Cache Architecture and Security Challenges in New Age Processor Architectures

Project-II (CS47006) report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilment for the award of the degree of

Dual Degree (B. Tech + M. Tech)

in

Computer Science and Engineering

by

Sumit Kumar

(22CS30056)

Under the supervision of

Professor Sarani Bhattacharya

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Spring Semester, 2025-26

April 9, 2026

DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: April 9, 2026 (Sumit Kumar)

Place: Kharagpur (22CS30056)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

KHARAGPUR - 721302, INDIA

CERTIFICATE

This is to certify that the project report entitled "**Cache Architecture and Security Challenges in New Age Processor Architectures**" submitted by **Sumit Kumar** (Roll No. 22CS30056) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Dual Degree (B. Tech + M. Tech) in Computer Science and Engineering is a record of bona fide work carried out by him under my supervision and guidance during Spring Semester, 2025-26.

Date: April 9, 2026

Place: Kharagpur

Professor Sarani Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721302, India

Abstract

Name of the student: **Sumit Kumar** Roll No: **22CS30056**

Degree for which submitted: **Dual Degree (B. Tech + M. Tech)**

Department: **Department of Computer Science and Engineering**

Thesis title: **Cache Architecture and Security Challenges in New Age Processor Architectures**

Thesis supervisor: **Professor Sarani Bhattacharya**

Month and year of thesis submission: **April 9, 2026**

Modern computing systems increasingly rely on heterogeneous architectures where CPUs and GPUs share the same physical hardware and, critically, the same Last-Level Cache (LLC). This shared cache creates a side-channel through which one component can silently observe the memory access behaviour of another, violating process isolation. This thesis investigates a practical GPU-to-CPU cache side-channel attack based on the **Prime+Probe** technique, where the GPU acts as the attacker and CPU acts as the victim.

The attack uses CUDA Unified Virtual Memory and physical address translation via the Linux pagemap interface to target specific LLC sets, measuring re-access timing with GPU hardware clock counters to detect victim-induced cache evictions. Experiments across two GPU platforms show that GPU and CPU processes are not cache-isolated, and that a CPU workload's memory footprint can be leaked to a co-located GPU process purely through timing, highlighting a real security risk in multi-tenant heterogeneous computing environments.

Acknowledgements

I am deeply grateful to my advisor, **Professor Sarani Bhattacharya**, for her guidance, encouragement, and many insightful discussions throughout this work. I would also like to thank senior collaborators, **Arunava Chaudhuri** and **Shubhi Shukla** for their feedback, ideas and support during experimentation and analysis.

Contents

Chapter 1

Introduction and Motivation

1.1 The Modern Heterogeneous System

In most machines today, a CPU and a GPU sit on the same board and share the same physical memory. From the outside this looks like a clean split of responsibilities, with the CPU handling general computation and the GPU handling parallel workloads, but at the hardware level they are not as separate as we tend to think.

Both processors connect to the same DRAM. When the GPU needs to fetch data that is not in its own on-chip caches, that request goes through the PCIe bus and lands in the CPU's Last-Level Cache (LLC) before reaching main memory. The LLC was built to reduce DRAM latency for CPU programs, but because of how the memory bus is organised, it ends up handling the GPU's remote fetches as well. There is no hardware mechanism that keeps the two processors' LLC traffic separate. They use the same cache sets, at the same time, without knowing about each other.

CUDA Unified Virtual Memory (UVM) [1] makes this even more concrete. With UVM a single virtual address space is shared between the CPU and the GPU. A page that is resident in CPU DRAM is accessible to the GPU over PCIe, and every time the GPU fetches a cache line from that page it goes through the CPU's LLC. So both processors are not just sharing the same DRAM, they are contending for space in the same cache hierarchy on every access.

1.2 Why Shared Cache Sets Are a Problem

The LLC is set-associative, meaning each cache line at physical address P maps to a fixed set S given by:

$$S = \left\lfloor \frac{P}{64} \right\rfloor \bmod N_{\text{sets}} \tag{1.1}$$

where N_{sets} depends on the cache size and associativity. The important thing here is that this mapping depends only on the *physical* address, not the virtual one, and it applies to both the GPU and the CPU equally.

This means that a GPU process and a CPU process running at the same time can end up competing for the exact same LLC set, even though their virtual addresses are completely different and neither process can read the other's memory.

The consequence is straightforward to see. Suppose the GPU loads some data into LLC set S. Now the CPU runs and accesses its own data, which by coincidence also maps to set S. The set fills up and some of the GPU's cache lines get evicted to make room. Later, when the GPU tries to access that data again, it is no longer in the LLC. The GPU has to go all the way to DRAM, which takes several times longer than a cache hit would.

If the GPU times this re-access, it gets information it was not supposed to have: whether the CPU touched that particular set while the GPU was not looking. Fast re-access means the CPU did not disturb set S. Slow re-access means it did. By repeating this across many sets, the GPU can figure out which parts of memory the CPU was working with, without ever reading any of the CPU's data directly. This is the Prime+Probe side-channel attack, applied across the CPU-GPU boundary.

1.3 How This Project Came About

The idea for this work came from looking at what actually happens when a GPU and CPU run together on the same machine. When a GPU is doing some processing while a CPU is running a compute-heavy job on the side, both processors hit the LLC at the same time. The LLC has no concept of which processor owns which data. It just serves whoever asks.

The natural question was: can the GPU use this to learn something about what the CPU is doing? In a cloud environment, for instance, two different users might be sharing the same physical machine, one running a GPU workload and another running a CPU-side AI inference pipeline with sensitive model weights. If the GPU user can observe LLC timing differences caused by the CPU user, that is a real information leak, even if no memory is ever shared or read directly.

What made the attack practical rather than just theoretical was Linux's `/proc/self/pagemap` interface. This file exposes the physical frame number (PFN) of any virtual page, which means a process with root access can look up the exact physical address of its own memory. Given the physical address, the LLC set number follows directly from the formula above. So instead of guessing which sets the victim might be using, the GPU process can deliberately allocate CUDA Unified Virtual Memory pages whose physical addresses map to the same LLC sets as the victim's data, and then run Prime+Probe on those specific sets. The attack becomes targeted rather than a blind scan.

1.4 Scope of This Work

This thesis builds and evaluates a GPU-to-CPU Prime+Probe attack. The attacker is a CUDA process that uses CUDA Unified Virtual Memory and `/proc/self/pagemap` to target specific LLC sets, then measures probe timing with the GPU hardware clock counter (`clock64()`). The victim is a CPU thread running AI model inference with no knowledge of the attacker.

Experiments run on an NVIDIA RTX 3060 and an NVIDIA A30, with five AI models as victim workloads and a cross-GPU variant on the A30. The following chapters cover the background, implementation, and results.

Chapter 2

Literature Review

2.1 Cache Side-Channel Attacks

Cache timing as a way to leak information has been studied since the early 2000s. Osvik, Shamir, and Tromer [2] put the Prime+Probe technique on solid ground in 2006. The attacker fills a cache set with its own data, waits for the victim to run, then checks if that data is still there. A slow re-access means the victim evicted it, which tells the attacker that the victim used that set. They tested this on AES in OpenSSL and recovered the key. What makes Prime+Probe useful is that it only needs a shared cache, not shared memory pages, so it works across completely unrelated processes.

Yarom and Falkner [3] took a different route with Flush+Reload. Instead of filling a set, you flush a specific cache line using `clflush` and time how long it takes to reload. The signal is cleaner and the resolution is finer, but it requires the attacker and victim to share the same physical memory page. In our case the GPU and CPU do not share memory pages that way, so Flush+Reload does not directly apply. Prime+Probe is the right fit here.

Liu et al. [4] made the important step of taking Prime+Probe to the LLC and showing it works across separate cores and even across VM boundaries. Before that, most attacks were on L1 or required the attacker to be on the same core as the victim. Their 2015 paper is probably the most direct CPU-side predecessor to our

work. The hard part they had to solve was finding addresses in their own virtual space that map to the same LLC set as the victim's data, since physical addresses are not directly visible to userspace. They worked around this using huge pages, which give more predictable virtual-to-physical mappings. We use a more direct approach by reading `/proc/self/pagemap` to get the actual physical address, which lets us target any LLC set we want without relying on huge pages.

2.2 GPU Side-Channel Work

The GPU side-channel literature is much thinner than the CPU side. GPUs were not traditionally considered targets for adversarial code, but that has been changing as they become general-purpose compute platforms in shared environments.

Naghibijouybari et al. [5] did a fairly comprehensive study in 2018 showing that two CUDA processes running concurrently on the same GPU can spy on each other through shared performance counters and memory access timing. They could infer what model architecture a co-located process was running and sometimes what inputs it was processing. Their setting is GPU-to-GPU on a single device, which is not what we are doing, but the work established that GPU kernels can make meaningful timing observations about other processes sharing the same hardware.

Jiang et al. [6] showed something more specific, that fine-grained timing inside a GPU kernel can recover cryptographic keys from a co-located AES computation. Again this is within a single GPU, but it confirmed that `clock64()` style measurements inside a CUDA kernel are precise enough to distinguish cache hit from cache miss latencies. That is exactly the measurement we rely on during the probe phase.

2.3 Cross-Component and Heterogeneous Attacks

Very little work has looked at attacks that cross the CPU-GPU boundary. Most papers either stay entirely on the CPU side or entirely within one GPU.

The closest thing on the CPU side is Cache Telepathy [7] by Yan et al., which uses LLC side channels to reverse-engineer neural network architectures running on the CPU. The victim setup is very close to ours, a CPU process doing DNN inference, with model weights in DRAM flowing through the LLC during the forward pass. By watching which LLC sets get touched, the attacker can figure out the layer dimensions and structure of the network. In their case the attacker is another CPU process. We are asking whether the same kind of observation is possible from a GPU process instead, which is a meaningfully different scenario because the GPU accesses the LLC through PCIe rather than through the CPU's own memory controller.

The `/proc/self/pagemap` interface we use for physical address lookup has shown up in a different context in rowhammer research [8]. Seaborn and Dullien used it to find physically contiguous DRAM rows to trigger bit flips. We are using it for a completely different purpose, but the point is that reading physical addresses from userspace with root access is an established technique with known cost and reliability, which is why we adopted it.

2.4 Where This Work Fits

Looking at everything above, there is a fairly clear pattern. The CPU side-channel literature is mature, Prime+Probe, Flush+Reload, and LLC attacks across processes and VMs have all been well studied. The GPU side-channel literature is smaller but growing, mostly focused on same-GPU attacks where two CUDA processes share a device. What has not been studied is the direction we are interested in, a GPU process observing a CPU process through the shared LLC.

This gap exists partly because the scenario is less obvious at first. Why would an attacker on the GPU want to spy on the CPU? In a multi-tenant cloud setup it is quite realistic. One tenant's job runs on the GPU, another tenant runs a CPU-side inference pipeline with proprietary or sensitive model weights. The GPU tenant shares the same physical LLC with the CPU tenant simply because of how the memory bus is wired. There is no software policy that prevents this and the hardware does not isolate them either.

The specific combination we use, CUDA Unified Virtual Memory to get the GPU's managed pages into CPU DRAM so they share the LLC with the victim, pagemap to target specific LLC sets precisely, and `clock64()` timing inside the probe kernel, has not appeared in the literature before. Each individual piece exists in some form in prior work, but putting them together into a working GPU-to-CPU attack is what this thesis contributes.

One more thing worth noting is that the AI inference workload as a victim is a deliberate choice, not just a convenient one. Model weights for modern networks are large, stay in DRAM, and get accessed in structured patterns during the forward pass. This makes them a realistic target for cache-based leakage in exactly the kind of environment where GPU and CPU resources are shared. The Cache Telepathy paper showed that the LLC access pattern during inference is rich enough to reveal model structure. Our work asks whether that signal is still observable when the attacker moves to the GPU, and whether the per-set timing differences are distinct enough across different models to be practically useful.

Chapter 3

Background and Methodology

This chapter covers the technical background behind the attack: GPU memory architecture and address translation, LLC set addressing, CUDA Unified Virtual Memory, physical address translation via `/proc/self/pagemap`, and the Prime+Probe timing loop.

3.1 GPU Memory Architecture and Address Translation

A discrete GPU has its own device memory (GDDR/HBM) and its own Memory Management Unit called the GMMU. Each Streaming Multiprocessor (SM) has a private L1 TLB. On an L1 TLB miss the request is forwarded to a shared L2 TLB, and on an L2 TLB miss it enters a page walk queue. The GMMU's page table walker then walks the GPU's own page table, which is stored in device memory, to get the physical address.

GPUs use a multi-level Radix Page Table (RPT) as standard [1]. For a four-level RPT a single page walk makes up to four sequential memory accesses, each of which may miss the GPU's L2 cache. This adds significant latency and is one reason GPU workloads are sensitive to TLB miss rates.

Remote pages and CPU DRAM. The GPU's page table holds entries for pages that are currently in GPU device memory. Pages that live in CPU DRAM are called *remote pages*. The GPU driver maintains PTEs for these in the CPU's page table. When the GPU tries to access a remote page, the GMMU generates a page fault, and the GPU driver handles it by migrating the page from CPU DRAM to GPU memory over PCIe before resuming execution.

Figure 3.1 illustrates this. The GPU page table holds PTEs 0–3, pointing to Pages 0–3 in GPU memory. Pages 4 and 5 are remote: they live in CPU DRAM and their PTEs are held only on the CPU side. When the GPU accesses PTE 4 (step 1), it gets a page fault. The GPU driver walks the CPU page table (steps 2 and 5), migrates Page 4 from CPU DRAM to GPU memory (step 4), and updates the GPU page table (step 3) so future accesses find it locally.

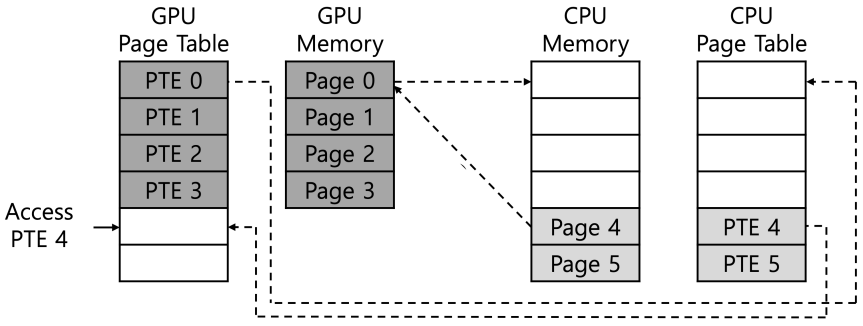

FIGURE 3.1: PTE eviction and allocation in GPUs [1]. When the GPU accesses a page not in GPU memory (PTE 4), the GMMU raises a fault, the driver walks the CPU page table, and the page migrates from CPU DRAM to GPU memory over PCIe. In our attack we prevent this migration so the GPU's accesses always go through the CPU LLC.

This migration mechanism is exactly what we subvert. In our attack we prefetch UVM pages back to CPU DRAM before every prime and probe phase, forcing the GPU to always access them as remote pages over PCIe. Every such access passes through the CPU's LLC, which is the shared resource the attack exploits.

3.2 The Last-Level Cache and Physical Address Indexing

The Last-Level Cache is a set-associative cache shared across all CPU cores. On the machines we used, the LLC sits between the CPU cores and main memory, and every memory access that misses the per-core L1 and L2 caches goes through it before reaching DRAM.

The LLC is divided into sets. Each set holds a fixed number of cache lines, determined by the associativity of the cache. The RTX 3060 host machine has a 12 MB LLC with 16-way associativity, giving 12288 sets. The A30 host has a 60 MB LLC with 15-way associativity, giving 65536 sets. When a memory access comes in, the processor uses the physical address of the requested cache line to decide which set it belongs to. Specifically, bits 6 through $6 + \log_2(N_{\text{sets}}) - 1$ of the physical address are used as the set index:

$$S = \left\lfloor \frac{P}{64} \right\rfloor \bmod N_{\text{sets}} \tag{3.1}$$

where P is the physical address and the division by 64 accounts for the 64-byte cache line size.

This mapping applies equally to all processors on the system. When the GPU fetches a cache line through PCIe, it goes through the same LLC and maps to the same set as any CPU access to the same physical address. This is the property the attack depends on.

Figure 3.2 shows the measured access latency for each level of the memory hierarchy on the RTX 3060 host machine, benchmarked using pointer-chasing chains to prevent prefetching. An LLC hit takes 28 cycles while a DRAM access takes 190 cycles, a ratio of nearly 7×. This is the latency gap that the probe phase exploits: a slow probe (DRAM latency) indicates the victim evicted the attacker's line, while a fast probe (LLC latency) means the set was undisturbed. The pagemap syscall at 605 cycles is shown separately as it is a software cost, not a hardware cache level.

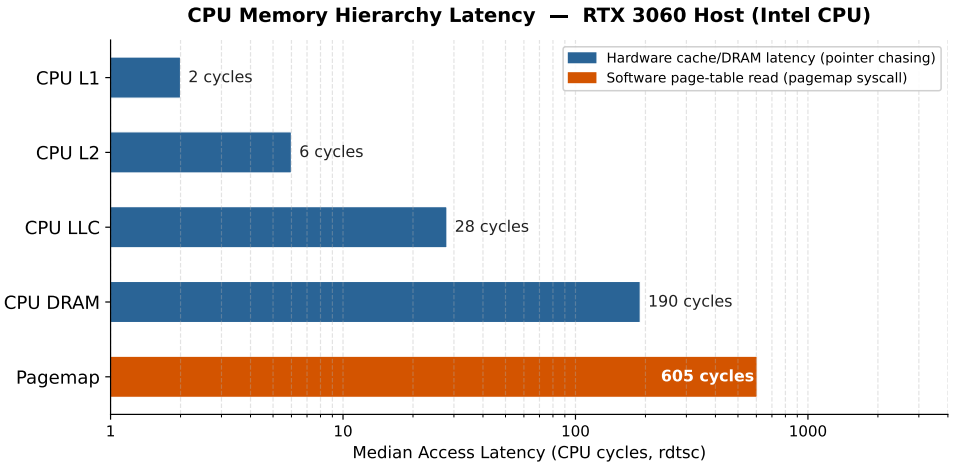

FIGURE 3.2: Measured CPU memory hierarchy latency on the RTX 3060 host (Intel CPU, pointer-chasing benchmark). L1=2, L2=6, LLC=28, DRAM=190 cycles. The pagemap syscall (605 cycles) is a one-time software cost per page during address discovery.

3.3 CUDA Unified Virtual Memory and the Shared LLC Path

Normally, GPU memory and CPU memory are physically separate. A GPU has its own GDDR memory, and CPU programs live in DRAM. Data has to be explicitly copied between them with `cudaMemcpy`.

CUDA Unified Virtual Memory changes this by giving both processors a shared virtual address space. Memory allocated with `cudaMallocManaged` can be accessed by both the CPU and the GPU using the same virtual address. The CUDA runtime manages where the physical pages actually live and migrates them as needed.

For our attack, we need the managed pages to stay in CPU DRAM, not in GPU memory. If a page migrates to the GPU, the GPU accesses it locally from its own GDDR memory and the CPU LLC is not involved at all. To keep pages in CPU DRAM, we prefetch them to the CPU device explicitly:

```
cudaMemPrefetchAsync(ptr, size, cudaCpuDeviceId, stream);
```

With the pages pinned in CPU DRAM, every GPU access to those pages goes over PCIe and passes through the CPU's LLC. This is what creates the shared cache path

between the GPU attacker and the CPU victim. Both processors are now reading and writing through the same LLC sets.

3.4 Physical Address Translation via Pagemap

For Prime+Probe to work, the attacker needs to fill the *same* LLC sets that the victim is using. With separate virtual address spaces this is not obvious, because two virtual addresses that look different may or may not map to the same physical location and therefore the same LLC set.

Linux provides the `/proc/self/pagemap` interface, which maps virtual page numbers to their corresponding physical frame numbers (PFN). By reading 8 bytes at offset VPN $\times$ 8 in this file, a process can retrieve the physical frame number of any of its own pages. The physical address of a specific byte within that page is then:

$$P = (\text{PFN} \times \text{page\_size}) + \text{page\_offset} \tag{3.2}$$

Since Linux kernel 4.0, the PFN field in pagemap is zeroed for unprivileged processes. Root access is required to get the real physical address. This is why the attack requires running with `sudo`: without the real PFN, we cannot compute LLC set indices and the targeting becomes blind.

Once we have the physical address of a managed memory page, the LLC set it maps to follows directly from the formula in the previous section. We allocate a large array of UVM pages, read the physical address of each cache line, and keep only the lines that map to the target LLC set. These become the prime and probe addresses for the attack.

3.5 The GPU L2 Cache Problem

There is a complication that is easy to miss. The GPU has its own L2 cache, which is separate from the CPU's LLC. When the GPU accesses a UVM page that is resident

in CPU DRAM, the data comes over PCIe, passes through the CPU LLC, and then gets cached in the GPU's L2 as well.

This creates a problem for the probe phase. After the victim CPU process has run and potentially evicted our prime data from the LLC, we want the probe access to go back through the LLC so we can measure the latency. But if the probe address is still sitting in the GPU's L2 cache from a previous access, the probe will hit the GPU L2 and we will see a fast result regardless of what the CPU did to the LLC. The LLC-level eviction by the victim becomes invisible.

To fix this, we run a flush kernel before the probe phase. The flush kernel accesses a large separate array, large enough to thrash the entire GPU L2 (16 MB on the RTX 3060, and similar on the A30), forcing all previously cached lines out. After the flush, any subsequent access to our probe addresses must go through the LLC again, which is where the victim's eviction signal lives.

3.6 The Prime+Probe Loop

With the above pieces in place, the full attack loop works as follows.

Prime. A GPU kernel iterates over the set of LLC-congruent addresses identified via pagemap and reads each one. These accesses go over PCIe, through the CPU LLC, and into the GPU L2. The target LLC set is now filled with the attacker's cache lines.

Flush GPU L2. A separate flush kernel accesses a large scratch buffer to evict all the prime data from the GPU's own L2. After this step, the prime data still sits in the CPU LLC, but is no longer in the GPU L2.

Victim runs. The CPU victim process runs one forward pass of the AI model. This accesses model weights and activations from DRAM, some of which map to the same LLC sets as our prime data. These accesses push the attacker's lines out of those sets.

Probe. The GPU kernel accesses the same prime addresses again and measures the access time for each one using the GPU hardware clock counter `clock64()`. Because

the GPU L2 was flushed, these accesses go through the LLC. If the victim disturbed a set, the probe access misses the LLC and goes to DRAM, taking several times longer. If the victim did not touch that set, the data is still in the LLC and the probe is fast.

The timing difference between an LLC hit and an LLC miss (from the GPU's perspective, this is roughly the difference between a PCIe transfer that hits LLC versus one that goes all the way to DRAM) is large enough to be reliably detectable with `clock64()`.

3.7 Synchronisation Between the Attacker and the Victim

One practical issue is coordinating the three phases so they happen in the right order. The prime must complete before the victim starts, and the probe must happen after the victim finishes its inference pass.

We handle this with a pair of shared atomic flags in managed memory, visible to both the CPU and GPU threads. The GPU signals when the prime is done, the CPU victim waits for that signal before running inference, and once inference completes it sets a flag that unblocks the GPU probe kernel. This gives us a clean sequential execution of prime, victim, probe without any busy waiting on either side.

3.8 Attack Platforms

We ran experiments on two machines. The first is a workstation with an NVIDIA RTX 3060 GPU. The RTX 3060 has a 3.84 MB GPU L2 cache, and the host CPU has a 12 MB 16-way LLC with 12288 sets. The second machine is a server with two NVIDIA A30 GPUs. The A30 has a 24 MB GPU L2, and the host has a 60 MB 15-way LLC with 65536 sets. The larger LLC on the A30 platform means we need a larger UVM array to cover enough LLC sets, and the larger GPU L2 means the flush kernel needs a bigger scratch buffer.

On the A30 we also ran a cross-GPU variant where GPU 0 handles the pagemap discovery and allocates the managed memory, while GPU 1 runs the prime and probe kernels. The two GPUs communicate through peer-to-peer memory access. The details of that variant are covered in the implementation chapter.

Chapter 4

Implementation

The attack is implemented in two CUDA programs. The first, `prime_probe_ai_scan.cu`, runs the single-GPU version where one GPU acts as the attacker and a CPU thread runs the victim model on the same machine. The second, `prime_probe_ai_scan_xgpu.cu`, is the cross-GPU variant for the A30 server, where GPU 0 handles address discovery and GPU 1 runs the prime and probe phases. Both programs are compiled with `nvcc` and linked against LibTorch for loading and running the AI models.

4.1 Threat Model

We consider a multi-tenant cloud environment in which two virtual machines (VMs) are co-located on the same physical host (Figure 4.1) and share microarchitectural resources, including the last-level cache (LLC). The hypervisor enforces strict software isolation between VMs; however, no hardware cache partitioning is assumed. The victim VM (Tenant A) is provisioned with CPU-only resources and runs a convolutional neural network (CNN) inference workload. The execution of the CNN involves frequent memory accesses to model weights and activations, which are mapped in the CPU's virtual address space and managed through multi-level CPU page tables. These page tables are actively accessed during execution due to address translation and memory management operations.

The attacker VM (Tenant B) is provisioned with both CPU and GPU resources and leverages CUDA Unified Virtual Memory (UVM). Under UVM, GPU memory accesses may trigger address translation events that involve both GPU-side page tables (GMMU) and CPU-side page tables. In particular, when a requested page is not resident in GPU memory, the GPU generates a page fault that is serviced by the CPU. This process involves CPU page table walks and updates performed by the GPU driver, as well as data migration between CPU and GPU memory. The attacker exploits the fact that CPU page table accesses (including page walks, updates, and page fault handling) are served through the shared LLC. These accesses introduce observable cache activity that can be monitored by a co-located attacker.

Attacker's capabilities: The attacker operates within its own VM and requires no privileges beyond standard OS-level access. It can obtain physical address mappings (e.g., via `/proc/self/pagemap`) to identify LLC set mappings. Using GPU-accelerated memory accesses enabled by UVM, the attacker can generate controlled pressure on the memory hierarchy and synchronize probing with GPU-induced memory events. Crucially, the attacker leverages GPU-triggered memory behavior (e.g., page faults, migrations, and translation requests) to amplify and correlate CPU page table activity, enabling more precise observation of cache set usage associated with page table accesses.

Attacker's objective: The attacker aims to infer victim workload characteristics by observing CPU page table activity through the LLC side channel. Specifically, the goal is to reconstruct memory access patterns indirectly via page table accesses, and use this information to identify or distinguish CNN inference workloads.

In our experiments we approximate this setup by running both the GPU attacker and the CPU victim on the same physical machine, in the same process address space, coordinated through atomic flags. This gives us a clean controlled environment to measure the timing signal. In a real deployment the victim and attacker would be in separate VMs with no shared memory, which would only increase the noise slightly since the LLC sharing happens at the hardware level regardless of virtualisation boundaries.

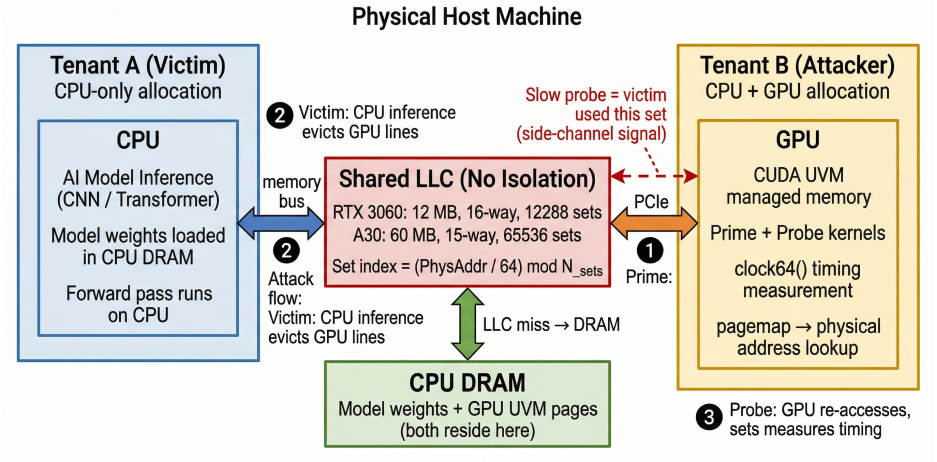

FIGURE 4.1: Threat model: two tenants on the same physical host. Tenant A runs CPU-only AI inference (victim). Tenant B runs a GPU process (attacker) that observes Tenant A through shared LLC contention.

4.2 Address Discovery

The first thing the program does after startup is find which addresses in its managed memory array map to the target LLC set. This is handled by two small functions, `vtop` and `llc_set`.

`vtop` opens `/proc/self/pagemap`, seeks to the entry for the virtual page containing the address, and reads the 8-byte entry. Bit 63 of that entry indicates whether the page is present in physical memory. Bits 0 through 54 give the physical frame number. Multiplying the PFN by the page size and adding the page offset gives the full physical address. If the PFN field is zero (which happens when running without root) the function returns an error, which is why `sudo` is required.

`llc_set` calls `vtop` and then applies the LLC indexing formula: divide the physical address by 64 to get the cache line number, then take it modulo the number of sets. The number of sets is computed from the cache size and associativity passed as command-line arguments.

The main setup loop allocates a large managed memory array (64 MB on the RTX 3060, 128 MB on the A30) and walks through it in 64-byte steps, calling `llc_set` on each cache line. Lines that return the target set number are collected into a list. The loop stops once it has found enough congruent lines, typically a few hundred to a

few thousand depending on the array size and LLC configuration. In practice we found around 300 to 2000 congruent lines per target set in a 64 MB array, which is more than enough since the LLC associativity on these machines is 15 or 16 ways.

The managed memory pages are prefetched to the CPU device before the scan:

```
cudaMemPrefetchAsync(data, size, cudaCpuDeviceId, 0);
```

This is important. Without this call the CUDA runtime may migrate some pages to GPU memory, and those pages will not have a valid physical address in the CPU DRAM address space. The pagemap read will either return zero or an address that does not correspond to a CPU-side LLC set.

4.3 GPU Kernels

There are three CUDA kernels in the attack.

Prime kernel. The prime kernel takes the list of congruent line indices and reads each one four times across multiple thread blocks. Running four passes ensures that all the congruent lines are loaded into the LLC regardless of the replacement policy. The kernel is launched with 8 blocks of 256 threads each, so 2048 threads are working in parallel to fill the set:

```
prime_kernel<<<8, 256>>>(data, idx_dev, n_used);
```

Flush kernel. The flush kernel accesses a separate GPU-local buffer (allocated with `cudaMalloc`, not managed memory) large enough to evict the entire GPU L2 cache. On the RTX 3060 this buffer is 16 MB; on the A30 it is 32 MB. The kernel XORs each element to prevent the compiler from optimising the accesses away. After this kernel finishes, none of the prime data remains in the GPU L2, so the subsequent probe must fetch from the CPU LLC or DRAM:

```
flush_gpu_l2<<<16, 256>>>(flush_buf, FLUSH_INTS);
```

Probe kernel. The probe kernel runs on a single thread (block 0, thread 0) and accesses each congruent line one at a time, measuring the time for each access individually using `clock64()`. Serial access is important here: if multiple threads probed concurrently they would interfere with each other's timing. A `__threadfence_system()` call before and after each timed access ensures the GPU does not reorder the clock read and the memory access:

```
__threadfence_system();
uint64_t t0 = clock64();
volatile int v = data[idx[i]];
__threadfence_system();
uint64_t t1 = clock64();
timings[i] = t1 - t0;
```

The timings array is allocated as managed memory so the CPU can read the results after the probe kernel completes without an explicit device-to-host copy.

4.4 CPU Victim Thread

The CPU victim runs in a separate `std::thread`. It loads the AI model from a TorchScript `.pt` file using `torch::jit::load` and evaluates it in CPU mode with `torch::kCPU`. All model weights stay in CPU DRAM throughout. The input tensor is created to match the expected shape for each model: a $2 \times 3 \times 224 \times 224$ image tensor for the vision models, and a 2×128 integer token sequence for the transformer models.

The thread runs in a loop waiting for a signal from the GPU side. Three `std::atomic <bool>` flags coordinate the timing: `g_start_victim` tells the CPU to begin inference, `g_victim_done` tells the GPU that inference is complete, and `g_victim_run` is used to cleanly shut down the thread at the end. The victim thread does nothing between signals other than spin on the start flag.

4.5 The Measurement Loop

Each run of the program performs `reps` repetitions. Within each repetition there are two measurements: a baseline and an attack.

For the **baseline**, the GPU primes the target set, flushes its L2, and then waits for a short idle period before probing. The idle period is a `usleep(500)` call, chosen to approximate the time it takes the victim to run one inference pass. This gives us a timing reference for what a probe looks like when no victim is active.

For the **attack**, the GPU primes and flushes, then signals the victim thread to run one forward pass. The GPU busy-waits on `g_victim_done` and only launches the probe kernel after the victim signals completion. This ensures the probe always happens immediately after one complete inference pass.

The per-line timings from each probe are stored and accumulated across all repetitions. At the end, the median, 99th percentile, and maximum are computed for both baseline and attack distributions for each congruent line. These statistics, along with their ratios (attack divided by baseline), are written to a CSV file. A high ratio on a given line means the victim consistently evicted it, indicating that the victim's working set overlaps with that LLC set.

4.6 Cross-GPU Variant

On the A30 server, which has two GPUs, we also implemented a cross-GPU variant in `prime_probe_ai_scan_xgpu.cu`. The motivation is to test whether the attack remains practical when the address discovery and the probing happen on separate physical devices.

GPU 0 (the pagemap GPU) allocates the managed memory array and runs the address discovery scan. The pages are prefetched to CPU memory before the scan so that `/proc/self/pagemap` returns valid physical addresses. GPU 0 does not run any prime or probe kernels.

GPU 1 (the probe GPU) runs all three kernels: prime, flush, and probe. It accesses the managed memory that was allocated by GPU 0. This works because CUDA Unified Virtual Memory is visible to all CUDA devices on the system, and peer-to-peer access is enabled between the two A30 GPUs at startup:

```
cudaDeviceEnablePeerAccess(probe_gpu, 0);
```

GPU 1's flush buffer is allocated locally with `cudaMalloc` on device 1, so flushing the GPU L2 only affects GPU 1's cache, which is the one doing the probing. The rest of the measurement loop and synchronization are identical to the single-GPU version.

One issue we encountered during development was that prefetching the managed pages to GPU 0 instead of to the CPU caused the pagemap scan to return zero PFNs. When pages are in GPU memory, they are not mapped in the CPU's page tables and `/proc/self/pagemap` cannot see them. The fix is always to prefetch to `cudaCpuDeviceId` before running the pagemap scan, regardless of which GPU will later be doing the probing.

4.7 AI Models Used as Victims

We used five models as victim workloads, all loaded in TorchScript format and evaluated entirely on the CPU:

- **MobileNet-V2**: a lightweight image classification model with 3.4 M parameters. Relatively small memory footprint.
- **SqueezeNet 1.0**: an even smaller classification model designed for low-resource settings.
- **EfficientNet-Lite0**: a more recent efficient architecture with structured depthwise separable convolutions.

- **TinyYOLO-v2**: a small object detection model with a different access pattern from the classification models, as it has a fully convolutional backbone followed by detection heads.
- **MiniTransformer**: a custom small transformer model exported to TorchScript on CPU. The positional encoding uses a `torch.arange` call that must be traced on the CPU device to avoid hardcoded CUDA device references in the exported model.

Models were warmed up with three forward passes before the measurement loop to ensure that the operating system had paged in all model weight pages and that any JIT compilation overhead was absorbed before the timed runs.

4.8 Running the Experiments

Both programs take the same core arguments: path to the model file, target LLC set number, number of congruent lines to use (points), number of repetitions, LLC size in KB, and LLC associativity. For example, to scan set 768 on the RTX 3060 with 32 congruent lines and 64 repetitions:

```
./prime_probe_ai_scan mobilenet_v2.pt 768 32 64 12288 16
```

The heatmap experiments were run with a shell script that swept across 16 evenly spaced LLC sets and 3 independent runs per model, generating one CSV file per set per run. The Python analysis scripts then read these CSVs to produce per-model heatmaps of the attack signal across sets and runs.

Chapter 5

Results and Analysis

This chapter presents the experimental results from three sets of experiments. First, a synthetic workload experiment on the RTX 3060 that sweeps five different GPU+CPU workload combinations to characterise the raw attack signal. Second, the main AI model experiment on the RTX 3060 with five inference models as victims. Third, the cross-GPU variant on the A30. All experiments use 3 independent runs per configuration.

5.1 What the Timing Signal Looks Like

Before looking at per-model results it helps to understand what the raw timing signal looks like. In both the baseline and attack conditions, the GPU probe kernel records the clock64() latency for each congruent cache line. The baseline is measured with the CPU idle, and the attack is measured immediately after one CPU inference pass.

For most probes, the two distributions are nearly identical. The median probe latency across all models and sets sits around 2540–2560 GPU clock cycles in the baseline condition. In the attack condition, the median barely moves, staying within 10–20 cycles of baseline for almost all measurements. This is expected: the LLC is large (12 MB on the RTX 3060), and on any given probe only a fraction of the sets we target will overlap with the memory the CPU victim happened to access in that particular inference pass.

The signal lives in the *tail* of the distribution. When the victim's inference does touch a set we are probing, our primed lines get evicted and the probe latency spikes. These spikes show up clearly in the 99th percentile and maximum of the probe timing distribution, while the median stays flat. This bursty behaviour is characteristic of Prime+Probe: the eviction event is intermittent, not continuous, so tail metrics are the right thing to look at.

5.2 Synthetic Workload Experiments (RTX 3060)

Before testing AI inference models, we ran a sweep of five synthetic GPU+CPU workload combinations to understand how the attack signal scales with workload intensity. In each scenario the GPU attacker runs its normal prime and probe phases while a CPU thread runs a fixed synthetic computation as the victim. The five scenarios are:

1. **None / None** – no CPU or GPU victim workload (pure baseline)
2. **GPU matmul** 128×128 **/ CPU matmul** 64×64 – light compute
3. **GPU matmul** 512×512 **/ CPU matmul** 256×256 – heavy compute
4. **GPU conv2d** $64 \times 64 \times 32$ **/ CPU matmul** 64×64 – convolution victim
5. **GPU bandwidth 16 MB / CPU FFT 8K** – memory-bandwidth victim

Figure 5.1 shows the p99 ratio heatmap for all five scenarios. The x-axis is the address index within a set (up to 64 congruent addresses per set) and the y-axis is the LLC set ID. Brighter red means a higher eviction ratio.

The none/none scenario shows almost no signal, confirming that the baseline probe latency is stable when neither processor is doing memory-intensive work. The matmul scenarios show dramatically stronger signals: the heavy 512×512 case turns large portions of the heatmap deep red, with ratios reaching 10^4 at some address-set combinations. This makes sense because large matrix multiplications stream a lot

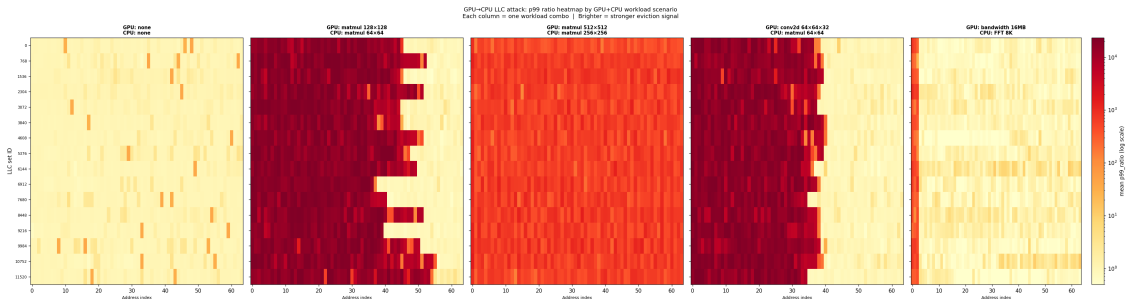

FIGURE 5.1: p99 ratio heatmaps for five synthetic workload scenarios on the RTX 3060. Left to right: no workload, light matmul, heavy matmul, conv2d, and bandwidth+FFT. The color scale is logarithmic. Heavier CPU workloads produce much stronger and more widespread eviction signals.

of data through the CPU's LLC, evicting the attacker's primed lines very reliably on every pass.

The conv2d and bandwidth+FFT scenarios sit between the two extremes, showing moderate eviction patterns with some hot sets. These results confirm that the attack signal is not specific to any one workload type. Any CPU computation with a significant LLC footprint will produce a detectable eviction pattern in the GPU probe timing.

5.3 Results on the RTX 3060 (AI Models)

Table 5.1 summarises the results across all five models on the RTX 3060. The two most important columns are the maximum observed p99 ratio (attack p99 divided by baseline p99) and the number of LLC sets where this ratio exceeded 1.2, indicating a clear eviction signal.

TABLE 5.1: RTX 3060 Prime+Probe results across all sets and runs. p99 ratio = attack p99 / baseline p99. Sets >1.2 = number of sets where the ratio exceeded 1.2 across any congruent line.

Model	**Baseline p99**	**Attack p99**	**Max p99 ratio**	**Sets >1.2×**
TinyYOLO-v2	2540	2567	2.48×	8
MiniTransformer	2545	2551	2.45×	1
MobileNet-V2	2543	2549	1.11×	0
EfficientNet-Lite0	2555	2561	1.10×	0
SqueezeNet-1.0	2562	2565	1.11×	0

TinyYOLO-v2 produces the strongest signal by a significant margin. Eight different LLC sets show a p99 ratio above 1.2, and the strongest single observation is at LLC set 10752, where the attack p99 reaches 6302 cycles against a baseline of 2541 cycles, a ratio of 2.48×. At set 3072 we observed 5737 vs 2663 (2.15×), and at set 1536, 5072 vs 2393 (2.12×). TinyYOLO takes a $1 \times 3 \times 416 \times 416$ input tensor and has a larger memory footprint than the classification models, which means its forward pass touches more LLC sets and generates more evictions.

MiniTransformer shows a strong but more localised signal. One set produces a p99 ratio of 2.45×, comparable to TinyYOLO's peak, but across all sets only one crosses the 1.2 threshold. The transformer's attention mechanism and weight matrices appear to concentrate their LLC pressure into a smaller number of sets.

MobileNet-V2, EfficientNet-Lite0, and SqueezeNet-1.0 all show very weak signals. The maximum p99 ratio stays below 1.12 for all three, and no set exceeds the 1.2 threshold. These are all compact models designed for efficient inference, with a relatively small memory footprint. Their weight accesses during the forward pass simply do not generate enough LLC pressure in the sets we are probing to produce a reliable eviction signal.

Figure 5.2 shows the full p99 ratio heatmap for all five models on the RTX 3060. Each column in a panel represents one congruent address within that LLC set, so the heatmap captures both which sets are hot and which specific addresses within those sets drive the signal. TinyYOLO shows scattered dark patches across multiple sets. MiniTransformer shows a concentrated dark band at one set. The three compact models stay uniformly light.

Figure 5.3 shows the same data as per-set line plots (one line per run), which makes the reproducibility across runs easier to read. The three run lines stack closely for both TinyYOLO and MiniTransformer, confirming the hot sets are consistent across independent runs.

Figure 5.4 shows the probe timing CDF for TinyYOLO-v2, which most clearly illustrates what the attack signal looks like in the raw distribution. The two CDFs are nearly identical up to around the 90th percentile. Above that point the attack

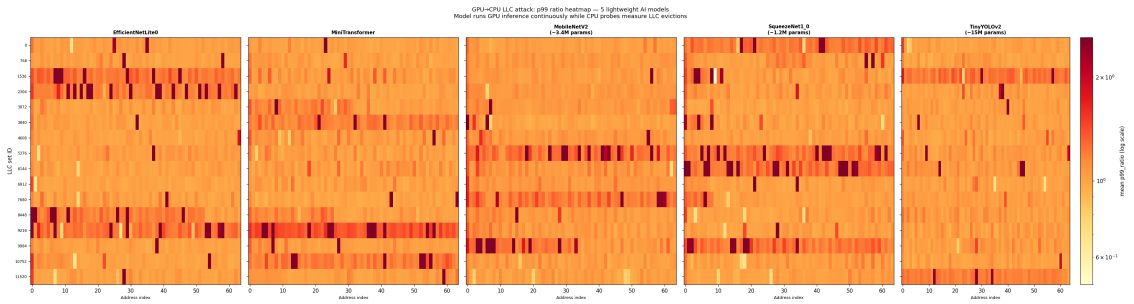

FIGURE 5.2: p99 ratio heatmap (LLC set ID vs. address index) for all five AI models on the RTX 3060. Brighter red indicates stronger eviction signal. TinyYOLO-v2 shows multiple hot set regions; MiniTransformer has one concentrated hot band; the three compact models show no significant pattern.

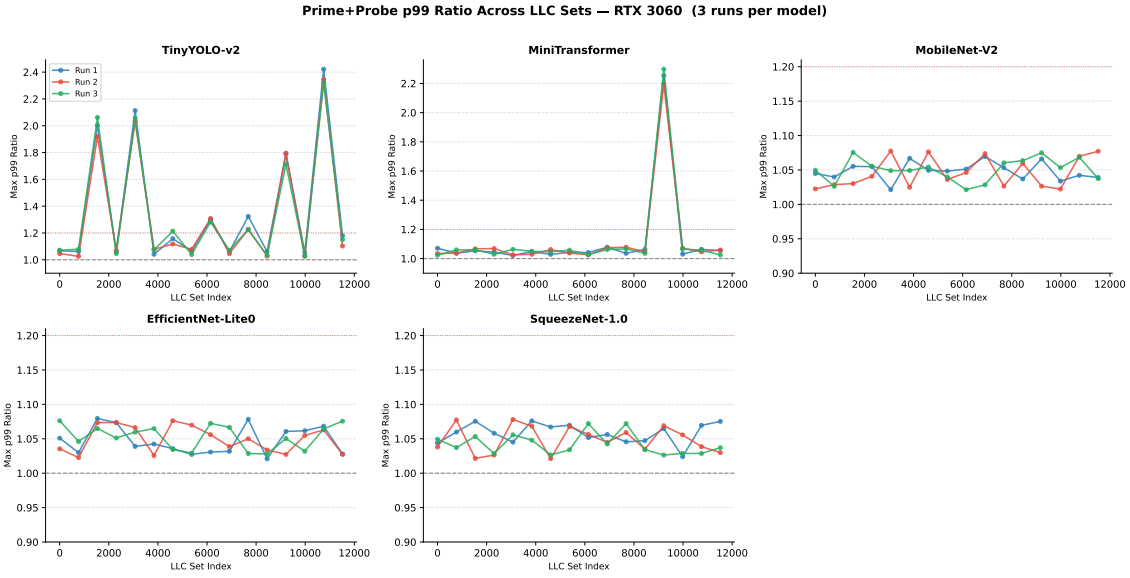

FIGURE 5.3: p99 ratio across 16 LLC sets for all five models on the RTX 3060, shown as line plots with one line per run. TinyYOLO-v2 and MiniTransformer show reproducible spikes above the 1.2× threshold; the three compact models remain flat near 1.0.

distribution develops a heavy tail, with a fraction of probe accesses returning latencies above 4000 cycles while the baseline stays tightly bounded below 2700 cycles. This tail separation is the fundamental signal the attack exploits.

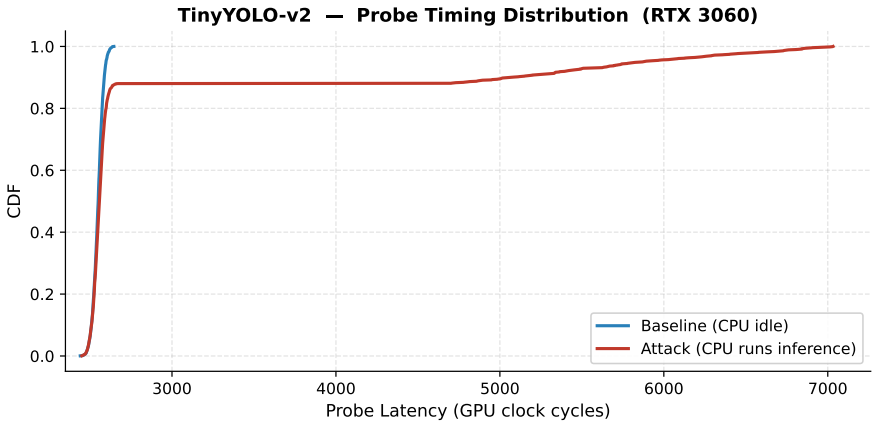

FIGURE 5.4: Probe timing CDF for TinyYOLO-v2 on the RTX 3060, showing baseline (CPU idle) and attack (CPU runs inference) distributions. The two curves diverge in the upper tail, where the attack distribution shows latency spikes from LLC evictions caused by the victim's forward pass.

5.4 Results on the A30 (Cross-GPU Variant)

Table 5.2 shows the results from the cross-GPU experiment on the A30 server, where GPU 0 performed address discovery and GPU 1 ran the prime and probe phases.

TABLE 5.2: A30 cross-GPU Prime+Probe results. GPU 0 handles pagemap discovery, GPU 1 runs prime and probe. p99 mean and max are across all sets and runs.

Model	p99 ratio mean	p99 ratio max	max ratio mean
TinyYOLO-v2	1.031	1.118	0.96
MobileNet-V2	1.030	1.098	0.96
MiniTransformer	1.030	1.718	0.98
SqueezeNet-1.0	1.028	1.138	0.96
EfficientNet-Lite0	1.026	1.100	0.95

The cross-GPU results show a consistent but smaller signal than the RTX 3060 single-GPU results. The p99 ratio mean sits at 1.026–1.031 across all models, meaning that on average the p99 probe latency is about 3% higher in the attack condition than in the baseline. This is a smaller effect than the RTX 3060, where specific sets showed ratios above 2×.

The A30's LLC is five times larger than the RTX 3060's (60 MB vs 12 MB), with 65536 sets compared to 12288. We only swept 16 sets in our experiments, which means a smaller fraction of the total LLC is covered. The victim's memory accesses

are spread across a much larger set space, reducing the probability that any given probed set gets disturbed in a single inference pass.

MiniTransformer again shows the most prominent individual spike, with a p99 max of 1.718× at one set. This is consistent with the RTX 3060 result where MiniTransformer also showed a concentrated high-ratio set. The cross-GPU architecture does not destroy the signal; it just reduces its amplitude due to the larger LLC and additional P2P memory access path noise.

The max_ratio mean being slightly below 1.0 (around 0.95–0.98) for the A30 results indicates that the absolute maximum probe latency occasionally drops slightly in the attack condition compared to baseline. This is likely noise from the P2P access path introducing additional variability in the timing measurements.

Figure 5.5 shows the full p99 ratio heatmap for all five models on the A30. The y-axis now spans 65536 sets, which immediately shows how much larger the A30's LLC is compared to the RTX 3060. MiniTransformer has a visible dark horizontal band around set 32768. The other models show only scattered spots. Compared to the RTX 3060 heatmap in Figure 5.2, the A30 heatmap is noticeably more uniform, reflecting the diluted signal across a larger set space.

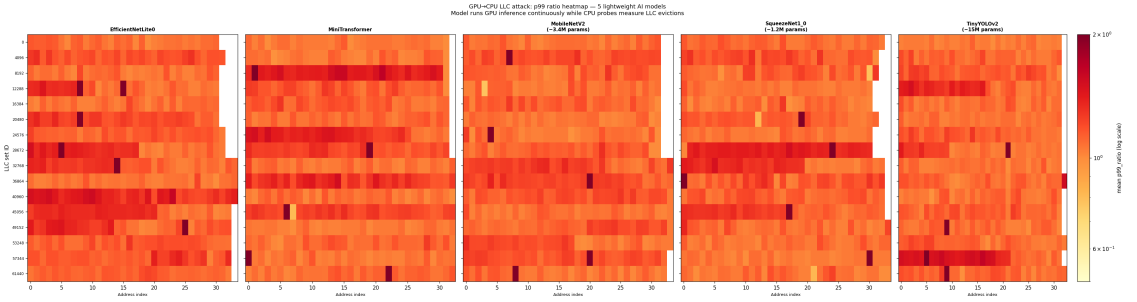

FIGURE 5.5: p99 ratio heatmap for all five AI models on the A30 cross-GPU experiment. The y-axis spans 65536 LLC sets (vs. 12288 on the RTX 3060). MiniTransformer shows a concentrated dark band around set 32768; the other models produce only sparse hot spots.

Figure 5.6 shows the same data as line plots across the 16 sampled sets. Most sets show ratios clustered near 1.0, with only MiniTransformer producing a visible spike. This confirms the quantitative picture from Table 5.2.

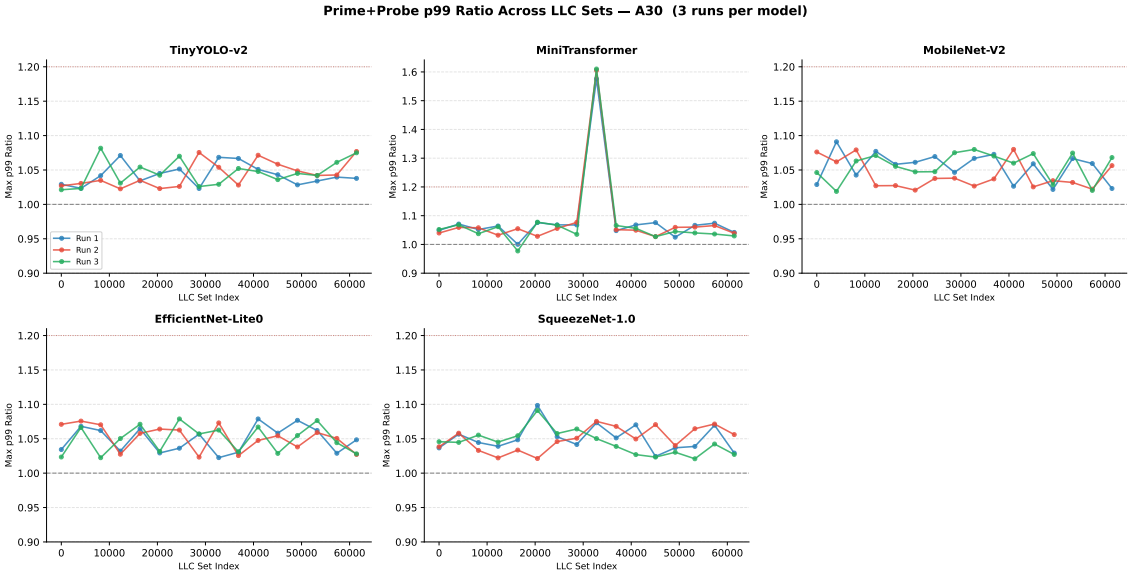

FIGURE 5.6: p99 ratio across 16 LLC sets for all five models on the A30 cross-GPU experiment (3 runs per model). The overall signal is weaker than on the RTX 3060 due to the larger LLC, but MiniTransformer still shows a notable spike at one set.

5.5 Reproducibility Across Runs

One important question is whether the signal is consistent or just noise that happens to produce high ratios occasionally. Across all three independent runs on both platforms, the sets that show elevated ratios are largely the same. For TinyYOLO on the RTX 3060, sets 10752, 3072, and 1536 appear as hot spots in all three runs. For MiniTransformer, the same single set dominates in all runs.

This reproducibility confirms that the eviction pattern is driven by the model's actual memory access pattern, not random timing variation. The same LLC sets carry the same model's data across different inference passes, which is expected given that the model weights are fixed and the forward pass executes the same computation each time.

5.6 Discussion

The model-dependence of the signal is the most informative result. TinyYOLO and MiniTransformer produce strong signals while the three compact classification models barely register. This aligns directly with memory footprint: a larger model sends more weight data through the LLC per forward pass, creating more chances for overlap with the GPU attacker's primed sets. In a realistic scenario, the per-set timing pattern could help an attacker identify not just that inference is running, but which model class is running.

The A30 cross-GPU result shows the attack is not tied to one specific GPU architecture. The signal survives cross-device P2P access, with the main limiting factor being the much larger LLC that dilutes victim evictions across more sets than we swept.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis demonstrates a practical GPU-to-CPU Prime+Probe side-channel attack on the shared Last-Level Cache. The GPU acts as the attacker using CUDA Unified Virtual Memory to keep its pages in CPU DRAM, making every probe access pass through the CPU's LLC over PCIe. Physical address targeting via `/proc/self/pagemap` lets the attacker prime and probe exact LLC sets. The only privilege required is root.

Experiments on the RTX 3060 show that TinyYOLO-v2 and MiniTransformer produce clear, reproducible eviction signals (p99 ratios up to 2.48×), while the three compact classification models produce no detectable signal. The difference is driven by memory footprint: larger models disturb more LLC sets per forward pass. The cross-GPU variant on the A30 confirms the attack generalises across architectures; the weaker signal there is explained by the five-times-larger LLC diluting victim evictions across more sets than we swept.

The per-set timing pattern is model-dependent and reproducible, meaning the same LLC sets appear as hot spots across independent runs for the same model. This is enough for an attacker to distinguish which model class the victim is running, purely from cache timing, without any access to the victim's code or data.

6.2 Future Work

Set coverage. On the A30 we swept only 16 of 65536 sets. A proportional sweep would surface stronger signals and build a more complete per-model LLC usage profile.

Model architecture inference. The current attack detects which model is running but does not recover internal structure. Extending the analysis to infer layer dimensions and hyperparameters, as Cache Telepathy [7] did for the CPU-to-CPU case, is a direct next step.

Automated fingerprinting. Training a classifier on LLC timing heatmaps collected over multiple models and runs would make the attack fully automated, removing the need for manual heatmap interpretation.

Real cloud evaluation. Our experiments approximate the multi-tenant scenario by running attacker and victim in the same process. Testing in a real KVM or container environment, where synchronisation cannot use shared memory, would validate whether the measured signal survives realistic deployment conditions.

Reverse direction. The CPU-as-attacker, GPU-as-victim variant is worth investigating for GPU workloads that use UVM with CPU-side page residency. Conventional GPU kernels migrate data to GDDR and would not be vulnerable, but UVM-heavy workloads may be.

Bibliography

[1] Sungbin Jang, Junhyeok Park, Osang Kwon, Yongho Lee, and Seokin Hong. Rethinking page table structure for fast address translation in GPUs: A fixed-size hashed page table. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, pages 325–337, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer, 2006.

[3] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732. USENIX Association, 2014.

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

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

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

[7] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium*, pages 2003–2020. USENIX Association, 2020.

[8] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat USA*, 2015.