positions were the same as for *X*, as seen below. This observation further strengthens the argument that non *don't-care* bit positions decide if an address is added to *X* or *Y*. To verify this argument, we compare the values of non *don't-care* bits and look for a pattern. Specifically, we compute the even-parity of the non *don't-care* bits, *i.e.,* even-parity of values at bit-positions [2, 5, 8]. The value was 0 for *X* and 1 for *Y* in their respective *minterms*. An even-parity is equivalent to a XOR operation. This is an expected result, as the addresses were generated using the toy indexing function (Figure 10.1).



The above exercise provided the value of bit positions `[2, 5, 8]` (numbered from right, starting with 0) that decides between the groups X and Y. Thus, the groupings of size 4 indeed provided a good insight on bit-positions and their corresponding values for the indexing function. To compute the value of the remaining bit positions, we need to try more combinations of sets to form X and, consequently, Y. We carry out the experiments and discuss the combinations which provided promising results.

X	Non *don't-care* bits	XOR-value	Output bit
∪{0, 2, 4, 6}	[0, 3, 6]	0	o_0
∪{0, 1, 4, 5}	[1, 4, 7]	0	o_1
∪{0, 1, 2, 3}	[2, 5, 8]	0	o_2

Table 10.1: Indexing into set-0. This table shows our observations regarding grouped sets, Non *don't-care* bits and the corresponding XOR values to uniquely identify set-0.

Table 10.1 summarizes the combinations used to find all the bit positions that affect the indexing function. Notice that the common element in X is 0 in all rows. Thus, for a given address, by computing the XOR of corresponding bits, we can decide if the address is indexed into set-0 or not. The XOR-value and position of corresponding bits are exactly what the toy indexing function (Figure 10.1) would generate for any address belonging to set-0. This observation gives us enough confidence in the analysis methodology.

We performed the above analysis for a single set, *i.e.,* set-0. However, by finding similar tuples for the remaining sets, we can verify the indexing function. Tables [10.2a, 10.2b, 10.2c, 10.2d, 10.2e, 10.2f, 10.2g] provide similar results for other sets. Note that in each table, only 1 set is common across all rows. We observe that the non *don't-care* bits are the same in each table; however, the XOR-value changes for each set. This observation gives substantial evidence that the analysis using *Merge(X)* provides good results when the indexing function is XOR-based. Thus, if a system uses XOR-based indexing function, we can construct the indexing function using the above methodology, *i.e.,* common set across rows, and non *don't-care* bit-positions with their corresponding XOR values.

There is a limitation to the number of addresses present in the eviction sets generated with the extended pointer-chasing algorithm. This limitations arises from the fact that `cudaMallocManaged` API returns a random address every time we allocate memory. As we cannot generate all possible

X	Non *don't-care* bits	XOR-value	Output bit
∪{1, 3, 5, 7}	[0, 3, 6]	1	o_0
∪{0, 1, 4, 5}	[1, 4, 7]	0	o_1
∪{0, 1, 2, 3}	[2, 5, 8]	0	o_2

(a) Indexing into set-1.

X	Non *don't-care* bits	XOR-value	Output bit
∪{0, 2, 4, 6}	[0, 3, 6]	0	o_0
∪{2, 3, 6, 7}	[1, 4, 7]	1	o_1
∪{0, 1, 2, 3}	[2, 5, 8]	0	o_2

(b) Indexing into set-2.

X	Non *don't-care* bits	XOR-value	Output bit
∪{1, 3, 5, 7}	[0, 3, 6]	1	o_0
∪{2, 3, 6, 7}	[1, 4, 7]	1	o_1
∪{0, 1, 2, 3}	[2, 5, 8]	0	o_2

(c) Indexing into set-3.

X	Non *don't-care* bits	XOR-value	Output bit
∪{0, 2, 4, 6}	[0, 3, 6]	0	o_0
∪{0, 1, 4, 5}	[1, 4, 7]	0	o_1
∪{4, 5, 6, 7}	[2, 5, 8]	1	o_2

(d) Indexing into set-4.

X	Non *don't-care* bits	XOR-value	Output bit
∪{1, 3, 5, 7}	[0, 3, 6]	1	o_0
∪{0, 1, 4, 5}	[1, 4, 7]	0	o_1
∪{4, 5, 6, 7}	[2, 5, 8]	1	o_2

(e) Indexing into set-5.

X	Non *don't-care* bits	XOR-value	Output bit
∪{0, 2, 4, 6}	[0, 3, 6]	0	o_0
∪{2, 3, 6, 7}	[1, 4, 7]	1	o_1
∪{4, 5, 6, 7}	[2, 5, 8]	1	o_2

(f) Indexing into set-6.

X	Non *don't-care* bits	XOR-value	Output bit
∪{1, 3, 5, 7}	[0, 3, 6]	1	o_0
∪{2, 3, 6, 7}	[1, 4, 7]	1	o_1
∪{4, 5, 6, 7}	[2, 5, 8]	1	o_2

(g) Indexing into set-7.

Table 10.2: Identifying the groups required the uniquely index all the sets. Each table has one common set, which can be uniquely identified by the corresponding groups and XOR values.

addresses, we can only form *incomplete* eviction sets. We need to simulate lack of addresses generated in one run of the extended pointer-chase experiment to mimic the scenario in original problem. To do that, we restrict our analysis to a small set of continuous addresses, say 64 (out of 512). In *Merge(X)*, when X is formed using the union of sets over all possible addresses, we label it as X, as discussed earlier in the section. In a restricted setting, *i.e.,* lack of addresses, we label it as X'. By computing *Merge(X')*, our goal is to observe the difference that lack of addresses can cause to the *don't-care* bit positions.

We compute *Merge(X')* such that, $X' = \cup\{0', 1', 2', 3'\}$. Note that this does not contain all possible addresses. We notice that the *don't-care* bit positions are at [0, 1, 3, 4]. On comparing them with earlier observation for *Merge(X)*, we see that bits 6 and 7 are missing from the list. This is true for all the X present in Table 10.1, as shown in Table 10.3. This suggests that lack of addresses make a difference in the observed *don't-care* positions. We use this observation to tackle the problem that arises in the original scenario *i.e., incomplete* eviction sets. Specifically, we assume that the pattern is incomplete due to absence of addresses in the eviction sets and extend the pattern of '−' till a certain bit-position threshold.

X'	Missing *don't-care* bit positions
$\cup\{0', 2', 4', 6'\}$	[7, 8]
$\cup\{0', 1', 4', 5'\}$	[6, 8]
$\cup\{0', 1', 2', 3'\}$	[6, 7]

Table 10.3: Analysis of set-0 with limited number of addresses. This table analyses the groups formed with incomplete eviction sets. It helps us understand the outcome of group-wise analysis with eviction sets captured in Section 4.2.1, and in constructing the indexing functions.

To summarize, there are certain differences in the analysis with the toy indexing function and the original problem. We list the differences, and how we handled them:

Learning 1. For a XOR-based indexing function, by constructing *two* distinct groups of size 4, we were able to cleanly differentiate between non *don't-care* bits and their corresponding XOR-values. This kind of grouping is in the case of 8 sets and will change based on the number of eviction sets.

Learning 2. Incomplete eviction sets in the sample restrict the creation of all *don't-care* bits. However, using the observations made from *Merge(X)* and *Merge(X')* scenarios, we can *extrapolate* the '−'-pattern till a certain threshold bit-position and empirically verify the number of adjacent tuples used in the indexing function over all the samples generated by the extended pointer-chase experiment.

B Uncovering the TLB Indexing Function

With our understanding from Appendix A on XOR-based indexing functions, we devise an algorithm to verify the presence of such an indexing function on our GPU. We make the following assumptions while using the algorithm in Snippet 9:

① The indexing function is XOR-based.

② The function uses all the bits between a range of bit positions in the virtual address.

```
/* generated using the extended pointer chase algorithm. */
Input : eviction_sets
/* a set per eviction_set, that tracks bits and corresponding XOR-value
   of those bits used to identify the eviction set. */
Output: For each set ∈ eviction_sets, a set {<bits[] :: XOR-value>}

per_set_map = {};
U = eviction_sets;
for set in eviction_sets {
    bits_identified = ϕ;
    /* We pre-decide "contributing_bits" as list of bit positions in virtual address that
       are used in indexing function. For example, contributing_bits = [20,40], says bits
       from [20-40] in virtual address are used in indexing function. */
    while(bits_identified != contributing_bits) {
        /* select_sets(a,b), selects "a" previously untried sets from among eviction_sets,
           excluding "b". */
        X ⇐ set ⋃ select_sets(3,set);
        /* Create Y from all sets not in X. */
        Y ⇐ U - X;

        merge_X ⇐ Merge(X);
        merge_Y ⇐ Merge(Y);

        /* get_dont_care(a) returns the bit position where the minterm passed as argument,
           "a" has don't-care bits. */
        if get_dont_care(merge_X) ≈ get_dont_care(merge_Y) {
            bits ⇐ get_non_dont_care_bits(merge_X);
            per_set_map.add(set, bits, get_xor(bits));
            bits_identified ⇐ bits_identified ⋃ bits;
        }
    }
}
return per_set_map
```

Snippet 9: Algorithm to verify XOR-based indexing function

The algorithm in Snippet 9 produces the output shown below for *each* set in the passed `eviction_sets`. Each line of output is of the form `<bits[] :: XOR-value>`, where `bits[]` is a list of bit positions in the virtual address and XOR-value is the XOR (⊕) of the corresponding bits. For example, in one of the

rows, the XOR-value is 1 for the bit positions corresponding to `[20, 23, 26, ...]` in the virtual address. This output helps in indexing virtual addresses into sets, using the corresponding bits and their respective XOR-values.

```
<[20, 23, 26,...]::1>
<[21, 24, 27,...]::0>
<[22, 25, 28,...]::1>
```

We now proceed to explain the algorithm in Snippet 9, taking suitable examples as and when necessary. For simplicity, we restrict the discussion to a single set (line 9) and a single iteration of the `while` loop at line 14. The goal of the algorithm is to generate bit-positions and their corresponding XOR-values for each set in the eviction sets generated by the extended pointer-chase algorithm (shown by the output above). To reiterate, each set in eviction sets consists of virtual addresses that map to a TLB set. The input to the algorithm is the said eviction sets (`eviction_sets`). For L2 TLB, we have 8 such sets. We show our analysis over L2 samples. By extending the same logic and empirically verifying over L3 samples, we constructed the L3 indexing function as well. In this section, we focus on L2, unless specified otherwise.

Initially, for each set, we have not identified any bit that contributes to the indexing function (line 10). We track the bits that contribute to the indexing function for a set with `bits_identified` list. As assumed, we keep track of all the bits that *might* contribute to the indexing function using `contributing_bits`. While analyzing the eviction sets, we had to fine-tune `contributing_bits` to consider *incomplete* eviction sets, *i.e.,* we do not have exhaustive information of virtual addresses mapping to TLB sets.

First, we pick z other sets from eviction sets for analysis and assign them to X. We use the method `select_sets(z,b)` to select z sets from U, excluding the set b. The method also ensures that the same tuple of z sets is not repeated twice, ensuring termination of analysis for a given set. We create Y using all the sets, not in X. The purpose of selecting $z+1$ sets is to create *two* disjoint groups of eviction sets. For analysis of L2 TLB, we choose $z = 3$ ($n = 2 \times z$; where n is the number of sets in the input). This kind of group creation helps us observe the bit-values that contribute to choosing a set among X or Y (Learning 1).

Next, we use an implementation of Quine-McCluskey (QM) [47] solver to get a minimized boolean expression for the addresses present in X and Y. The boolean expression consists of *minterms* in *Sum of Products (SOP)* form. The *minterms* mark bits that do not contribute to the expression with the *don't-care* symbol ('$-$'). We get all the addresses belonging to X using the `get_addresses` method. Next, we use an implementation of QM solver to form a minimized boolean function for these addresses using the `QM_solver` method. Shown below is the output for a given X from `QM_solver`, where

each line is a *minterm*. The presence of '−' leads us to conclude that, for a given X, the bits in positions marked with '−' decide if an address is mapped to sets used to form X. Rest of the bits determine if they fall in X or Y (as there are only two groups), with a specific function using these bit positions (assumed XOR). This observation also justifies why we formed *two* groups of size 4, *i.e.,* X and Y.

```
11111111111010010010 -- 1 -- 1
11111111111010010100 -- 0 -- 0
11111111111010010100 -- 1 -- 1
11111111111010010100-1 -- 0 -- 1
```

From the output, we see a particular recurring pattern in positions of '−' across all *minterm*s (highlighted in yellow). However, a keen reader will note that one *minterm* is having more*don't-care* symbols (last row). We believe such an inconsistency in the pattern is because of *incomplete* set of addresses present in the eviction sets. Provided a complete set of addresses, the pattern will be present across all *minterm*s, *i.e.,* all *minterm*s will have the same amount of '−'s.

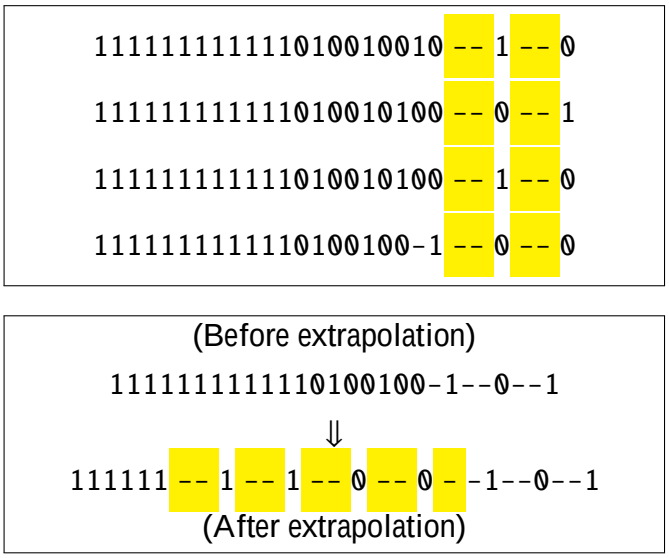
```
1  /* The function uses the QM solver and returns the minterm that captures
2     the recurring pattern of don't-care bits if present, ϕ otherwise. */
3  minterm_t Merge(group) {
4      addresses = get_addresses(group);
5      minterms = QM_solver(addresses);
6      return get_rep_mintern(minterms);
7  }
```

Snippet 10: An implementation of *Merge* function.

The method `get_rep_minterm` extracts a *minterm* with the recurring pattern as shown above. A poor combination used to form X may also generate *minterm*s. However, it will not have such a recurring pattern. At that point, the method will discard the combination by returning a ϕ to the calling method. The steps from using a QM solver to capturing the pattern in a single *minterm* is encapsulated in the method *Merge(group)* (Snippet 10).

Next, we follow similar steps to get the pattern capturing *minterm* for Y. For Y created from the sets, not in the same X, `QM_solver` gives the output shown below. From the output, we observe that the recurring pattern across *minterm*s is quite similar for both X and Y, *i.e.,* '−' were in the same bit positions. This further strengthens the argument that non *don't-care* bit positions decide if an address falls in X and Y. We check this condition in line 26 of the algorithm. This step is avoided when ϕ is returned from *Merge(group)*. Note the presence of $\approx$ symbol, denoting that though the pattern will be

similar; it might not be the same. As the non *don't-care* bits decide among *X* and *Y*, we extract these bits with the method `get_non_dont_care_bits`. This method not only captures the visibly apparent non *don't-care* bits, but additionally *extrapolates* the pattern till `contributing_bits` (Learning 2). We show how the method extrapolates from a pattern with an example.



We see that initially, the chosen *minterm* has fewer '−'s. The method extrapolates the '−'-pattern observed (seen at the right) and continues to build the pattern till it reaches the upper limit of `contributing_bits`. The '−' which were extrapolated are highlighted in yellow. The purpose of extrapolation of the pattern is due to *incomplete* eviction sets, as stated earlier. The extrapolation can be performed using a static pattern matching over a small size of the *minterm* (*e.g.,* 6 bits), configured initially. The extrapolation can also be driven by repeating the pattern in the initial $log(n)$ bits, where n is the number of sets in the structure. This step recurs the pattern of '−' over contiguous bits in the *minterm*. For simplicity, we extend the pattern observed in the first few bits and repeat it till we reach the upper limit of `contributing_bits`.

Finally, we form a XOR-based function from these bits, which are in non *don't-care* bit positions. These bits are then added to the currently identified bits (`bits_identified`) saved in a map that maintains this information for each set (`per_set_map`), and move to the next iteration. We repeat the iteration, until we identify all the bits for a set. Once the analysis is complete for *one* set, we move to the next set present in `eviction_sets`. When the algorithm terminates, we get the output shown earlier in this section for each set in `eviction_sets`.

Post analysis over all collected L2 samples, we observed that for each set in eviction set, the tuples in the left of the output *i.e.,* the `bits[]` that contribute to indexing function are in-fact same, however the value on the right *i.e.,* the XOR-value differs. This further strengthens the assumption that the function used for indexing is indeed a XOR-based function.

We constructed the indexing function for L2 using this observation. For each set in eviction sets, the algorithm generated *three* tuples. This output is an intuitive result and expected one, as to cleanly differentiate 8 sets, we need 3 bits ($log_2(8)$). The indexing function in Figure 4.4a cleanly indexes into the L2 TLB. We call it XOR-3. We observed preliminary results for L3 eviction set samples similar to L2. Using our observations from L2, we extended the hash function to accommodate 128 sets of L3. We call this function XOR-7 as in Figure 4.4b. Both the hash functions were tried and evaluated against all the generated samples and gave expected results. The validity of these hash functions is further supported with the covert channel construction, as discussed in Chapter 6.

References

[1] Amazon. P3 instances with v100, 2020. URL https://aws.amazon.com/ec2/instance-types/p3/. 1, 7

[2] AMD. Software optimization guide for amd family 17h models 30h and greater processors, 2019. URL https://www.amd.com/system/files/TechDocs/56305.zip. 2

[3] Peter Bakkum and Srimat Chakradhar. Efficient data management for gpu databases. 2012. 40

[4] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005. 1, 9

[5] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems*, 2006. 1, 9

[6] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure tlbs. In *Proceedings of the International Symposium on Computer Architecture*, 2019. 43

[7] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. *Computing Research Repository (CoRR), arXiv*, 2020. 42

[8] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the gpu. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018. 43

[9] Google. Cloud gpus, 2019. URL https://cloud.google.com/gpu/. 1, 7

[10] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security Symposium*, 2018. 1, 8, 19, 43, 46

[11] W. . Hu. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, 1991. 45

REFERENCES

[12] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure computing with cloud gpus. In *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020. 43

[13] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design*, 2015. 19

[14] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *Computing Research Repository (CoRR), arXiv*, 2018. 8, 42

[15] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing T4 GPU via microbenchmarking. *Computing Research Repository (CoRR), arXiv*, 2019. 8, 42

[16] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a gpu. In *International Symposium on High Performance Computer Architecture*, 2016. 1, 9, 43

[17] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A novel side-channel timing attack on gpus. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017. 1, 9, 43

[18] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. Exploiting bank conflict-based side-channel timing leakage of gpus. *ACM Transactions on Architecture and Code Optimization*, 2019. 43

[19] G. Kadam, D. Zhang, and A. Jog. Bcoal: Bucketing-based memory coalescing for efficient and secure gpus. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020. 43

[20] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. Rcoal: Mitigating gpu timing attack via subwarp-based randomized coalescing techniques. In *IEEE International Symposium on High Performance Computer Architecture*, 2018. 43

[21] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. Big data causing big (tlb) problems: taming random memory accesses on the gpu. In *International Workshop on Data Management on New Hardware*, 2017. 5, 8, 17, 22, 24, 42

[22] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. Ric: Relaxed inclusion caches for mitigating llc side-channel attacks. In *54th ACM/EDAC/IEEE Design Automation Conference*, 2017. 1

REFERENCES

[23] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2014. 42

[24] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 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*, 2020. 1, 7, 19

[25] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, 2015. 9

[26] C. Luo, Y. Fei, P. Luo, S. Mukherjee, and D. Kaeli. Side-channel power analysis of a gpu aes implementation. In *IEEE International Conference on Computer Design*, 2015. 42

[27] Chao Luo, Yunsi Fei, and David Kaeli. Side-channel timing attack of rsa on a gpu. *ACM Transactions on Architecture and Code Optimization*, 2019. 43

[28] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012. 45

[29] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien" Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions and Defenses*, 2015. 19

[30] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. In *IEEE Transactions on Parallel and Distributed Systems*, 2016. 17, 42

[31] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *IEEE/ACM International Symposium on Microarchitecture*, 2017. 1, 9, 10, 37, 41, 42

[32] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018. 10, 43

[33] Michael Neve, Seifert, and Jean-Pierre. Advances on access-driven cache attacks on AES. In *Selected Areas in Cryptography*, 2007. 2, 9, 32

[34] Jouppi Norman. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM SIGARCH Computer Architecture News*, 1990. 18

[35] Nvidia. CUDA C++ programming guide. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2020-07-20. 6

[36] Nvidia. Unified memory for cuda beginners, 2017. URL https://devblogs.nvidia.com/unified-memory-cuda-beginners/. 1, 6

[37] Nvidia. Maximizing unified memory performance in cuda, 2017. URL https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/. 35

[38] Nvidia. Gpus everywhere, 2019. URL https://blogs.nvidia.com/blog/2017/05/08/microsoft-azure-gpu-instances/. 1

[39] Nvidia. Documentation for multi-process service, 2019. URL https://docs.nvidia.com/deploy/mps/index.html. 1, 7

[40] Nvidia. Nvidia open-gpu-doc repository, 2019. URL https://github.com/NVIDIA/open-gpu-doc. 21, 28

[41] Nvidia. Nvidia multi-instance gpus, 2020. URL https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. 41

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of ”aes”. In *RSA Conference on Topics in Cryptology*, 2006. 1, 2, 9, 32

[43] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Microbenchmarking the gt200 gpu. Technical report, Computer Group, ECE, University of Toronto, 2009. 42

[44] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005. 1, 2, 32

[45] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *IEEE/ACM International Symposium on Microarchitecture*, 2012. 3, 21

[46] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. Cuda leaks: A detailed hack for cuda and a (partial) fix. In *ACM Transactions on Embedded Computing Systems*, 2016. 42

REFERENCES

[47] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 1952. 20, 52

[48] M. K. Qureshi. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019. 1

[49] R. H. Saavedra and A. J. Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Transactions on Computers*, 1995. 13

[50] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. Neighborhood-aware address translation for irregular gpu applications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018. 5

[51] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 2019. 7, 41, 44

[52] Ristenpart Thomas, Tromer Eran, Shacham Hovav, and Savage Stefan. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security*, 2009. 10

[53] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security Symposium*, 2015. 10

[54] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy (SP)*, 2019. 17

[55] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007. 1

[56] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020. 43

[57] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: Thwarting cache attacks via cache set randomization. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019. 1

REFERENCES

[58] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software*, 2010. 13, 17, 42

[59] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. Gpuguard: Mitigating contention based side and covert channel attacks on gpus. In *Proceedings of the ACM International Conference on Supercomputing*, 2019. 36, 41, 43

[60] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. 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)*, 2017. 1

[61] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019. 7, 9

[62] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014. 9, 41

[63] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016. 45