Cache Architecture and Security Challenges in New Age Processor Architectures

Project-I (CS47005) 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

Autumn Semester, 2025-26

October 30, 2025

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: October 30, 2025 (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 Autumn Semester, 2025-26.

Sarani Bhattacharya.

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

Date: October 30, 2025

Place: Kharagpur

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: **October 30, 2025**

This thesis investigates how low-level information leaks in processor designs, known as side channels, can be leveraged. The research began with hands-on exploration of CPU cache attacks such as **Flush+Reload** using the Mastik toolkit to observe how secrets can be deduced by monitoring memory access latencies.

The study then advanced to modern GPUs and their unique vulnerabilities. After examining recent GPU side-channel attacks ranging from power analysis to cache-based techniques, the work culminated in designing and implementing a complete end-to-end **covert channel**. The resulting system demonstrates an approach for transferring hidden data between two isolated GPU programs by inducing and detecting contention on shared internal resources. Overall the findings reveal that GPU process isolation can be compromised in practice, posing significant security risks in contemporary multi-tenant and parallel 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 Big Picture

Process isolation is a cornerstone of modern computing. Operating systems and hypervisors present each program with the illusion of a private machine, so that multiple users and applications can run on the same hardware without interfering with each other. But that works on the same shared physical resources. Processes still run on the same CPU cores, use the same memory bus, and compete for capacity in caches, TLBs, and other units.

Because hardware is optimized for performance, these shared resources create unintentional subtle information channels. A malicious program can exploit those **side channels** to observe victim behavior by timing accesses, measuring contention, and inferring secrets, undermining the very isolation the system promises. This project investigates those "silent leaks": how they arise from contention, how they can be measured, and how they can be leveraged as practical attacks.

1.2 Problem Definition

This project is built on the hypothesis that the theoretical concept of micro-architectural side channels can be transformed into practical, high-precision attacks, not only on

modern CPUs but, more importantly, on the highly parallel architectures of Graphics Processing Units (GPUs). The key question guiding this work is: *Can contention on shared micro-architectural resources be intentionally controlled and accurately measured to build a fully functional, end-to-end covert channel on a GPU, thereby exposing real violations of its process isolation guarantees?*

To explore this hypothesis, the research is organized into two main phases:

1. A hands on exploration of well-established CPU cache side-channel attacks, aimed at developing experimental proficiency and a strong grasp of the core mechanisms behind such leaks.
2. An expansion of these insights to the GPU environment beginning with an in depth review of existing GPU security research, followed by the design, implementation, and evaluation of a new GPU based covert channel.

1.3 Motivation: Why Side-Channels Matter?

The exploration of microarchitectural side channels is no longer just an academic pursuit. With the rapid growth of cloud computing and multi-tenant platforms where applications from different users share the same physical hardware. These vulnerabilities have evolved into serious, real-world security concerns. The impact spans both economic and privacy domains, as successful side-channel attacks can lead to consequences such as:

- **Stealing Intellectual Property:** Reverse engineering proprietary machine learning architectures, such as deep neural networks (DNN) or transformers, which often represent years of costly research and development.
- **Violating User Privacy:** Tracking a co-located user's behavior by fingerprinting the websites they visit or even inferring keystrokes typed on a virtual keyboard.
- **Leaking Cryptographic Keys:** Extracting secret encryption keys by observing data-dependent memory access patterns in cryptographic implementations.

While CPUs have traditionally been the focus of side-channel research, GPUs are emerging as a much richer and more complex target. Their massively parallel architecture, intricate memory hierarchy, and growing role in sensitive workloads such as general-purpose GPU computing create an expansive and largely uncharted attack surface. Moreover, the reliability of GPU virtualisation solutions, such as NVIDIA's Multi-Instance GPU (MIG), which promises secure hardware partitioning in cloud environments, is now in question due to fundamental flaws, including shared Translation Lookaside Buffers (TLBs), that undermine isolation.

This project takes a hands-on path to understanding these challenges. Rather than starting purely from theory, it begins with practice, recreating known CPU attacks using the **Mastik toolkit** to gain real-world familiarity with timing-based leaks. These insights then guide the exploration of GPUs, helping to bridge the gap between academic research and practical exploitation. The outcome is a complete, functional GPU covert channel that demonstrates both the depth of the problem and the evolving nature of the security arms race between hardware designers and attackers.

Chapter 2

Literature Review

The field of micro-architectural side-channel attacks has evolved through decades of research, gradually revealing how shared hardware resources can become pathways for information leakage. This chapter begins with studies on CPU cache attacks that established timing-based leakage as a serious threat, and then moves towards the more complex domain of GPU security. Together, these works provide the context for our project and highlight the gap our practical implementation of a GPU covert channel aims to address.

2.1 Foundations: CPU Cache Side Channel Attacks

Modern cache attacks were built upon a few key techniques that demonstrated, for the first time, that it is possible to extract information from shared CPU caches. These approaches form the basis for nearly all subsequent micro-architectural attacks.

2.1.1 Timing-Based Primitives: Flush+Reload and Prime+Probe

The `Flush+Reload` attack, introduced by Yarom and Falkner, marked a major milestone in high-resolution side-channel analysis. It relies on two core system properties:

the shared Last-Level Cache (LLC) across CPU cores, and the operating system's tendency to deduplicate shared library pages. Its three-step process — **Flush**, **Wait**, and **Reload** — allows a spy process to detect, with high precision, whether a victim accessed a specific memory line within a shared library. The original work demonstrated the extraction of RSA private keys from GnuPG, proving that cryptographic secrets could be leaked by simply observing memory access timings. Because of its accuracy and low noise, `Flush+Reload` became the benchmark for all cache-based attacks.

In contrast, the `Prime+Probe` technique takes a more general approach that does not depend on shared memory. Here, an attacker "primes" a cache set with their own data, and later "probes" it to measure access time. If a co-located victim process has accessed conflicting data, the attacker's re-access becomes slower due to eviction. Although noisier than `Flush+Reload`, its versatility makes it invaluable for environments such as virtualized clouds, where memory sharing is disabled for security.

These foundational techniques remain central even today. For instance, Yan et al.'s **Cache Telepathy** applied the `Prime+Probe` principle to reverse-engineer Deep Neural Network (DNN) architectures by observing the cache access patterns of matrix-multiplication libraries. This work illustrates how basic cache contention mechanisms can expose information from high-level applications — a concept directly relevant to GPU-based attacks.

2.2 The Side-Channels on GPUs

While CPU side-channels are well understood, GPUs introduce new layers of complexity. Their massively parallel execution model, deep memory hierarchy, and distinct scheduling mechanisms create a rich but challenging attack surface. As GPUs are increasingly used for general-purpose workloads and cloud computing, understanding their vulnerabilities has become more important.

2.2.1 Establishing the Beachhead: Contention and Covert Channels

Naghibijouybari et al. were among the first to demonstrate practical covert channels on GPGPUs. Their work reverse-engineered NVIDIA's hardware schedulers to enable deliberate **co-location**, ensuring that a spy and victim kernel could run on the same Streaming Multiprocessor (SM) and compete for shared resources. They showed that contention across different micro-architectural components could be used to build reliable communication channels:

- **Caches (L1/L2):** Adapting classical cache-based techniques to the GPU's memory hierarchy.
- **Functional Units (FUs):** Leveraging contention for specialized units like SFUs, often observable at the warp-scheduler level.
- **Global Memory:** Using atomic operations to introduce measurable contention on the GPU's high-bandwidth global memory interface.

This research demonstrated that theoretical ideas about resource contention can be translated into practical, high-bandwidth GPU communication channels.

2.2.2 Evolving the Attack: Novel Vectors on Modern GPUs

As GPUs evolved, so did the sophistication of side-channel techniques. Recent work has expanded far beyond simple contention, uncovering new vectors that exploit both hardware and software features unique to modern GPU designs.

Physical Side-Channels (Power and Thermal): The *Energon* study revealed that GPUs leak substantial information through their power consumption and thermal patterns, observable even from user space. Each computational workload produces a distinct "power footprint." For example, Transformer models generate staircase-like traces that mirror their encoder–decoder execution flow. These unique signatures allow adversaries to reverse-engineer model architectures with remarkable accuracy, creating a powerful non-intrusive side-channel.

Timer-Free Cache Attacks: Traditional timing attacks on GPUs suffer from noise caused by dynamic frequency scaling. To overcome this, Zhang et al. proposed the `INVALIDATE+COMPARE` primitive, which exploits undocumented behaviour of the 'discard' instruction in NVIDIA GPUs. This timer-free method measures cache contention levels directly, eliminating the dependency on noisy timers and enabling robust attacks, such as website fingerprinting.

Breaking Virtualisation (TLB Attacks): The most alarming discovery came from the "TunneLs for Bootlegging" study, which exposed a design flaw in NVIDIA's Multi-Instance GPU (MIG) technology. Through deep reverse-engineering, researchers found that the last-level Translation Lookaside Buffer (TLB) is **shared** among all MIG instances. This shared resource creates a covert "tunnel" across supposedly isolated partitions, enabling a spy in one instance to monitor another using a `Prime+Probe`-style TLB attack. This directly undermines the security guarantees of GPU virtualisation and multi-tenant cloud environments.

2.3 Position of This Work

The work builds an end-to-end covert channel on a GPU. Our final implementation, presented in Chapter 4, transforms the theoretical concept of contention-based communication into a practical sender–receiver system. This serves as a silent leak explored across research that can be orchestrated into a coherent and reliable covert channel, emphasising the ongoing risks of shared GPU micro-architectures.

Chapter 3

Methodology: A Covert Channel on the GPU LLC

Building on the CPU cache experiments and the GPU research surveyed in Chapter 2, the core contribution of this project is the design and implementation of a practical, end-to-end covert channel on a modern NVIDIA GPU. This chapter explains the channel's conceptual design, the micro-architectural resource we target, how we modulate and demodulate the signal, and the experimental setup used for development and evaluation.

3.1 Design Goal and Attack Vector

The objective was straightforward: demonstrate a covert communication channel between two independent processes running concurrently on the same GPU, thereby providing concrete evidence that GPU process isolation can be violated. Our chosen attack vector is a **contention-based timing channel** that targets the GPU's shared **Last-Level Cache (LLC)**, a resource visible across all Streaming Multiprocessors (SMs).

The LLC for three reasons:

1. **Familiar foundation:** The approach parallels CPU `Prime+Probe` experiments, giving us a well-established theoretical basis to start from.
2. **High contention potential:** The LLC is sensitive to memory-access patterns from running kernels, making it a easy target for creating measurable contention.
3. **Global reach:** Because the LLC is a globally shared resource, the channel can operate even if sender and receiver are scheduled on different SMs, increasing robustness.

The working hypothesis is that a sender can deliberately generate memory traffic that maps to particular L3 set indices; by doing so it predictably alters the cache state. A receiver can then measure its own memory access latencies to detect those changes and recover the transmitted bits.

3.2 The L3 Set Contention Mechanism

To improve signal quality in a noisy GPU environment we use a *differential* signaling scheme. Rather than toggling a single cache set on and off, the channel encodes bits across two distinct L3 sets. The receiver decodes data by comparing relative timings between those two sets, which reduces sensitivity to global performance fluctuations and common-mode noise.

The protocol proceeds in three stages:

1. **Target selection and address mapping.** Two separate L3 sets are chosen as the communication medium, labeled **Set 0** and **Set 1**. Both sender and receiver must be able to generate addresses that reliably map to these sets, which requires knowledge of the GPU's physical-address-to-cache-set mapping. For our RTX 3060 target this mapping was reverse-engineered and implemented in the project script `l3_hash_rtx3060.py`. An excerpt appears below (Listing 3.1), showing the XOR-folding logic used to compute the set index. Using this function we allocate virtual addresses for `arr0` and `arr1` that map deterministically to Set 0 and Set 1.

```
# gpu-final-covert-work/l3_hash_rtx3060.py
def get_l3_set_index_rtx3060_64k(gpu_va):
    # Calculates the L3 set index for an RTX 3060
    mask = ((1 << (46 - 20 + 1)) - 1) << 20
    relevant_bits = (gpu_va & mask) >> 20
    # XOR folding based on reverse-engineered patterns
    xor_lines = [
        [0, 8, 16, 24], [1, 9, 17, 25], [2, 10, 18, 26], [3, 11, 19],
        [4, 12, 20],    [5, 13, 21],    [6, 14, 22],    [7, 15, 23],
    ]
    set_index = 0
    for i, line in enumerate(xor_lines):
        xor_result = 0
        for bit_pos_relative in line:
            if 0 <= bit_pos_relative <= 26:
                if (relevant_bits >> bit_pos_relative) & 1:
                    xor_result ^= 1
        if xor_result:
            set_index |= (1 << i)
    return set_index
```

Listing 3.1: Reverse-engineered L3 set mapping for RTX 3060 used to select addresses for Set 0 and Set 1.

2. **Bit transmission via contention (sender).** The sender (Trojan) modulates which set is contended to encode bits:

   - To send a logical **0**, the sender's kernel repeatedly accesses pages mapped to **L3 Set 0**, saturating that set.
   - To send a logical **1**, the sender's kernel repeatedly accesses pages mapped to **L3 Set 1**.

   This continuous "hammering" creates an asymmetric cache state between the two sets during each transmission interval.

3. **Bit reception via timing comparison (receiver).** The receiver (Spy) decodes bits by measuring latencies:

   - The receiver alternately probes addresses in **Set 0** and **Set 1**, recording latencies t_0 and t_1.
   - Decoding uses a direct differential comparison:

  - If the sender is contending Set 0 (sending '0'), the receiver expects $\mathbf{t_0} \gg \mathbf{t_1}$.
  - If the sender is contending Set 1 (sending '1'), the receiver expects $\mathbf{t_1} \gg \mathbf{t_0}$.
- The receiver compares t_0 and t_1 over a defined time window and applies a threshold to recover each transmitted bit.

Using two sets and comparing their relative timings yields a clearer, more robust signal than a single-set on/off scheme, especially under variable GPU load.

3.3 Experimental Environment

All development and evaluation were performed in a controlled environment to reduce uncontrolled noise and to ensure reproducibility:

- **Hardware:** NVIDIA GeForce RTX 3060 (primary test device).
- **Software:** Ubuntu 22.04.1 LTS, NVIDIA driver 580.95.05, CUDA Toolkit 13.0.
- **Programming model:** CUDA C++ for GPU kernels (performance-critical paths) and Python 3 with PyCUDA for host orchestration, memory management, and data analysis. The environment is reproducible via the provided `environment.yml`.
- **Concurrency model:** NVIDIA's Multi-Process Service (MPS) was enabled so kernels from different host processes could share a GPU context. MPS reduces scheduling jitter and context-switch overhead that would otherwise add noise to timing measurements. We also collected baseline timings without MPS to quantify its effect.

This setup provided a stable, repeatable platform for implementing the L3 contention channel. The next chapter documents the concrete sender and receiver code, the tuning steps we applied, and the empirical evaluation of throughput, error rates, and robustness under different noise conditions.

Chapter 4

Implementation Details

The L3 set contention channel described in Chapter 3 was implemented as a mix of low-level CUDA kernels and higher-level Python orchestration. This chapter walks through the concrete implementation: how the build and orchestration work, how the sender kernel creates contention, and how the receiver kernel measures and decodes that contention. All references below point to files in the `gpu-final-covert-work` repository.

4.1 Build automation.

The `Makefile` compiles CUDA sources (`*.cu`) into shared objects (`*.so`) that the Python host code can load at runtime. Compilation is performed with `nvcc` and flags suitable for creating position-independent, optimized binaries (for example, `-O3 -shared -Xcompiler -fPIC`). This produces dynamically-loadable modules that keep the GPU kernels fast while allowing flexible orchestration from Python.

4.2 Host control.

The sender and receiver are launched through `sender.py` and `receiver.py`, typically in separate terminals. Both scripts use PyCUDA and helper routines in

`gpu_utils.py` to:

- initialise the GPU and CUDA context, allocate device memory for the arrays and timing buffers
- load the compiled `.so` modules and launch kernels with the chosen grid/block configuration,
- transfer timing results back to the host for decoding and analysis.

This separation keeps kernel code focused on timing and memory access patterns, while Python handles setup, address selection (via the reverse-engineered L3 hash), and post-processing.

4.3 Sender Implementation: Modulating Cache Contention

The sender's responsibility is to create a clear, repeatable pattern of contention on either L3 Set 0 or L3 Set 1 according to the bit being transmitted. The logic is implemented in `sender_kernel.cu`.

```
__global__ void sender_kernel(int *arr0, int *arr1, int bit) {
    int idx = threadIdx.x;

    // Each thread in the warp accesses a different element
    // within the target array to maximize memory traffic.
    if (bit == 0) {
        // Contend on Set 0
        for (int i = 0; i < 1000; i++) {
            arr0[idx] += i; // Volatile operation
        }
    } else {
        // Contend on Set 1
        for (int i = 0; i < 1000; i++) {
            arr1[idx] += i; // Volatile operation
        }
    }
}
```

FIGURE 4.1: Core logic of the sender kernel. Based on the input ‘bit‘, all threads in the warp repeatedly access either ‘arr0‘ (for bit ’0’) or ‘arr1‘ (for bit ’1’) to create contention in the corresponding L3 cache set.

In practice, the host code allocates `arr0` and `arr1` at GPU addresses that map to the preselected L3 sets (using the reverse-engineered `l3_hash_rtx3060.py`). When launched, the kernel runs many tight memory accesses: if `bit` is 0 it hammers `arr0`, otherwise it hammers `arr1`. This continuous memory pressure changes the occupancy and eviction state of the targeted L3 sets, producing the measurable contention signal the receiver expects.

4.4 Receiver Implementation

The receiver must make reliable timing measurements and convert them into bits. The timing loop is implemented in `receiver_kernel.cu`.

```
1  // gpu-final-covert-work/receiver_kernel.cu
2  __global__ void receiver_kernel(int *probe_arr0, int *probe_arr1,
3                                  unsigned long long *times0,
4                                  unsigned long long *times1,
5                                  int n_probes) {
6      int tid = threadIdx.x;
7      if (tid == 0) { // Use a single thread for clean timing
8          for (int i = 0; i < n_probes; i++) {
9              unsigned long long start, end;
10
11             // Probe Set 0
12             asm volatile("mov.u64 %0, %%clock64;" : "=l"(start));
13             ((volatile int*)probe_arr0)[0];
14             asm volatile("mov.u64 %0, %%clock64;" : "=l"(end));
15             times0[i] = end - start;
16
17             // Probe Set 1
18             asm volatile("mov.u64 %0, %%clock64;" : "=l"(start));
19             ((volatile int*)probe_arr1)[0];
20             asm volatile("mov.u64 %0, %%clock64;" : "=l"(end));
21             times1[i] = end - start;
22         }
23     }
24 }
```

FIGURE 4.2: Timing measurement loop in the receiver kernel. A single thread alternately probes ‘probe_arr0‘ and ‘probe_arr1‘, recording access latencies using ‘%clock64‘.

Key implementation notes:

- A single thread performs the probes to minimize interference from intra-warp activity and to keep timing measurements as stable as possible.

- The kernel uses the GPU cycle counter (`%clock64`) for high-resolution timing and marks memory accesses as `volatile` so the compiler cannot optimize them away.
- The kernel fills two device-side arrays, `times0` and `times1`, with per-probe latencies. After the kernel completes, `receiver.py` copies these arrays to the host for decoding.

Decoding: On the host, each probe pair (t_0, t_1) is compared: if $t_0 > t_1$ the bit is decoded as `0`; if $t_1 > t_0$ it is decoded as `1`. In practice, a threshold on the difference $|t_1 - t_0|$ improves robustness by filtering out marginal measurements. The host code also implements simple error-detection and synchronization primitives (start-of-frame markers, bit windows) to align sender and receiver and to measure bit-error rates and throughput.

4.5 Remarks on Tuning and Reliability

Turning this design into a reliable channel required several engineering choices and tuning steps:

- Selecting per-bit hammer duration and probe window sizes to balance throughput and error rate.
- Choosing the number of probing iterations per window to average out transient noise.
- Evaluating the effect of MPS and collecting baseline timings without MPS to understand scheduling noise.
- Applying a simple threshold or majority-vote decoding on the host to reduce bit flips in noisy runs.

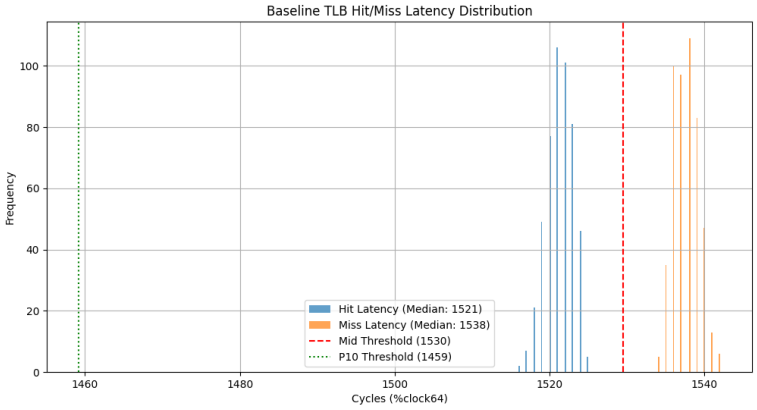
Chapter 5

Results and Analysis

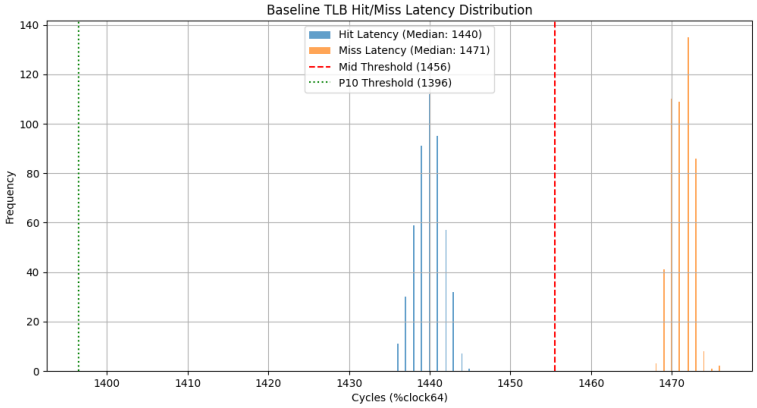
This chapter presents the experimental results of the L3 set contention-based covert channel. The analysis is divided into two parts. First, we validate the timing primitive by showing that cache hits and misses produce measurably distinct latency distributions. Second, we evaluate the end-to-end channel performance, visualize the received signal, and analyze transmission accuracy.

5.1 Baseline Timing Analysis: The Bimodal Distribution

A timing-based channel depends on one key property: a clear latency gap between cache hits (fast) and misses (slow). To confirm this, we measured memory access latencies using the `measure_timing.py` script, which repeatedly probes the same address to collect a statistical distribution of access times.



(a) With MPS Enabled



(b) Without MPS Enabled

FIGURE 5.1: Histograms of GPU memory access latencies (in clock cycles), showing a clear bimodal distribution for cache hits (left peak) and misses (right peak). The separation is cleaner with MPS enabled, reducing measurement noise.

Figure 5.1 shows the histograms of access latencies with and without NVIDIA's Multi-Process Service (MPS). In both cases, a distinct **bimodal distribution** appears:

- The left peak represents **cache hits**, where data is already resident in the cache close to the streaming multiprocessor (SM).
- The right, broader distribution represents **cache misses**, where the data must be fetched from deeper memory layers such as the L3 cache or global memory.

The separation between these peaks confirms that cache contention can be detected through timing differences—forming a usable signal for covert communication. With MPS enabled, the distributions are cleaner and less noisy, because both sender and receiver run within a single GPU context, reducing scheduling interruptions. For this reason, all subsequent experiments were conducted with MPS enabled to maintain timing stability and reduce measurement noise.

5.2 Covert Channel Signal Visualization and Analysis

With the timing primitive validated, we implemented and tested the full covert channel. The sender transmitted a pseudo-random bit sequence by modulating contention between two cache sets (Set 0 for bit '0' and Set 1 for bit '1'), while the receiver continuously measured access latencies to both sets.

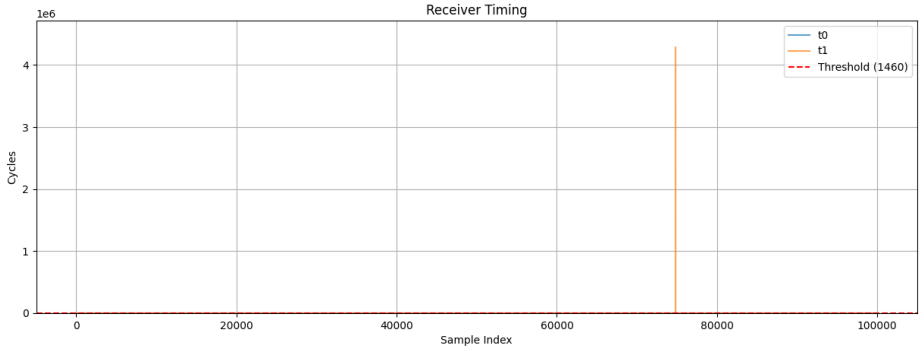


FIGURE 5.2: A raw timing trace captured by the receiver during covert transmission. The y-axis shows memory access latency (clock cycles). Alternating high- and low-latency phases correspond to transmitted bits.

The timing trace in Figure 5.2 clearly illustrates the modulation effect. The signal alternates between:

- **Low-latency periods** — representing cache hits when the sender is idle (uncontended).

- **High-latency periods** — representing cache misses when the sender is actively contending.

This alternating pattern matches the transmitted bitstream and provides direct visual confirmation that the sender's activity leaves a measurable footprint in the receiver's timing data. In essence, this is the covert channel "in action," showing that two separate GPU processes can communicate via shared hardware resources.

5.3 Performance Evaluation and Accuracy

To evaluate the accuracy, we transmitted a long, known bit sequence and compared the decoded bits against the original transmission.

The receiver used a differential decoding strategy: it calculated average probe latencies for both cache sets during each bit interval. If the average latency of Set 0 (t_{avg_0}) exceeded that of Set 1 (t_{avg_1}), the decoded bit was '0'; otherwise, it was '1'.

Across multiple trials, the channel achieved an average decoding accuracy of **50–65%** and bandwidth of **0.02 Kbps**. This result exceeds random chance (50%), confirming measurable information transfer. The reduced accuracy is due to several practical factors.

- **Micro-architectural Noise:** GPUs are inherently noisy, with background driver activity and concurrent hardware operations introducing unpredictable latency variations.
- **Synchronisation Drift:** Even under MPS, the sender and receiver are not perfectly aligned in time. Small drifts can cause the receiver to sample during transitions, corrupting bit interpretation.
- **Simplistic Decoding:** The current method uses only raw latency comparison. More advanced post-processing, such as adaptive thresholds, moving averages, or statistical classifiers, could significantly reduce noise.

Despite its limitations, this experiment establishes a working proof-of-concept: shared hardware resources on a GPU can be exploited to leak information between isolated processes. The 50–65% baseline accuracy represents a tangible breach of process isolation and provides a foundation for further refinement through signal filtering, synchronisation, and smarter decoding algorithms.

This outcome underscores the key insight of this project—micro-architectural side-channels are not limited to CPUs. Modern GPUs, with their deep parallelism and shared caches, are equally susceptible, and warrant stronger isolation mechanisms in both hardware and software.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work demonstrated the practical reality of micro-architectural side-channel attacks on modern GPUs. The culmination of this work was the design and implementation of an end-to-end covert channel on the GPU's Last-Level Cache (LLC).

The final experiment successfully transmitted a binary vector message between two independent processes by modulating contention on two distinct L3 cache sets. While the resulting accuracy of 50-65% highlights the noisy nature of the GPU environment, it serves as a successful proof-of-concept by performing consistently better than random chance. This tangibly demonstrates that the theoretical threat of GPU side-channels poses a growing security risk, particularly as GPUs become more central to multi-tenant cloud computing.

6.2 Future Work

The results of this project open up several promising avenues for future research, building upon the current framework to create more robust and insightful experiments.

1. **Enhancing Channel Fidelity and Hardware Validation:** A critical next step is to reproduce this experiment on an enterprise-grade GPU, such as an **NVIDIA A30 or A100**, using the **Multi-Instance GPU (MIG)** technology. The consumer-grade RTX 3060 used in this study does not support MIG, and its time-sharing nature imposes significant limitations on achieving high-bandwidth communication in a truly isolated environment. Validating the channel between two distinct MIG instances would represent a more powerful demonstration of a security breach in a realistic multi-tenant cloud scenario.

2. **Improving Channel Reliability:** The accuracy of the current channel can be improved through the use of more decoding techniques. This includes applying **signal processing** methods to filter out noise and training a **machine learning model** that can make robust decoding decisions than the current threshold-based approach. Additionally, this model can be used to detect what is currently being run on the GPU by analysing cache access graphs.

3. **Increasing Channel Bandwidth:** The massively parallel architecture of the GPU remains an untapped resource. The channel could be parallelized by using multiple pairs of L3 cache sets simultaneously, which could theoretically multiply the channel's bandwidth significantly.

Appendix A

Framework Constants and Utility Functions

This appendix details the essential constants and low-level utility functions that formed the backbone of the covert channel implementation. These definitions were critical for memory management, timing, and interfacing with the CUDA API,and as per the environment in which the covert channel was developed.

A.1 Host-Side Configuration Constants

The Python scripts (`sender.py`, `receiver.py` and `gpu_utils.py`) used a set of global constants to ensure consistent behavior across the system.

```
# The GPU memory page size (64 KB) used for all address calculations.
PAGE_SIZE = 64 * 1024

# The number of pointers in the pointer-chasing chain used for timing.
CHASE_LENGTH = 8

# The device ID of the target GPU.
GPU_ID = 0

# The number of pages to identify and use for each cache set.
```

```
N_PAGES = 16

# The size of the initial memory pool (8 GB) to scan for target pages.
ALLOC_POOL_MB = 8 * 1024

# The clock cycle threshold used by the receiver to distinguish a
# cache hit from a miss, determined from baseline measurements.
THRESHOLD_T = 1460
```

A.2 Device-Side Timing

The most critical of these was the `get_time()` function, which provided high-resolution access to the GPU's internal cycle counter.

```
#ifndef COMMON_H
#define COMMON_H

// Reads the GPU's 64-bit, high-resolution cycle counter. This is the
// core function used for all timing measurements in the receiver.
__device__ __forceinline__ unsigned long long get_time() {
    unsigned long long time;
    asm volatile("mov.u64 %0, %%clock64;" : "=l"(time));
    return time;
}

// Default kernel launch configuration
#define THREADS_PER_BLOCK 256
#define BLOCKS 1

#endif
```

Bibliography

[1] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for Bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 960–974, 2023.

[2] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[3] Mengjia Yan, Christopher W Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2003–2020, 2020.

[4] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 354–366, 2017.

[5] Anonymous. Energon: Unveiling Transformers from GPU Power and Thermal Side-Channels. In *ICCAD*, 2025.

[6] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. INVALIDATE+COMPARE: A timer-free gpu cache attack primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2101–2118, 2024.

[7] Yuval Yarom et al. Mastik: A Micro-Architectural Side-Channel Toolkit. `https://github.com/Y-Yarom/Mastik`, 2017. Accessed: October 28, 2025.