POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

TEXAS: probing workloads in shared GPU environments through TEXture and ASync Pipeline Side Channels

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Samuele Motta**

Student ID: 252672
Advisor: Prof. Luca Cassano
Co-advisors: Elia Lazzeri
Academic Year: 2025-26

Abstract

The widespread adoption of GPU sharing in cloud computing environments, research clusters, and multi-user workstations has introduced a significant security concern: when multiple users execute workloads concurrently on the same physical GPU, shared microarchitectural resources become a potential source of information leakage. This thesis presents TEXAS, a contention-based side-channel attack deploying two spy kernels on distinct hardware pipelines of NVIDIA GPUs: the Texture Units pipeline (TEX spy kernel) and the asynchronous global-to-shared memory copy pipeline (Async spy kernel), introduced in NVIDIA's Ampere architecture. Both spy kernels operate entirely from unprivileged user-level code, requiring no elevated permissions and no modifications to the victim's execution environment.

TEXAS incorporates a dual-stream classification framework that fuses the timing traces of both spy kernels into a unified BiLSTM-based classifier. By combining signals from two independent pipelines, the framework achieves higher discriminative power than either spy kernel used in isolation. The classifier is structured hierarchically: it first identifies the general workload category — among CNNs, Transformers, RNNs, diffusion models, and cryptocurrency mining — and then refines the classification to the model family and, where possible, the specific model. For CNNs, the framework additionally attempts to infer architectural properties through regression on the collected latency traces.

TEXAS is evaluated on three NVIDIA GPUs — RTX 3050, RTX 4070, and RTX 5070 — using 30 models across five categories. The dual-stream classifier consistently achieves weighted F1 scores above 0.90 at the category level across all tested configurations, demonstrating generalization across GPU architectures when retrained on data from the target device. At the model level, F1 scores range from 0.63 for structurally similar models to 0.96 for architecturally distinct ones, establishing that fusing signals from multiple independent pipelines provides a more robust fingerprinting primitive than any single-source approach.

Keywords: GPU side-channel attack, workload fingerprinting, contention-based attack, CUDA, deep learning security

Abstract in lingua italiana

La crescente diffusione della condivisione delle GPU in ambienti di cloud computing, cluster di ricerca e workstation multi-utente ha introdotto una rilevante problematica di sicurezza: quando più utenti eseguono workload in modo concorrente sulla stessa GPU fisica, le risorse microarchitetturali condivise possono diventare una fonte di fuga di informazioni. Questa tesi presenta TEXAS, un attacco side-channel basato sulla contesa che impiega due spy kernel su pipeline hardware distinte delle GPU NVIDIA: la pipeline delle Texture Units (TEX spy kernel) e la pipeline di copia asincrona dalla memoria globale alla memoria condivisa (Async spy kernel), introdotta nell'architettura Ampere di NVIDIA. Entrambi gli spy kernel operano con privilegi utente standard, senza richiedere permessi elevati né modifiche all'ambiente di esecuzione della vittima.

TEXAS incorpora un framework di classificazione dual-stream che combina le tracce temporali dei due spy kernel in un classificatore unificato basato su BiLSTM. Unendo i segnali di due pipeline indipendenti, il framework ottiene un potere discriminativo superiore rispetto a ciascun spy kernel usato singolarmente. Il classificatore è organizzato gerarchicamente: identifica prima la categoria del workload — tra CNN, Transformer, RNN, modelli di diffusione e mining di criptovalute — e affina poi la classificazione fino alla famiglia di modelli e, quando possibile, al modello specifico. Per le CNN, tenta inoltre di inferire proprietà architetturali tramite regressione sulle tracce di latenza raccolte.

TEXAS è valutato su tre GPU NVIDIA — RTX 3050, RTX 4070 e RTX 5070 — su 30 modelli in cinque categorie. Il classificatore dual-stream raggiunge F1 score pesati superiori a 0.90 a livello di categoria su tutte le configurazioni testate, dimostrando la generalizzazione tra architetture diverse quando riaddestrato sul dispositivo target. A livello di modello, gli F1 score variano da 0.63 per modelli strutturalmente simili a 0.96 per modelli architetturalmente distinti, confermando che combinare segnali da più pipeline indipendenti produce una primitiva di fingerprinting più robusta rispetto a qualsiasi approccio a sorgente singola.

Parole chiave: attacchi side-channel su GPU, fingerprinting dei workload, attacchi basati sulla contesa, CUDA, sicurezza del deep learning

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**
1.1 Outline of the Thesis 4

2 Background **5**
2.1 GPU Architecture 5
2.2 CUDA Execution Model 8
2.3 Texture Units and TEX Pipeline 9
2.4 Asynchronous Execution in GPUs 11

3 Related Work **13**
3.1 Covert-Channel Attacks 13
3.2 Fingerprinting Attacks 14
3.3 Cache-Based Attacks 14
3.4 Discussion of Limitations 15

4 Proposed Attack **17**
4.1 Threat Model 17
4.2 The TEXAS Attack 19
4.2.1 TEX Spy Kernel 20
4.2.2 Async Spy Kernel 21
4.2.3 Sequential Execution and Feature Fusion 21
4.2.3.1 Data Preprocessing 22
4.2.3.2 Dual-Stream Architecture 23
4.2.3.3 Training Procedure 24

4.3 Implementation Details . . . 24
4.3.1 Timing Mechanism . . . 24
4.3.2 Pseudo-Random Access Generation . . . 25

5 Evaluation **27**
5.1 Experimental Setup . . . 27
5.2 Target Workloads . . . 28
5.3 Attack Kernels Validation . . . 32
5.3.1 TEX Attack Kernel . . . 32
5.3.2 Async Attack Kernel . . . 38
5.4 TEXAS Assessment . . . 45
5.4.1 CNN Classification . . . 48
5.4.1.1 Model Classification . . . 48
5.4.1.2 Architecture Regression . . . 51
5.4.2 Transformer Classification . . . 54
5.4.3 RNN Classification . . . 57
5.4.4 Diffusion Classification . . . 57
5.5 Performance Comparison Across Different GPUs . . . 58
5.5.1 Category-Level Classification . . . 59
5.5.2 Model-Level Classification . . . 61
5.6 Discussion . . . 63
5.6.1 Overall Classification Performance . . . 63
5.6.2 Analysis of Difficult Cases . . . 63
5.6.3 Practical Security Implications . . . 64
5.6.4 Limitations . . . 65

6 Countermeasures **67**
6.1 Existing Countermeasures . . . 67
6.1.1 Hardware Isolation: MPS and MIG . . . 67
6.1.2 Cache Partitioning . . . 68
6.1.3 Memory Coloring and Partitioning . . . 68
6.1.4 Timer Restriction . . . 68
6.2 Proposed Countermeasures . . . 69
6.2.1 Enforcing Non-Bare-Metal Access . . . 69
6.2.2 Noise Injection into Pipeline Latencies . . . 69
6.2.3 Access Control on Hardware Performance Counters . . . 70
6.2.4 Pipeline Scheduling Randomization . . . 70

7 Conclusions and Future Developments **71**
7.1 Future Developments 72

Bibliography **75**

A Appendix A **79**

List of Figures **81**

List of Tables **83**

List of Symbols **86**

Acknowledgements **87**

1 | Introduction

In recent years, we have witnessed a remarkable evolution in Graphics Processing Units (GPUs): originally designed to accelerate graphics rendering, nowadays GPUs have evolved into powerful general-purpose accelerators, capable of handling a variety of different intensive workloads. Deep learning inference and training, scientific simulations, cryptographic computations, and large-scale data processing are some examples of applications that saw an incredible boost in performance thanks to the massive computation parallelization. The introduction of programming frameworks such as NVIDIA's CUDA has made GPU computing accessible to a broad developer community, enabling applications to launch thousands of concurrent threads that cooperate on shared hardware resources.

As GPU workloads have grown in both diversity and economic value, so has the practice of sharing GPU hardware among multiple users. In cloud computing environments, research clusters, and multi-user workstations, it is common for several applications to execute concurrently on the same physical GPU. This sharing model is primarily motivated by cost efficiency: high-end GPUs require substantial upfront investment, and letting multiple tenants access the same device increases overall hardware utilization. Technologies such as NVIDIA's Multi-Process Service (MPS) support this co-location by allowing several CUDA contexts to run on a single GPU without needing them to explicitly coordinate. Yet, this form of resource sharing creates an inherent trade-off between efficiency and isolation: although concurrent execution boosts throughput, it also causes workloads from different users or applications to compete for the same microarchitectural resources, such as caches, memory pipelines, and specialized functional units.

This lack of strong isolation has significant security implications. In the CPU domain, decades of research have demonstrated that shared microarchitectural state can be exploited to leak information between processes. Attacks such as *Flush+Reload* [13], *Prime+Probe* [12], and their many variants have shown that an adversary who shares a processor cache with a victim can infer sensitive information — ranging from cryptographic keys to user behavior patterns — by carefully measuring timing variations caused by cache contention. These attacks exploit a common principle: when two processes compete for a shared hard-

ware resource, the execution time of one process carries information about the behavior of the other.

GPUs, despite their very different architecture, are subject to analogous threats. Modern GPU microarchitectures feature a deep hierarchy of shared resources, from L2 caches shared across all Streaming Multiprocessors (SM) to specialized units such as Texture Units and dedicated memory copy pipelines within each SM. When multiple kernels execute concurrently on the same GPU, they inevitably contend for these shared components. An attacker who can co-locate a carefully designed measurement kernel alongside a victim's workload may observe latency variations that reveal information about the victim's activity.

Recent years have seen a growing body of research exploring GPU side-channel attacks. Prior work has demonstrated the feasibility of covert channels on GPUs [1–3, 6, 8, 14], attacks exploiting cache contention [15], and techniques that leverage memory access patterns to fingerprint specific applications [8]. These studies have established that GPU side channels are a practical concern, not merely a theoretical possibility. However, several limitations persist in the existing literature. Many proposed attacks target a single hardware component, limiting the amount of information that can be extracted. Others require specific assumptions about the victim's behavior or demand elevated privileges that may not be available in realistic deployment scenarios. Furthermore, to the best of our knowledge, no prior work has explored how timing signals from multiple independent hardware resources can be combined to achieve more robust and fine-grained workload classification.

In this thesis, we address these gaps by presenting TEXAS (TEXture and ASync Pipeline Side Channels), a contention-based side-channel attack that targets two distinct components of the NVIDIA GPU microarchitecture through dedicated spy kernels. Both spy kernels operate entirely from unprivileged user-level code, requiring only the ability to launch CUDA kernels and read the GPU's cycle counter — capabilities available to any standard CUDA application.

The TEX spy kernel targets the Texture Units and their associated caching hierarchy within the Streaming Multiprocessor. It continuously issues texture fetch operations using pseudo-random access patterns and measures the latency of each fetch. When a victim workload concurrently utilizes the texture pipeline or generates competing memory traffic, the spy observes a measurable increase in fetch latency compared to an idle baseline. By collecting and analyzing these latency traces over time, TEXAS can detect the presence of a co-resident workload and distinguish between different types of GPU activity.

The Async spy kernel targets the asynchronous global-to-shared memory copy pipeline introduced in NVIDIA's Ampere architecture. This pipeline, accessed through the `cp.async` instruction, provides a dedicated hardware path for transferring data from global memory to shared memory without involving the register file. The spy kernel issues sequences of asynchronous copy operations and measures the time required for the hardware to complete them. Since this pipeline is a shared resource, contention from a victim's memory operations causes observable increases in the synchronization latency measured by the spy.

A key design decision of TEXAS is that the two spy kernels are executed sequentially — never simultaneously — to avoid mutual interference. Their timing traces are then fused into a single feature vector processed by a dual-stream classifier. Since different workload categories may produce stronger contention signatures on one pipeline than the other, this fusion provides complementary information that improves the classifier's ability to distinguish between workloads. The combined framework enables a hierarchical classification scheme: TEXAS first identifies the general category of the victim's workload (e.g., transformer model, convolutional neural network, recurrent neural network, diffusion model, or cryptocurrency mining), and then attempts to refine the classification to identify specific model families or individual models within each category. For convolutional neural networks, TEXAS additionally attempts to infer architectural properties such as the number of convolutional and pooling layers through regression on the collected timing traces.

We evaluate TEXAS on different NVIDIA GPUs using a diverse set of victim workloads spanning multiple categories of deep learning models and mining algorithms. The evaluation considers both the individual spy kernels and their combination, assessing classification accuracy at each level of the hierarchical scheme. Our results demonstrate that, in some cases, the dual-stream classifier achieves higher accuracy than either individual spy kernel alone, confirming the benefit of leveraging multiple independent side-channel signals.

The contributions of this thesis can be summarized as follows:

- We present TEXAS, a contention-based side-channel attack on NVIDIA GPUs that deploys two spy kernels — targeting the Texture Units pipeline and the asynchronous global-to-shared memory copy pipeline respectively — and fuses their timing traces into a dual-stream BiLSTM classifier for hierarchical workload fingerprinting.
- We design the TEX spy kernel, which exploits contention on the Texture Units pipeline through pseudo-random texture fetch operations to fingerprint co-resident

GPU workloads.

- We design the Async spy kernel, which exploits contention on the asynchronous global-to-shared memory copy pipeline — a hardware feature introduced in NVIDIA's Ampere architecture not previously studied in the context of side-channel attacks.
- We demonstrate that TEXAS operates entirely from unprivileged user-level code, requiring no elevated permissions, no access to the victim's memory or code, and no modifications to the victim's execution environment.
- We evaluate TEXAS on a representative set of GPU workloads and show that the dual-stream approach can reliably distinguish between workload categories and, in many cases, identify specific models within each category.

1.1. Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the necessary background on GPU architectures, the CUDA execution model, and the hardware components targeted by our attack. Chapter 3 surveys related work on GPU side-channel attacks, covering cache-based, contention-based, and timing-based approaches, and discusses their limitations. Chapter 4 presents TEXAS in detail, including the threat model, the design of both spy kernels, the sequential execution strategy and feature fusion mechanism, and implementation considerations. Chapter 5 describes the experimental setup, the target workloads used as benchmarks, and presents the evaluation results along with a discussion of their implications. Chapter 6 surveys existing countermeasures relevant to the attack surface described in this work and proposes additional mitigations targeting the TEX and asynchronous copy pipelines specifically. Finally, Chapter 7 concludes the thesis and outlines directions for future work.

2 | Background

In this chapter, we introduce the necessary background on modern GPU architectures and their execution model. We describe the key hardware components of NVIDIA GPUs, summarize the CUDA programming model, and discuss the mechanisms that support massively parallel and asynchronous execution. These concepts are essential for understanding how the hardware responds when the spy kernels presented later in the thesis stress its resources. Unless explicitly attributed to another source, all terminology in this chapter follows the wording used in NVIDIA's official documentation.

2.1. GPU Architecture

Modern GPUs are organized as hierarchical assemblies of processing units designed to maximize parallel throughput. At the highest level of a GPU are the Graphics Processing Clusters (GPC), all of which share one L2 cache that serves as the final on-chip cache before accessing the main off-chip system memory. Each GPC contains a Raster Engine, responsible for performing the fixed-functionality Graphics Pipeline stages, and several Texture Processing Clusters (TPCs). Each TPC is made up of two Streaming Multiprocessors (SMs)—these two SMs are the primary programmable computing units. Both general-purpose and graphics workloads are executed by these main computing units.

Figure 2.1: High-level architecture of an NVIDIA Ada Lovelace GPU, showing the hierarchy of GPCs, TPCs, and SMs sharing a common L2 cache [9].

From a computing point of view, the Streaming Multiprocessor is the most important part of the GPU. Each SM is partitioned into four Processing Blocks, which share a set of resources at the SM level: a unified L1 data cache and shared memory, one or more Texture Units (Section 2.3), and a Ray Tracing (RT) Core. The L1 cache and shared memory occupy the same physical storage and can be configured to allocate more capacity to either, depending on the workload.

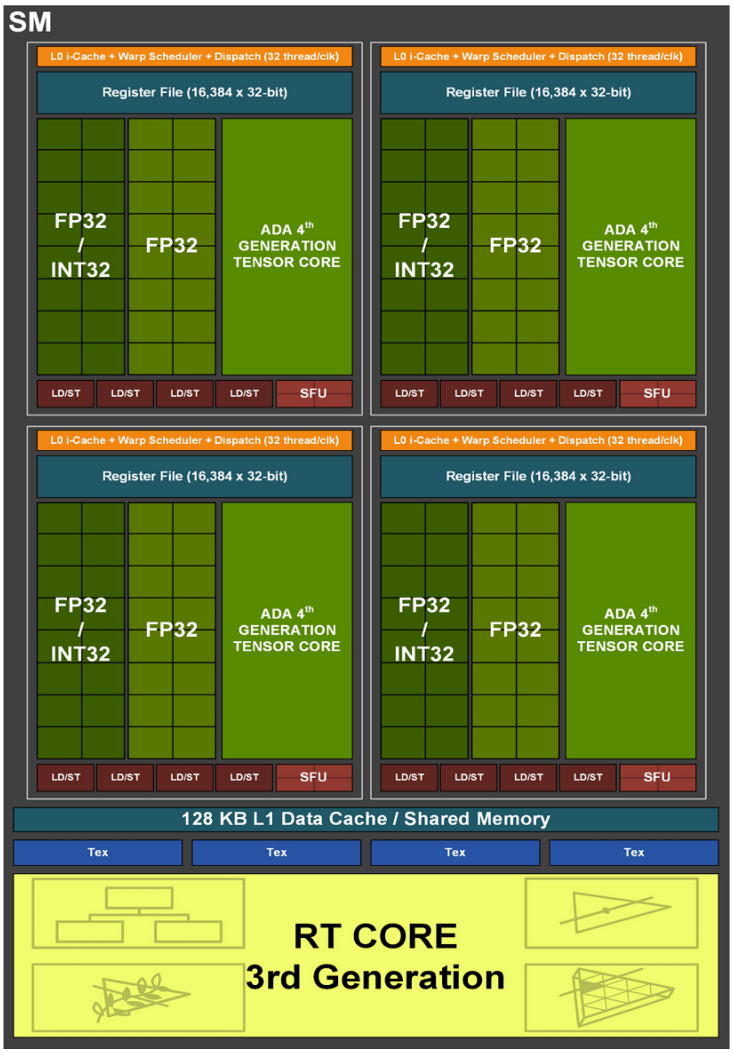


Figure 2.2: Internal architecture of a Streaming Multiprocessor (SM) in the NVIDIA Ada Lovelace architecture, showing the four Processing Blocks, the shared L1 cache and Texture Units [9].

Each Processing Block contains a register file, a Warp Scheduler with its corresponding Dispatch Unit, an L0 instruction cache, and multiple execution units. These components comprise CUDA Cores for performing integer and floating-point arithmetic operations, Tensor Cores dedicated to matrix multiply–accumulate computations frequently employed in deep learning workloads, a Special Function Unit (SFU) for evaluating transcendental functions such as sine and cosine, and a Load/Store unit responsible for generating and handling memory access requests. The Warp Scheduler selects warps that are ready for execution and dispatches their subsequent instructions to the corresponding execution units. Since each Processing Block has its own scheduler, the SM can issue instructions from up to four distinct warps.

These independent processing blocks function as hardware sub-cores sharing the broader intra-SM resources. In multi-process environments, when multiple independent programs run concurrently on the same GPU without hardware-level spatial isolation, the GPU scheduling infrastructure manages execution contexts across these physical structures via rapid time-slicing or command queue interleaving. This results in threads from different processes sharing underlying execution units and memory system resources, potentially leading to resource contention. Unlike traditional cache-state patterns that rely on detecting static data evictions from a cache hierarchy, contention-based execution tracking relies on measuring the transient latency variations caused by physical resource saturation within these shared pipelines.

2.2. CUDA Execution Model

GPUs were originally designed as specialized hardware accelerators for graphics rendering. Over time, GPU architectures evolved to include increasingly programmable components, moving to general-purpose computing thanks to the introduction of platforms such as CUDA (Compute Unified Device Architecture) from NVIDIA, which provides a programming model and software framework for executing non-graphics workloads on the GPU. The introduction of these platforms enabled developers to exploit parallel computing, allowing applications to launch thousands of threads that execute concurrently on the device.

Workloads on GPUs are executed in a SIMT (Single Instruction, Multiple Threads) fashion, as such, we can consider threads as the smallest component in the execution hierarchy of a CUDA program. Threads are scheduled and executed in groups of 32, called warps, and multiple warps can belong to a single block. Blocks are logical groupings of threads that are executed on the same SM, therefore, all the threads inside the same block share the L1 cache and can synchronize their work. The function launched from the host and executed in parallel by all threads in the grid is called a kernel. An example of defining and launching a kernel is shown in Algorithm 2.1.

Algorithm 2.1 Simple Kernel

```
__global__ void kernel(float* data) {
    int i = threadIdx.x;
    data[i] = data[i] * 2;
}

/*
gridDim: number of blocks
blockDim: number of threads per block
*/
kernel<<<gridDim, blockDim>>>(data);
```

At the microarchitectural level, the execution of these warps relies on a hardware-driven latency hiding strategy rather than the out-of-order execution and branch prediction mechanisms typically found in CPUs. When an active warp encounters a blocking operation—such as a high-latency global memory access or a data hazard—the hardware Warp Scheduler puts that specific warp into a stalled state. With minimal context-switching overhead, the scheduler selects another warp from the same sub-core that is flagged as ready and dispatches its next instruction.

For this mechanism to mask delays effectively, a high degree of hardware occupancy generally improves performance. If competing execution contexts introduce a heavy computational load that saturates the core execution pipelines, the overall pool of ready warps available within the sub-core may be reduced. As a consequence, concurrent threads can experience measurable execution delays because the underlying hardware scheduler cannot find alternative ready warps to effectively hide pipeline stalls. Furthermore, memory pipeline efficiency is strictly dependent on memory coalescing. When threads within a warp target contiguous global memory addresses, the hardware coalesces the requests into a single transaction. Conversely, pseudo-random or non-coalesced access patterns force multiple independent memory transactions, increasing pressure on internal fetch pipelines and amplifying overall resource contention.

2.3. Texture Units and TEX Pipeline

Modern GPUs include specialized hardware units designed to efficiently handle memory accesses with spatial locality, known as Texture Units. Originally introduced to accelerate texture mapping in graphics applications, these units are now also leveraged in general-

purpose computing due to their optimized caching mechanisms and dedicated data paths.

Texture Units operate through a dedicated pipeline, commonly referred to as the TEX pipeline. This pipeline is responsible for processing texture fetch requests issued by threads and returning the requested data, often applying additional operations such as address calculation, filtering, and format conversion.

In CUDA, texture memory is a read-only memory area that is optimized for certain access patterns. When a thread fetches a texture, the request goes through the TEX pipeline instead of the usual load/store path. This lets the hardware take advantage of spatial locality by using special caches, which speeds up memory access when threads access nearby memory locations.

The TEX pipeline is tightly integrated with the Streaming Multiprocessor. Each SM can issue texture requests, which are then handled by the Texture Units. These requests are typically generated at the warp level, allowing the hardware to coalesce or optimize memory accesses from threads within the same warp.

Another important aspect is caching. Texture Units are implemented with a dedicated shared cache, often called Texture Cache. This cache is optimized for read-only access and spatial locality, making it particularly effective when threads access neighboring data elements. Unlike standard linear data caches designed for temporal and one-dimensional (1D) locality, the Texture Cache hardware is explicitly tailored for two-dimensional (2D) or three-dimensional (3D) geometric proximity, significantly accelerating memory access times when threads target neighboring elements in a spatial grid. Under these conditions, texture fetches may achieve higher cache hit rates.

However, the use of Texture Units is not always beneficial. Their performance advantage depends heavily on the access pattern. Irregular or random memory accesses may not benefit from the texture cache and can result in performance comparable to or worse than standard global memory loads.

Because the Texture Units share components of the unified L1/Texture cache subsystem of the SM, their throughput is susceptible to cross-pipeline interference. Heavy memory-intensive applications, such as deep learning workloads executing large-scale dense matrix computations, generate substantial traffic across the entire GPU memory hierarchy. This activity can increase pressure on shared memory system resources and cache hierarchies within the SM. Consequently, even if a running workload does not explicitly execute graphics or texture instructions, the systemic pressure it exerts on the shared memory subsystem may introduce measurable latency variations into the texture fetch operations

executed by concurrent threads sharing the same hardware ecosystem.

2.4. Asynchronous Execution in GPUs

Recent GPUs support asynchronous execution, allowing instructions to be issued without requiring immediate completion. This is achieved by overlapping computation and memory operations, enabling the system to hide memory latency and improve overall throughput.

Starting from NVIDIA Ampere architecture, CUDA provides the `cp.async` instruction [11], which enables asynchronous data movement from global memory to shared memory. In legacy GPU architectures, moving data from global memory into software-managed shared memory required a rigid, synchronous two-stage process: threads executed a global load to move data into the sub-core's Register File, followed by a local store to write it into Shared Memory. This synchronous approach blocked a portion of the Register File and stalled warp execution during the transit. Unlike standard load instructions, `cp.async` does not stall execution while waiting for data to be fetched. Instead, it initiates a memory transfer that is handled by a dedicated hardware pipeline, bypassing the register file entirely and moving data directly from global memory to shared memory. This allows the issuing threads to continue execution without stalling.

Asynchronous copy operations are organized into groups. Each issued `cp.async` instruction contributes to a group of pending memory operations, which can be explicitly finalized using the `cp.async.commit_group` instruction. This mechanism allows multiple memory transfers to be batched together and submitted to the hardware for execution.

To ensure correctness, explicit synchronization is required to guarantee that the transferred data is available before being consumed. This is achieved through the `cp.async.wait_group` instruction, which blocks execution until all previously issued asynchronous operations in the specified group have completed.

Since shared memory is explicitly managed and has lower latency than global memory, it provides an efficient staging area for data accessed by threads after synchronization. Although the internal implementation of the asynchronous copy mechanism is largely undocumented, `cp.async` operations ultimately rely on shared components of the GPU memory hierarchy. Consequently, their completion latency may be affected by contention generated by concurrently executing workloads that heavily utilize memory resources. When execution reaches a `cp.async.wait_group` instruction, the issuing warp stalls until the outstanding asynchronous transfers have completed. Experimental observations pre-

sented later in this thesis indicate that the duration of this stall can vary depending on the activity of co-resident workloads. These timing variations may therefore be exploited as a side-channel signal for workload fingerprinting.

3 | Related Work

In recent years, the scientific community has exhibited a steadily increasing interest in GPUs. Although the general principles governing GPU hardware operation are relatively well understood, manufacturers typically refrain from disclosing comprehensive details regarding the implementation of specific architectures. This limited transparency has raised concerns about the security of these components from both software and hardware perspectives and motivated researchers to investigate these issues in more depth. Nevertheless, research in this area remains at a relatively early stage of development.

3.1. Covert-Channel Attacks

Early work on GPU covert channels focused on identifying shared microarchitectural resources that could be exploited to establish a communication channel between two co-resident processes. Ahn et al. [1] propose a covert channel based on the shared on-chip interconnect of NVIDIA GPUs. Through reverse engineering of the GPU's on-chip network topology, the authors identify how contention for shared interconnect bandwidth between neighboring cores can be systematically exploited.

Subsequent work has extended the scope of GPU covert channels to target different components and deployment scenarios. Zhang et al. [14] focus on the Translation Lookaside Buffer (TLB) of NVIDIA GPUs, reverse engineering its organization across multiple GPU generations and discovering that NVIDIA's Multi-Instance GPU (MIG) feature — which claims full memory system partitioning for secure cloud sharing — does not partition the last-level TLB. Exploiting this design flaw, the authors construct a covert channel capable of exfiltrating data across MIG-enforced isolation boundaries, achieving up to 31 kbps with approximately 99.8% accuracy on a commercial cloud platform.

Miao et al. [6] further expand the potential attack surface by examining the GPU uncore—elements like the video encoder, video decoder, and memory controller—as a source of covert and side-channel information leakage. They show that these typically neglected components can, in fact, be leveraged to construct covert channels and deduce informa-

tion about co-located workloads, even when isolation schemes such as MPS and MIG are in place.

More recently, Baddour et al. [2] propose an improved micro-architectural covert channel attack on GPUs, building on prior findings to achieve higher transmission reliability and bandwidth. Taken together, these works demonstrate that the GPU attack surface extends well beyond the cache hierarchy, encompassing interconnects, address translation structures, and peripheral hardware units.

3.2. Fingerprinting Attacks

While covert channel attacks focus on establishing a communication channel between two colluding parties, a related but distinct threat is that of workload fingerprinting, where an unprivileged attacker attempts to infer information about an unaware victim's activity by observing the effects of resource contention.

Dutta et al. [3] investigate the vulnerability of NVIDIA DGX multi-GPU systems to both covert and side-channel attacks. By reverse engineering the interconnected cache hierarchy of these machines, the authors show that an attacker on one GPU can cause contention on the L2 cache of another GPU over NVLink. Leveraging this capability, they develop a Prime+Probe attack that recovers the cache access pattern of a victim workload running on a remote GPU, and demonstrate that these patterns can be used to fingerprint the application with high accuracy. Additionally, they present a proof-of-concept attack capable of extracting hyperparameters of machine learning workloads, showing that side-channel leakage can reveal not only the identity of an application but also details of its internal configuration.

Nazaraliyev et al. [8] explore a different attack vector, exploiting the Row Hammer Mitigation (RFM) mechanism introduced in recent DRAM generations as an unintended side channel. Their work demonstrates that the RFM mechanism, originally designed to improve memory reliability, introduces observable timing variations that can be leveraged to fingerprint GPU workloads, representing a novel and previously unexplored source of microarchitectural leakage.

3.3. Cache-Based Attacks

Cache-based attacks constitute one of the most well-studied classes of microarchitectural side-channel attacks. In the CPU domain, foundational work by Osvik et al. [12] introduced the Prime+Probe technique, demonstrating that an unprivileged process can infer

the memory access patterns of a victim by monitoring cache contention, enabling the recovery of cryptographic keys from implementations such as AES in OpenSSL. This line of work established the core principle that shared hardware state can leak information across isolation boundaries, a principle that has since been extended to the GPU domain.

Zhang et al. [15] introduce Invalidate+Compare, a timer-independent GPU cache attack primitive that avoids one of the main requirements of conventional timing-based cache attacks: the availability of a high-resolution timer. By relying instead on cache invalidation and comparison operations, the authors demonstrate that cache-based attacks on GPUs remain feasible even when timing mechanisms are restricted or unavailable, significantly broadening the conditions under which such attacks can be carried out.

3.4. Discussion of Limitations

The body of work surveyed in the preceding sections has established that GPU side-channel and covert-channel attacks are a practical security concern across a wide range of hardware components and deployment scenarios. However, several limitations persist in the existing literature.

First, most proposed attacks target a single hardware component in isolation — whether the on-chip interconnect, the TLB, the L2 cache, or peripheral units such as the video encoder. While each of these works demonstrates leakage from its respective resource, none explores whether signals from multiple independent components can be combined to improve the robustness or granularity of workload inference. In practice, different workloads may produce stronger or weaker contention signatures on different hardware resources, and a classifier that relies on a single signal source may fail to distinguish between workloads that exhibit similar behavior on that particular component.

Second, the fingerprinting attacks proposed in prior work — most notably Dutta et al. [3] — operate in multi-GPU settings over interconnects such as NVLink, a scenario that differs substantially from the more common case of multiple workloads sharing a single physical GPU. The applicability of these techniques to single-GPU, bare-metal environments has not been thoroughly explored.

Third, to the best of our knowledge, no prior work has investigated either the Texture Units or the asynchronous global-to-shared memory copy pipeline introduced in NVIDIA's Ampere architecture as sources of side-channel leakage. Both components provide dedicated hardware paths that are shared among co-resident kernels, yet neither has been studied in the context of microarchitectural attacks.

This thesis addresses these gaps through TEXAS, a contention-based side-channel attack that targets two distinct components of the NVIDIA GPU microarchitecture — the Texture Units and the asynchronous copy pipeline — and combines their signals into a unified dual-stream classification framework to improve workload fingerprinting accuracy.

4 | Proposed Attack

We present TEXAS, a contention-based side-channel attack that exploits shared microarchitectural resources of NVIDIA GPUs to infer information about co-resident workloads. TEXAS deploys two spy kernels, each targeting a distinct hardware pipeline: the Texture Units pipeline (TEX spy kernel) and the asynchronous global-to-shared memory copy pipeline (Async spy kernel). We begin with a high-level overview of the attack, outlining its core intuition and overall procedure. We then describe the design of each spy kernel in detail, followed by the sequential execution strategy and feature fusion mechanism that constitute the operational core of TEXAS. Finally, we discuss implementation details and practical considerations.

4.1. Threat Model

In this work, we consider a scenario in which an attacker and a victim share the same physical GPU on a bare-metal system, with no virtualization or containerization layer between them. The victim runs its workload on the GPU, while the attacker has user-level access to the CUDA runtime and can independently launch kernels on the same device. This setting reflects common deployment scenarios such as shared workstations and multi-user research servers, where multiple applications may run concurrently on the same hardware.

It is important to clarify what "concurrent execution" means in this context. On a physical GPU, two kernels can only execute truly in parallel if they are scheduled on distinct Streaming Multiprocessors at the same time, and this depends on the residual SM occupancy left by the victim. In practice, without explicit co-execution mechanisms such as NVIDIA's Multi-Process Service (MPS), the GPU scheduler may interleave the two kernels through time-slicing at the SM level or through context switches at the GPU level. However, even under this non-strictly-simultaneous scheduling model, contention on shared resources — such as the L2 cache, the TEX pipeline, and the asynchronous copy pipeline — remains observable, because these resources are accessed by both kernels within overlapping time windows. TEXAS does not require the spy and victim kernels to

be scheduled on the same SM simultaneously; it relies only on the fact that both kernels compete for the same shared hardware paths during their respective execution windows.

The attacker's goal is to identify the workload currently being executed by the victim on the GPU. The classification is structured hierarchically: at the first level, the attacker aims to determine the general category of the workload, such as whether the victim is running a transformer-based model, a convolutional neural network, a recurrent neural network, a diffusion model, or a cryptocurrency mining algorithm. Once the category has been identified, the attacker attempts to refine the classification further, distinguishing between specific model families within that category, and, where possible, identifying the exact model being executed. Additionally, for convolutional neural networks, the attacker attempts to infer architectural properties of the victim model, such as the number of convolutional layers and pooling layers, through regression on the collected timing traces. The attacker does not attempt to extract model parameters, training data, or intermediate computation results.

The attacker is assumed to have no elevated privileges on the system. They can launch CUDA kernels, allocate GPU memory, and use standard timing mechanisms such as the `clock64()` instruction [10], all of which are available to any unprivileged CUDA user. The attacker has no direct access to the victim's memory, kernel code, or scheduling metadata. The attacker does not require prior knowledge of which specific model is being executed, nor does it need to modify the victim's execution environment.

A key assumption of the attack is that the attacker has access to the same GPU model as the victim. Since the timing characteristics of hardware resources vary across different GPU architectures, the classifier must be trained on latency traces collected from the same type of GPU that the victim is using. For example, a classifier trained on data collected from an NVIDIA RTX 4070 cannot be directly applied to traces collected from a different GPU model. This assumption is realistic in scenarios where the attacker knows or can determine the hardware configuration of the shared system, which is often the case in multi-user workstations and research servers.

TEXAS is carried out in three phases. In the first phase, the attacker collects latency measurements on their own system by running the two spy kernels alongside known workloads, building a labeled dataset of timing traces for each target model category and model. This data is then used to train a set of classifiers. In the second phase, the attacker runs the spy kernels alongside the victim's unknown workload on the shared GPU, collecting timing traces under the same conditions. In the third phase, the attacker feeds the collected traces into the classifiers trained during the first phase to infer the victim's

workload. The classification proceeds hierarchically, first identifying the general workload category and then attempting finer-grained distinctions where possible. TEXAS is passive in the sense that it does not interfere with the victim's execution beyond the inherent resource contention caused by co-execution on the shared GPU.

4.2. The TEXAS Attack

TEXAS exploits a fundamental property of modern GPU microarchitectures: shared hardware resources create observable interference between co-resident kernels. When multiple workloads compete for the same resource — such as a memory pipeline or a specialized functional unit — contention arises, leading to measurable variations in execution latency. By deliberately stressing specific shared hardware paths and measuring the resulting latency variations, TEXAS can detect the presence and characterize the behavior of a concurrent victim workload.

To maximize the discriminative power of the collected signals, TEXAS employs two spy kernels, each targeting a distinct component of the GPU memory subsystem. The TEX spy kernel targets the Texture Units and their associated caching hierarchy by issuing texture fetch operations and measuring the time required to service each request. The Async spy kernel targets the asynchronous global-to-shared memory copy pipeline, available on GPUs with compute capability 8.0 and above, by issuing asynchronous copy operations and measuring the time the hardware takes to complete them. Both spy kernels rely on the same underlying principle: when a victim kernel concurrently accesses the same hardware path, the spy observes an increase in operation latency that would not occur in isolation.

A key design decision of TEXAS is that the two spy kernels are never executed simultaneously. Running them in parallel would introduce mutual interference, as each kernel generates memory traffic that could corrupt the latency signal of the other. Instead, TEXAS collects the two timing traces sequentially — each spy kernel runs in a dedicated session alongside the victim's workload — and fuses the resulting signals at the classification stage. This sequential execution strategy isolates the contribution of each pipeline, while the subsequent feature fusion allows the classifier to exploit complementary information from both: a workload that produces a weak contention signature on the texture pipeline may produce a strong one on the asynchronous copy pipeline, and vice versa.

At a high level, each spy kernel repeatedly issues memory operations using pseudo-random access patterns, records per-operation timing information, and aggregates the measurements to produce a robust latency signal. The traces collected by the two spy kernels

are then concatenated into a joint feature representation and processed by a dual-stream classifier. The detailed design of each spy kernel, and the feature fusion mechanism, are presented in the following sections.

4.2.1. TEX Spy Kernel

The TEX spy kernel targets the Texture Units and their associated caching hierarchy within the Streaming Multiprocessor. The core idea is to continuously issue texture fetch operations via the TEX pipeline and measure the latency of each fetch using the GPU's cycle counter. When a victim kernel executes concurrently on the same GPU and utilizes the texture units or competes for memory bandwidth, the spy observes an increase in fetch latency relative to an idle baseline.

The spy kernel allocates a one-dimensional texture object mapped onto a region of device memory. In each iteration, the kernel computes a pseudo-random index using a Linear Feedback Shift Register (LFSR) and performs a `tex1Dfetch` operation at that index. The use of pseudo-random access patterns avoids sequential or strided accesses, reducing the likelihood of exploiting spatial locality in the texture cache. This design choice increases the probability that each fetch traverses the full TEX pipeline and remains sensitive to contention from co-resident workloads.

Each texture fetch operation's duration is individually measured using the `clock64()` instruction, which provides the current cycle counter value for each SM. The latency is then determined by calculating the difference between the cycle counter values obtained immediately prior to and following the texture fetch. Given that individual measurements may be subject to noise stemming from scheduling fluctuations and other microarchitectural influences, the kernel aggregates measurements from all 32 threads within the warp and subsequently computes the median latency. The median is employed as a more reliable estimator of the underlying latency distribution, particularly in the presence of outliers, when compared to the arithmetic mean.

The spy kernel operates in a streaming fashion. Instead of executing all iterations within a single long-running kernel launch, the total number of iterations is partitioned into multiple blocks. After each block completes, the timing data are transferred back to host memory and appended to a CSV file. This design avoids allocating a large device-side buffer to store the entire trace and enables continuous, long-duration measurement without interruption.

4.2.2. Async Spy Kernel

The Async spy kernel targets the asynchronous global-to-shared memory copy pipeline, introduced in NVIDIA's Ampere architecture (compute capability 8.0 and above). This pipeline, accessed through the `cp.async` instruction, provides a dedicated hardware path for moving data from global memory directly to shared memory, bypassing the register file. Since this pipeline is a shared resource among all kernels executing on the same GPU, any memory traffic generated by a co-resident victim kernel increases the time required for the spy's asynchronous copies to complete — a variation that TEXAS exploits to characterize the victim's behavior.

The spy kernel allocates a 256 MB region of global memory and a shared memory buffer within each block. At each iteration, the kernel computes a pseudo-random global memory offset using an LFSR and issues four consecutive `cp.async.cg.shared.global` instructions, each copying 16 bytes from the computed global address to the shared memory buffer. Each copy is immediately committed using `cp.async.commit_group`. After all four copies have been issued, the kernel measures the time required for all pending operations to complete by recording the cycle counter before and after a `cp.async.wait_group 0` instruction.

This measurement strategy is a key aspect of the spy kernel design. Rather than timing each individual copy, the spy times the synchronization point — the moment at which the hardware guarantees that all previously issued asynchronous transfers have completed. This captures the aggregate effect of contention on the asynchronous copy pipeline: if the victim is generating memory traffic that competes for bandwidth or pipeline slots, the wait time increases. As with the TEX spy kernel, the median across all 32 threads is used to produce a robust per-iteration measurement, and the kernel operates in a streaming block-based fashion for continuous data collection.

4.2.3. Sequential Execution and Feature Fusion

The two spy kernels of TEXAS each produce a distinct timing trace that captures contention on a different hardware pipeline. Since different workload categories may generate stronger contention signatures on one pipeline than the other, combining the information from both traces increases TEXAS's discriminative power beyond what either spy kernel can achieve in isolation.

As discussed in Section 4.2, the two spy kernels are executed sequentially rather than simultaneously. This is a deliberate design choice: running both kernels in parallel would

cause mutual interference, since the memory traffic generated by one spy kernel could inflate the latency measurements of the other, corrupting both signals. By running each spy kernel in a dedicated session alongside the victim's workload, TEXAS obtains two clean, independent timing traces that faithfully reflect the contention on their respective pipelines. The order in which the two sessions are conducted is irrelevant, provided that the victim is running the same type of workload throughout both.

4.2.3.1. Data Preprocessing

Before classification, each raw timing trace is preprocessed into a sequence of fixed-length windows. The raw trace is segmented using a sliding window of $W = 256$ measurements with a stride of $s = 16$ samples between consecutive windows, producing a dense set of overlapping subsequences. Each window is independently standardized via Z-score normalization: the window mean is subtracted and the result is divided by the window standard deviation,

$$\tilde{x}_i = \frac{x_i - \mu_w}{\sigma_w + \epsilon}$$

where μ_w and σ_w are the mean and standard deviation of the window and $\epsilon = 10^{-8}$ is a small constant for numerical stability. This per-window normalization makes the model invariant to absolute latency baselines — which vary across GPU models and operating conditions — while preserving the structural shape and micro-variability of the signal that carry discriminative information about the co-resident workload.

To prevent data leakage across the train/validation split, the dataset is partitioned at the CSV file level rather than at the window level. A contiguous block of measurements is first extracted from each CSV file, then split chronologically at the 80% mark: windows derived from the first portion go into the training set, and windows derived from the second portion go into the validation set. This guarantees that no two windows from the same temporal segment appear simultaneously in both splits, avoiding the scenario where adjacent windows — which share most of their content due to the small stride — could artificially inflate validation accuracy. For final inference evaluation, a completely separate set of CSV files, produced by independent execution runs not used during training, is reserved as a held-out test set.

4.2.3.2. Dual-Stream Architecture

The dual-stream classifier consists of two independent parallel branches, one per spy kernel, that process the TEX and Async window sequences simultaneously. Each branch is a two-layer Bidirectional LSTM with a hidden size of 64 units per direction. Since the input to each LSTM is a univariate time series (a single latency value per timestep), the input dimension is 1. The bidirectional processing doubles the effective hidden size, so each branch produces a 128-dimensional feature vector at the final timestep, obtained by taking the hidden state of the last timestep from the concatenated forward and backward outputs.

The feature vectors from the two branches are then concatenated into a single 256-dimensional joint representation. This fused vector is passed through a *fusion block* consisting of: a fully connected layer that projects the 256-dimensional input to a 128-dimensional space with a ReLU activation; a Batch Normalization layer to stabilize internal covariate shift during training; a Dropout layer with $p = 0.3$ to mitigate overfitting; and a final linear output layer with C units, where C is the number of target classes. Formally, for a pair of input sequences $\mathbf{x}_{\text{tex}}$ and $\mathbf{x}_{\text{async}}$, each of length W, the forward pass computes:

$$\begin{aligned}
\mathbf{f}_{\text{tex}} &= \text{BiLSTM}_{\text{tex}}(\mathbf{x}_{\text{tex}})[-1] \in \mathbb{R}^{128} \\
\mathbf{f}_{\text{async}} &= \text{BiLSTM}_{\text{async}}(\mathbf{x}_{\text{async}})[-1] \in \mathbb{R}^{128} \\
\mathbf{h} &= \text{ReLU}(\mathbf{W}_f\,[\mathbf{f}_{\text{tex}} \,\|\, \mathbf{f}_{\text{async}}] + \mathbf{b}_f) \in \mathbb{R}^{128} \\
\hat{\mathbf{y}} &= \mathbf{W}_o\,\text{Dropout}(\text{BN}(\mathbf{h})) + \mathbf{b}_o \in \mathbb{R}^{C}
\end{aligned}$$

where $\|$ denotes concatenation and $[-1]$ denotes extraction of the last-timestep hidden state. Class predictions are obtained by applying argmax to the output logits $\hat{\mathbf{y}}$.

The two BiLSTM branches are deliberately kept independent — they share no weights and no hidden state — so that each branch can specialize in extracting features from its respective pipeline without interference. The fusion occurs exclusively at the feature level, after each branch has independently processed its input sequence. This architecture maintains the interpretability of each signal source while allowing the classifier to exploit complementary information across pipelines.

For the regression tasks in the CNN domain (Section 5.4.1.2), the architecture is identical except that the output layer is replaced with a single linear unit, the loss function is changed from Cross-Entropy to Mean Squared Error, and a ReLU activation is applied to the output to enforce non-negative predictions.

4.2.3.3. Training Procedure

The network is trained using the Adam optimizer with a learning rate of $\eta = 10^{-3}$ and the standard Cross-Entropy loss. Training proceeds for up to 150 epochs with a batch size of 256 window pairs. An early stopping criterion is applied with a patience of 15 epochs, halting training if the weighted F1 score on the validation set does not improve: the model checkpoint with the highest validation F1 is saved and used for all subsequent evaluations. This ensures that the reported results correspond to the best-generalizing model observed during training rather than the final epoch.

4.3. Implementation Details

In this section, we describe the timing mechanism and pseudo-random access generation strategy shared by both spy kernels of TEXAS.

4.3.1. Timing Mechanism

Both spy kernels use the `clock64()` instruction to measure operation latency. This instruction returns a 64-bit cycle counter that is local to each Streaming Multiprocessor: since threads belonging to different SMs observe independent counters, timing comparisons are only meaningful within the same warp, which by definition executes on a single SM.

The two spy kernels differ slightly in how they bracket the timed section. In the Async spy kernel, explicit compiler barriers (`asm volatile("" ::: "memory")`) are inserted immediately before and after the `cp.async.wait_group` instruction. These barriers act as compiler-level reordering fences and prevent the compiler from moving instructions across the timing boundaries; they do not emit hardware memory fence instructions, but this is sufficient because `cp.async.wait_group` already enforces the necessary ordering at the hardware level. In the TEX spy kernel, no explicit barriers are needed: the `tex1Dfetch` instruction is issued directly between two `clock64()` reads, and the CUDA compiler does not reorder texture fetch instructions across surrounding register reads.

At each iteration, all 32 threads in the warp independently measure the latency of their own operation. The 32 resulting values are stored in shared memory, and thread 0 sorts them using a simple comparison-based sort and selects the value at index 16 as the representative measurement for that iteration. Since the array is 0 indexed and contains 32 elements, index 16 corresponds to the upper median of the sorted sequence — the larger of the two central values. The median is preferred over the mean because it is robust to

outliers caused by scheduling jitter, cache effects, or other transient microarchitectural events that may affect individual threads disproportionately.

A further implementation detail concerns the Async spy kernel specifically. After the `cp.async.wait_group` synchronization point, the kernel performs a deliberate XOR reduction over the data copied into shared memory, storing the result in a `volatile` variable. This prevents the compiler from eliminating the asynchronous copy operations as dead code due to the absence of visible side effects, ensuring that the copies are always executed and that the measured wait time reflects genuine hardware activity rather than an optimized-away no-op.

4.3.2. Pseudo-Random Access Generation

Both spy kernels generate memory access indices using a 32-bit Galois Linear Feedback Shift Register (LFSR) with the feedback polynomial `0xD0000001`. At each iteration, the LFSR state is updated with a single shift and conditional XOR operation:

$$\text{lfsr} = (\text{lfsr} \gg 1) \oplus (-(\text{lfsr} \wedge 1) \wedge \texttt{0xD0000001})$$

The polynomial `0xD0000001` is a primitive polynomial over GF(2), which guarantees that the LFSR has a maximal period of $2^{32} - 1$: the register visits every non-zero 32-bit state exactly once before cycling. Since the total number of measurements collected per spy kernel session (five million samples) is far below this period, the access sequence never repeats within a single experiment, avoiding periodic patterns in the latency traces that could bias the classifier.

The LFSR is seeded per-thread using the formula:

$$\text{seed} = \text{threadIdx.x} \times 7919 + 104729 + \text{start\_iter}$$

where `start_iter` is the global iteration offset for the current block and the constants 7919 and 104729 are prime numbers chosen to ensure that seed values are well-separated across threads, reducing the likelihood of overlapping LFSR sequences between different threads within the same warp. The inclusion of `start_iter` in the seed formula further ensures that consecutive blocks start from different positions in the LFSR sequence, preventing different blocks from accessing overlapping regions of the memory buffer and thus avoiding temporal correlations between consecutive measurement windows.

The choice of an LFSR over a standard pseudo-random number generator is motivated by its minimal computational overhead. Since the purpose of each spy kernel is to measure memory latency, index generation must not introduce significant compute load that could itself cause contention or distort the timing measurements. The LFSR requires only two operations per iteration — a shift and a conditional XOR — making its contribution to the overall execution time negligible.

The generated indices are mapped to memory locations differently by each spy kernel according to its access model. In the TEX spy kernel, the index is masked to the texture buffer size using a bitwise AND: `lfsr & (TEX_SIZE - 1)`, where `TEX_SIZE` is 2^{18} elements (1 MB). This size is chosen to exceed the texture cache capacity of a single SM, which is typically in the range of 128 KB on Ampere and Ada Lovelace architectures, ensuring that pseudo-random accesses are unlikely to be served from the texture cache and instead propagate through the broader memory hierarchy. In the Async spy kernel, the index is instead mapped to a 16-element aligned offset within a 256 MB global memory buffer, satisfying the alignment requirements of the `cp.async` instruction. The large buffer size guarantees that random accesses are spread across a region far exceeding any on-chip cache capacity, maximizing the probability that each transfer traverses the full global memory path and remains sensitive to contention generated by the victim workload.

5 | Evaluation

In the following, we examine the experimental phase in greater detail. We first describe the experimental setup and the target applications, and then we present and analyze the results. The experiments were conducted using three distinct GPUs: an NVIDIA RTX 3050, an NVIDIA RTX 4070, and an NVIDIA RTX 5070.

5.1. Experimental Setup

All experiments were executed with user-level privileges. Both the victim and attacker applications were run concurrently on the same GPU, which all users could access directly in a bare-metal configuration.

More precisely, the experiments were carried out in two distinct setups, depending on the GPU model:

- **NVIDIA RTX 4070**: experiments run on a standard Windows 11 desktop machine, where both victim and attacker applications were launched by non-privileged users.
- **NVIDIA RTX 3050 and NVIDIA RTX 5070**: experiments run on a Linux server with an Ubuntu distribution, where all applications were started over SSH with regular user permissions.

In both configurations, the victim and attacker kernels were launched as independent processes without any explicit co-execution mechanism such as NVIDIA MPS. As a consequence, true simultaneous SM-level execution of the two kernels is not guaranteed: depending on the victim's SM occupancy, the GPU scheduler may interleave them through time-slicing rather than executing them in parallel. This behavior is consistent with the threat model described in Section 4.1 and does not invalidate the attack, since contention on the shared hardware pipelines targeted by the spy kernels remains observable even when the two kernels overlap only partially in time.

5.2. Target Workloads

Our evaluation covers five categories of GPU workload: CNNs, Transformers, RNNs, diffusion models and cryptocurrency mining algorithms. These categories were chosen because they represent the most common types of computation encountered in shared GPU environments, and because they exhibit substantially different memory access patterns and pipeline utilization profiles, which makes them a meaningful benchmark for evaluating the discriminative power of the proposed attacks. For each category, we selected multiple models and datasets to avoid overfitting the analysis to a single representative and to assess the generalizability of the attack across variations within the same family. Table 5.1 provides a complete list of the models and datasets used for each category on the NVIDIA RTX 4070. It is important to notice that, for validation on the NVIDIA RTX 3050 and NVIDIA RTX 5070, we only used a subset of this list as described in Section 5.5, so that any differences in classification performance between hardware configurations can be attributed to the GPU architecture rather than to variations in the victim workload.

Table 5.1: Complete list of models and datasets used in the evaluation on the RTX 4070.

Model	Datasets	Phase
CNN		
DenseNet121	CIFAR-10 CIFAR-100 SVHN	Train Inference
DenseNet169	CIFAR-10 CIFAR-100 SVHN	Train Inference
DenseNet201	CIFAR-10 CIFAR-100 SVHN	Train Inference
MobileNet_V2	CIFAR-10 CIFAR-100 SVHN	Train Inference

Model	Datasets	Phase
ResNet-18	CIFAR-10 CIFAR-100 SVHN	Train Inference
ResNet-34	CIFAR-10 CIFAR-100 SVHN	Train Inference
ResNet-101	CIFAR-10 CIFAR-100 SVHN	Train Inference
Transformer		
BERT-base-uncased	IMDB Yelp Polarity AG News	Train Inference
GPT-2	Wikitext-2 IMDB Yelp Polarity AG News	Train Inference
Gemma-2b	Wikitext-2 IMDB Yelp Polarity AG News	Train Inference
Gemma-3-1b	Wikitext-2 IMDB AG News	Train Inference

Model	Datasets	Phase
CodeGemma-2b	Wikitext-2 IMDB AG News	Train Inference
MedGemma-4b	Wikitext-2 IMDB AG News	Train Inference
VaultGemma-1b	Wikitext-2 IMDB AG News	Train Inference
EmbeddingGemma-300m	Wikitext-2 IMDB AG News	Train Inference
PaliGemma-3b	CIFAR-10 CIFAR-100	Train Inference
DeepSeek-Coder-1.3b	Wikitext-2 IMDB AG News	Train Inference
DeepSeek-LLM-7b	Wikitext-2 IMDB AG News	Train Inference
DeepSeek-Math-7b	Wikitext-2 IMDB AG News	Train Inference

Model	Datasets	Phase
Qwen2-VL-2B	Wikitext-2 IMDB AG News	Train Inference
Diffusion		
Stable-Diffusion-v1-5	sdxl-0.9 naruto-blip-captions	Train Inference
Tiny-sd	sdxl-0.9 naruto-blip-captions	Train Inference
RNN		
LSTM	mnist_seq tiny_shakespeare	Train Inference
BiLSTM	mnist_seq tiny_shakespeare	Train Inference
GRU	mnist_seq tiny_shakespeare	Train Inference
Mining		
DaggerHashimoto ETCHash Autolycos KawPow NeoScrypt	—	Active

5.3. Attack Kernels Validation

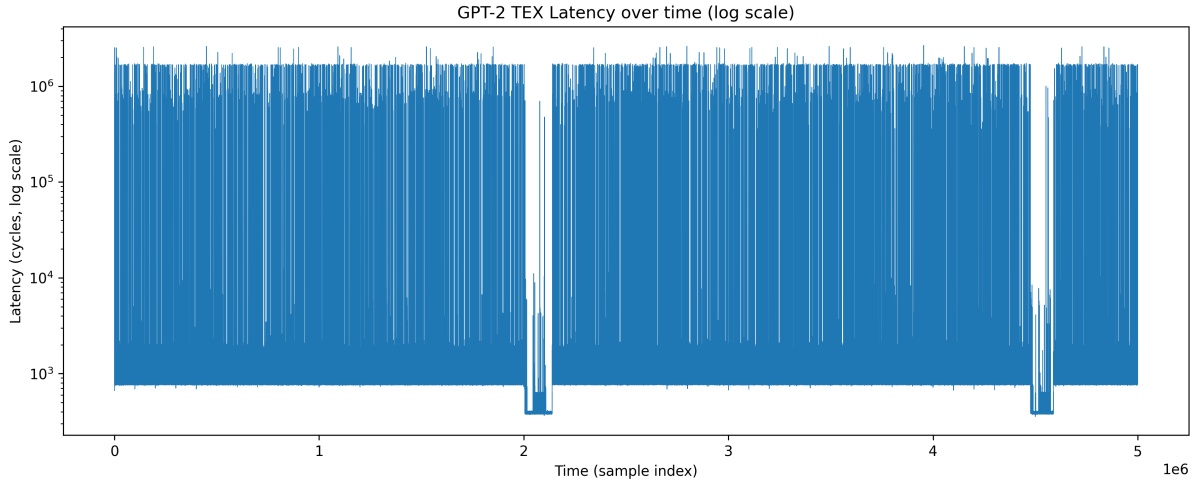
To evaluate our initial hypothesis, we began by concentrating on the NVIDIA RTX 4070 and executed all the models and datasets previously listed in Table 5.1 during both training and inference. In each phase, while the model was executing, we also ran the TEX and Async kernels (individually, never at the same time) and collected five million latency measurements per attack. We then plotted these results to inspect any visual discrepancies between the two attacks.

5.3.1. TEX Attack Kernel

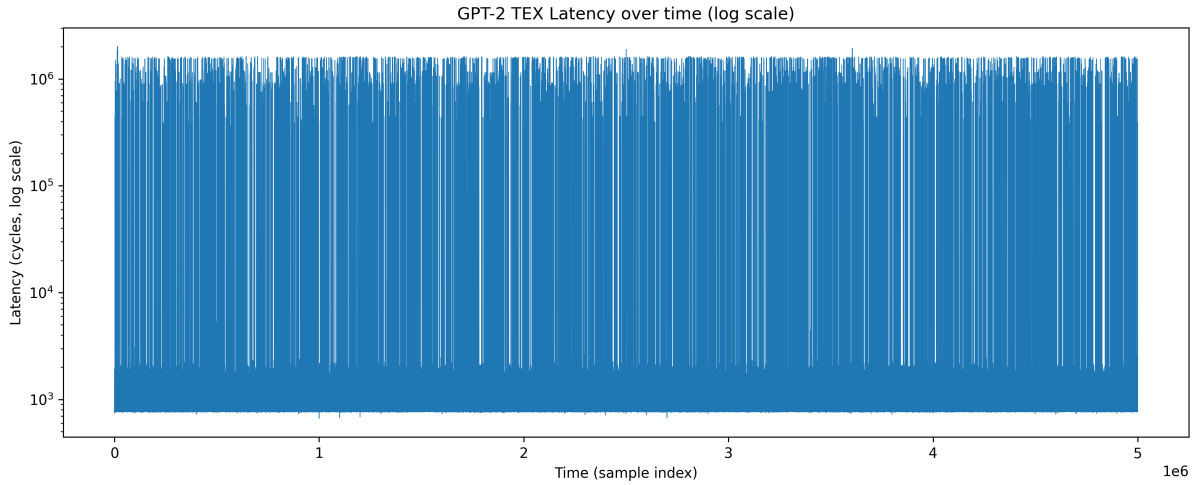
The latency profiles produced by the TEX attack kernel's execution on a variety of workloads hosted on the NVIDIA RTX 4070 are examined in this subsection. The TEX kernel specifically targets the GPU's texture pipeline, generating a continuous stream of texture-fetching operations designed to saturate the dedicated hardware units and stress the underlying cache hierarchy. Due to the extensive number of tests conducted across the various models and execution phases, a highly representative selection of the resulting charts is presented here — specifically showcasing two key profiles per model family — to provide a clear yet comprehensive overview of the observed trends.

Figures 5.1 through 5.4 illustrate the temporal distribution of the five million latency measurements collected during both the training and inference phases of the evaluated Deep Learning models. Furthermore, to evaluate the footprint of the attack outside the machine learning domain, Figure 5.5 contrasts the kernel's effects on two cryptocurrency mining algorithms (`DaggerHashimoto` and `NeoScrypt`).

As observed in the plots, the resulting latency signatures present distinct behaviors depending on the workload type and execution phase.

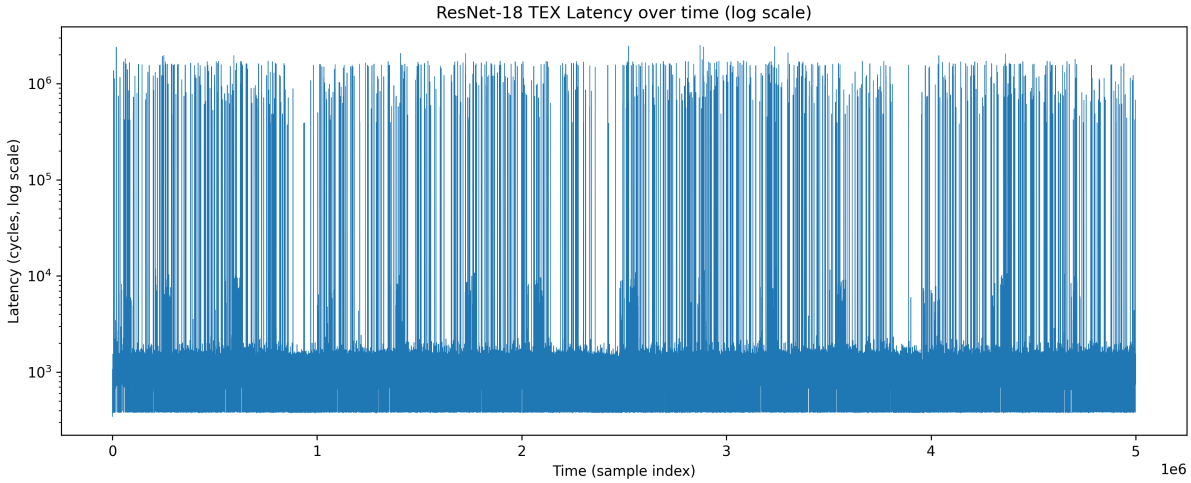


(a) Training execution.

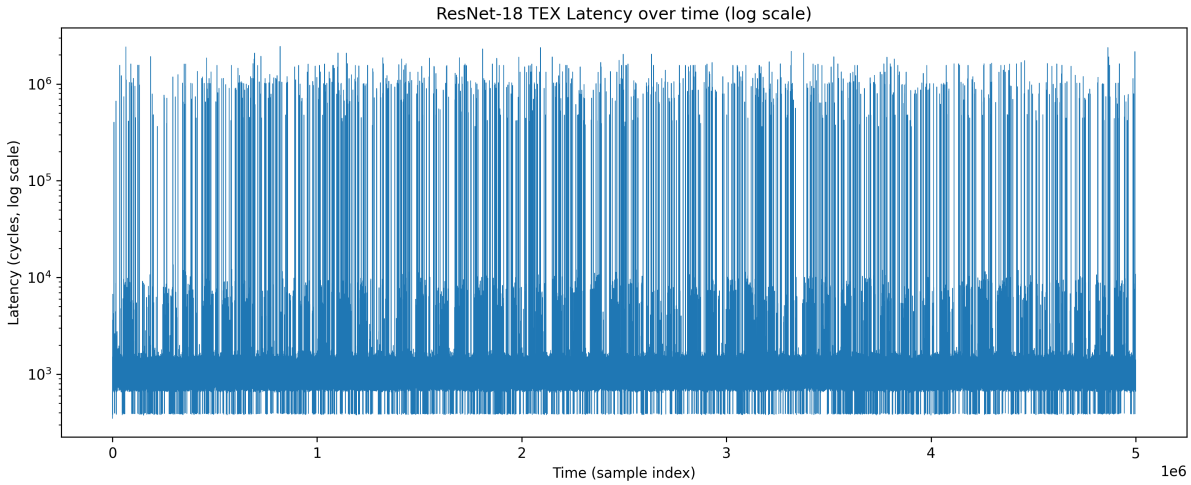


(b) Inference execution

Figure 5.1: Latencies measured on the TEX pipeline for the GPT-2 model running in training and inference on an NVIDIA RTX 4070 with the *ag_news* dataset.

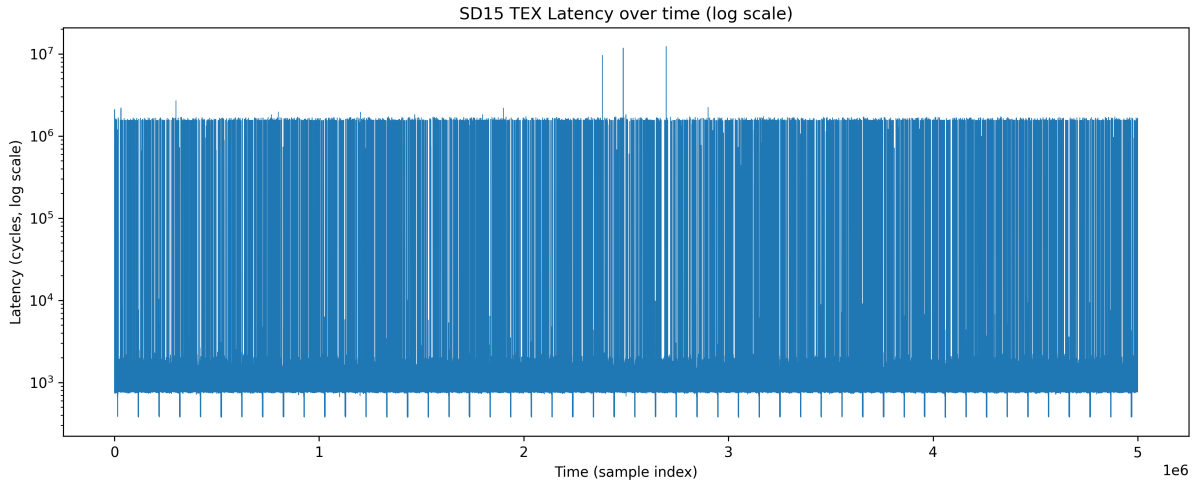


(a) Training execution.

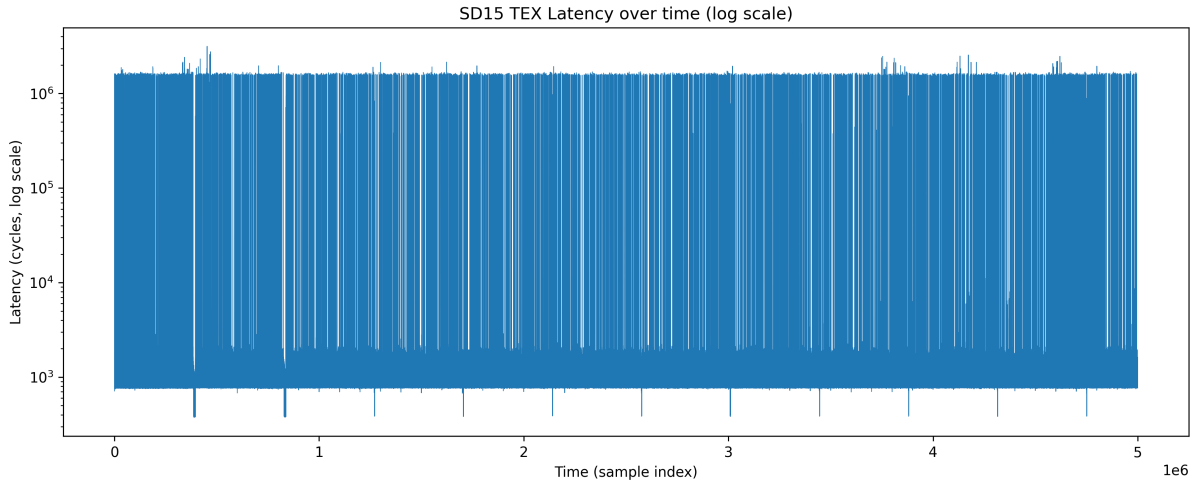


(b) Inference execution

Figure 5.2: Latencies measured on the TEX pipeline for the ResNet-18 model running in training and inference on an NVIDIA RTX 4070 with the *cifar100* dataset.

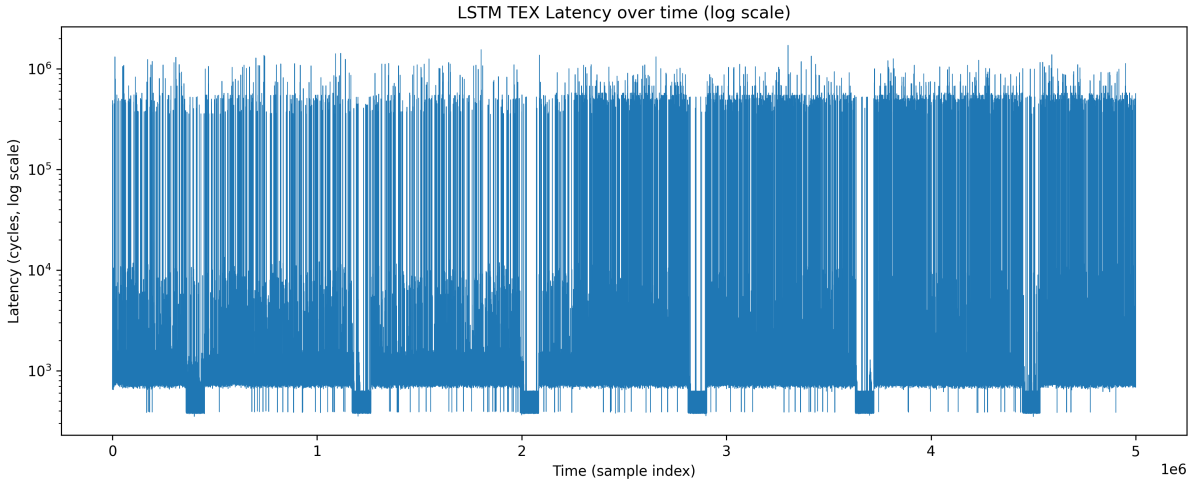


(a) Training execution.

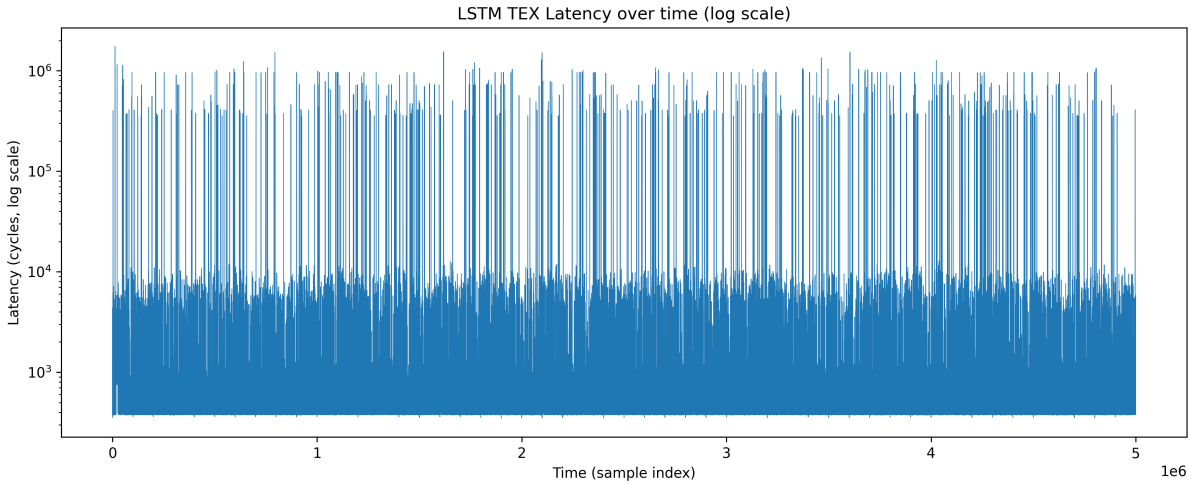


(b) Inference execution

Figure 5.3: Latencies measured on the TEX pipeline for the Stable-Diffusion-v1-5 model running in training and inference on an NVIDIA RTX 4070 with the *naruto-blip-captions* dataset.

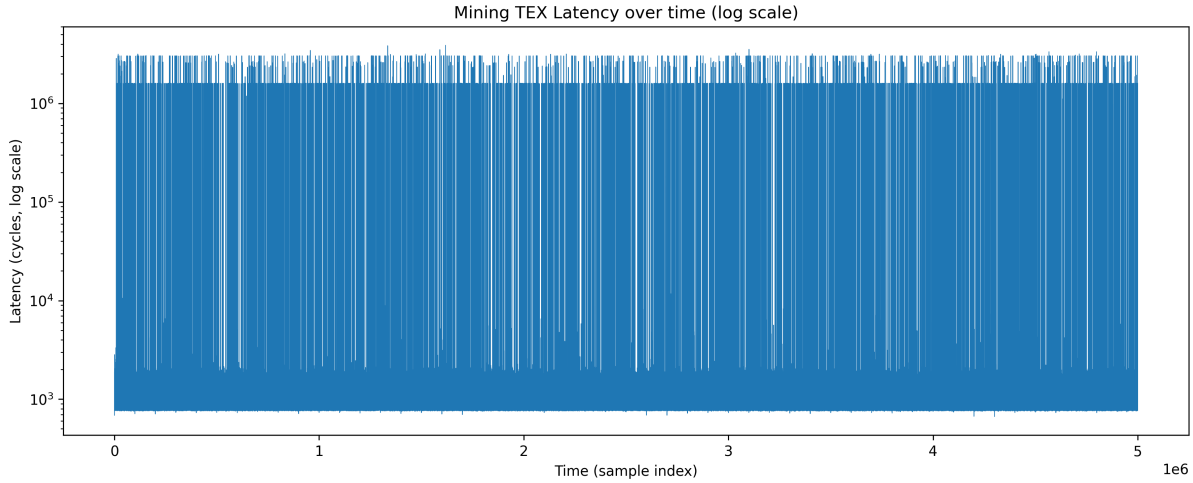


(a) Training execution.

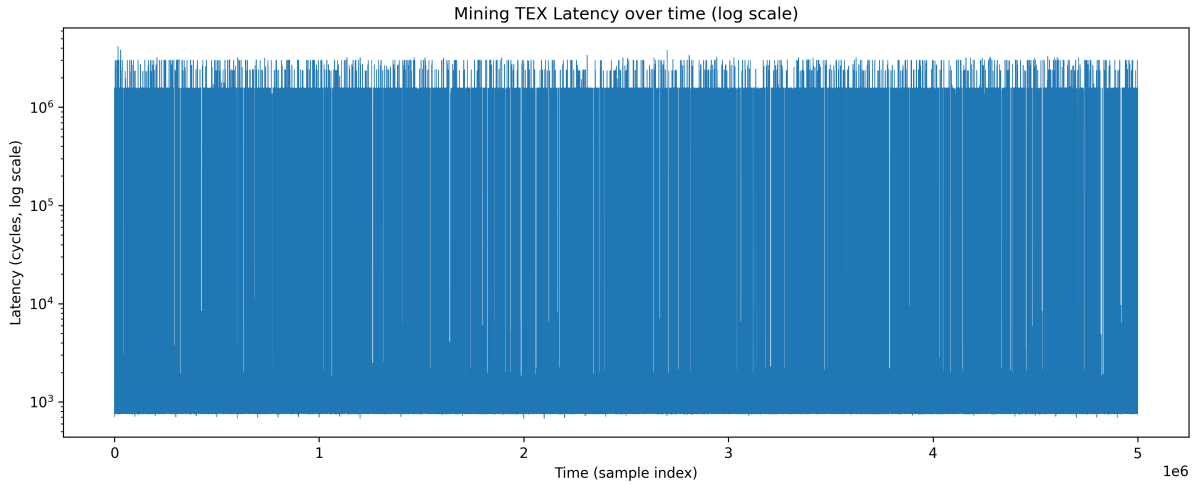


(b) Inference execution

Figure 5.4: Latencies measured on the TEX pipeline for the LSTM model running in training and inference on an NVIDIA RTX 4070 with the *mnist_seq* dataset.



(a) DaggerHashimoto mining execution.



(b) NeoScrypt mining execution.

Figure 5.5: Latencies measured on the TEX pipeline for the mining algorithms `DaggerHashimoto` and `NeoScrypt` on an NVIDIA RTX 4070.

To quantitatively evaluate the fingerprinting potential of these traces, we assess the ability of a Bidirectional Long Short-Term Memory (BiLSTM) classifier to distinguish between the workload categories defined in Section 5.2. For this single-stream experiment, the BiLSTM network is trained and validated exclusively on TEX traces. Since the Diffusion category was incorporated into the evaluation at a later stage of the analysis, it is excluded from the single-stream benchmarks. The evaluation is therefore carried out on the remaining four categories: CNN, Transformer, RNN, and Mining. The TEX-based classifier achieves a weighted F1 score of approximately 0.93 on the validation set. This result demonstrates that the texture pipeline latency alone carries highly discriminative features, providing sufficient information to reliably identify the active workload category. The corresponding training curves are illustrated in Figure 5.6.

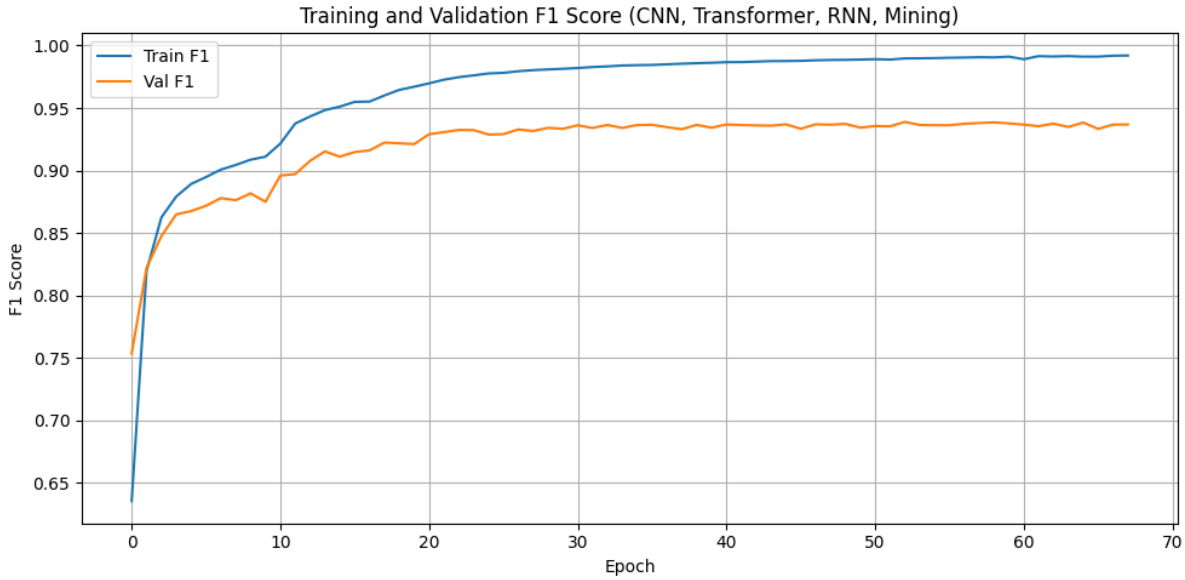


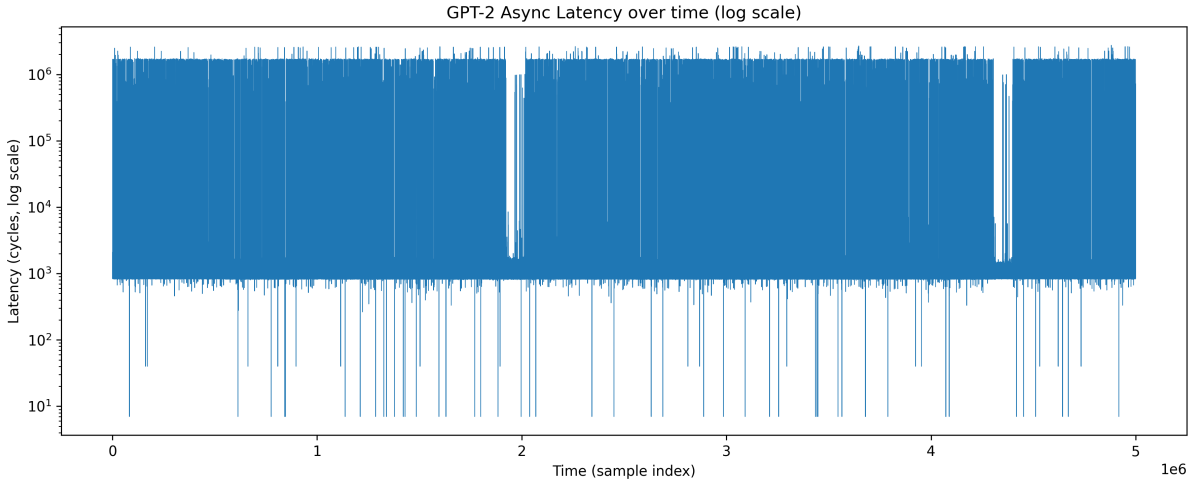
Figure 5.6: F1 score for the single-stream BiLSTM classifier trained on TEX latencies.

5.3.2. Async Attack Kernel

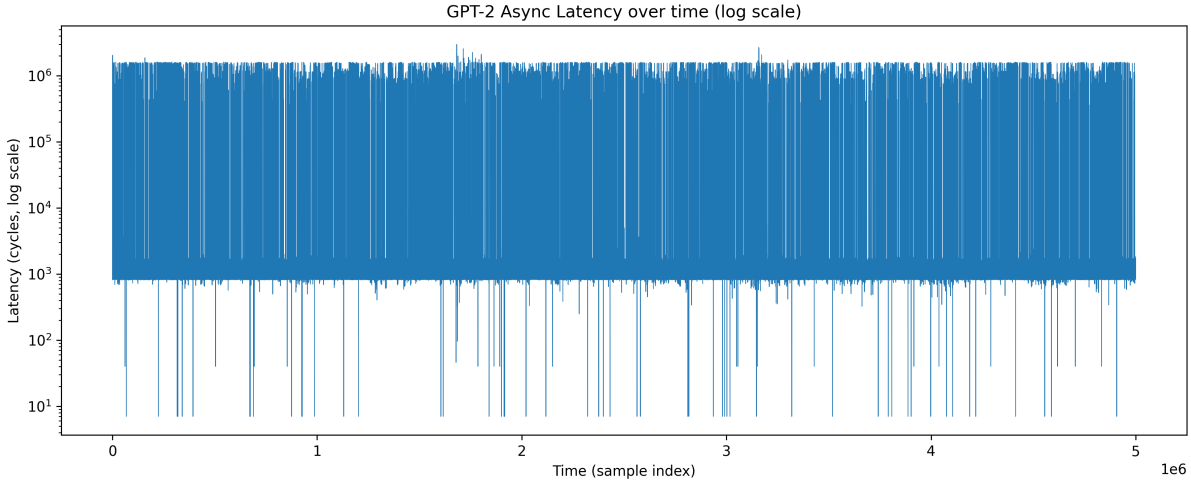
The latency profiles produced by the Async attack kernel's execution on a variety of workloads hosted on the NVIDIA RTX 4070 are examined in this subsection. The Async kernel specifically targets the GPU's asynchronous global-to-shared memory copy pipeline, issuing a continuous stream of concurrent operations designed to induce contention within the shared execution resources. Following the same approach and methodology outlined in Section 5.3.1, a highly representative selection of the resulting charts is presented here — specifically showcasing two key profiles per model — to provide a clear yet comprehensive overview of the observed trends.

Figures 5.7 through 5.10 illustrate the temporal distribution of the five million latency measurements collected during both the training and inference phases of the evaluated Deep Learning models under the Async attack. Furthermore, to evaluate the footprint of this specific attack outside the machine learning domain, Figure 5.11 contrasts the kernel's effects on two cryptocurrency mining algorithms (`DaggerHashimoto` and `NeoScrypt`).

As observed in the plots, the resulting latency signatures present distinct behaviors depending on the workload type and execution phase, highlighting a different impact footprint compared to the TEX kernel.

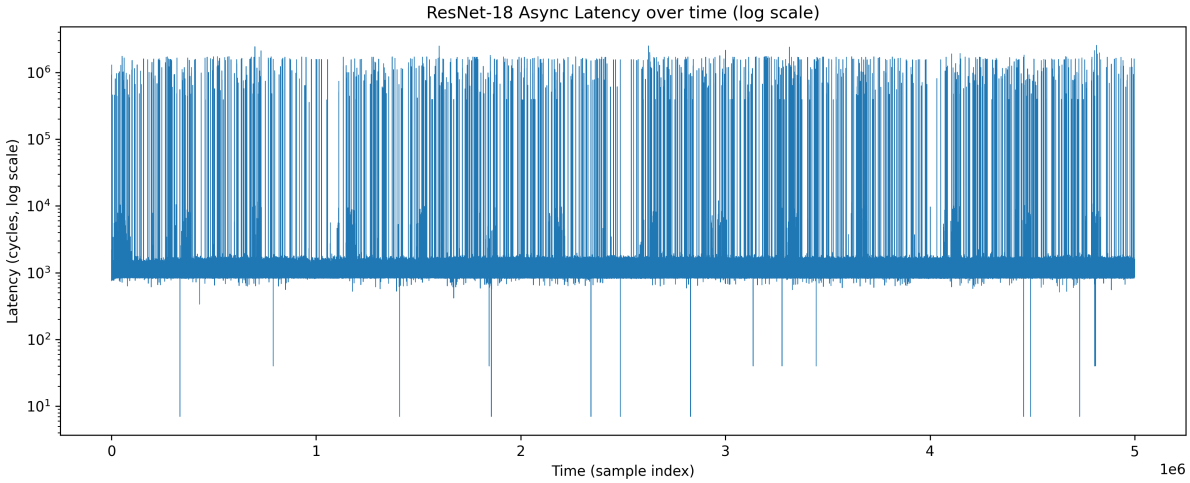


(a) Training execution.

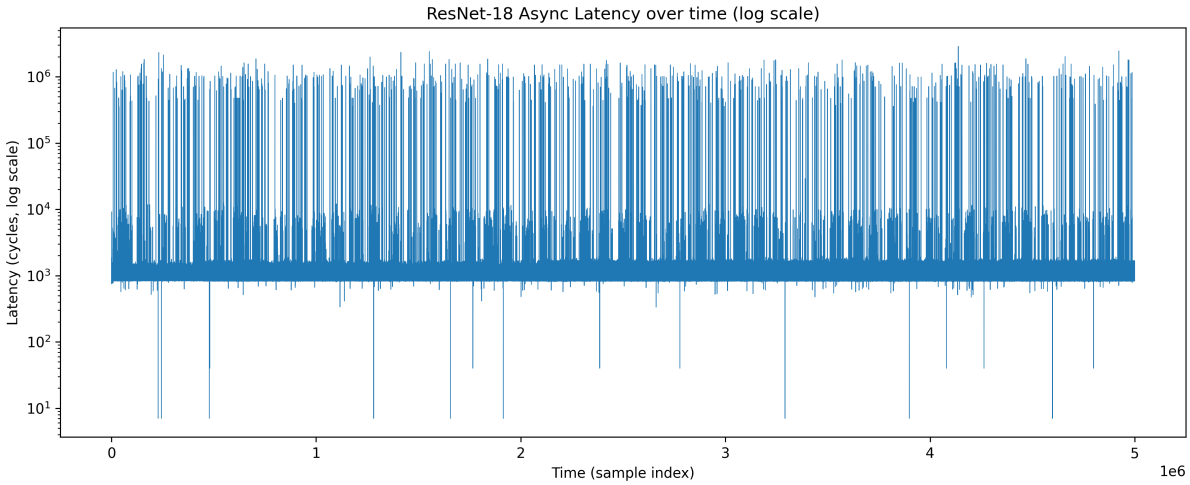


(b) Inference execution

Figure 5.7: Latencies measured on the Async pipeline for the GPT-2 model running in training and inference on an NVIDIA RTX 4070 with the *ag_news* dataset.

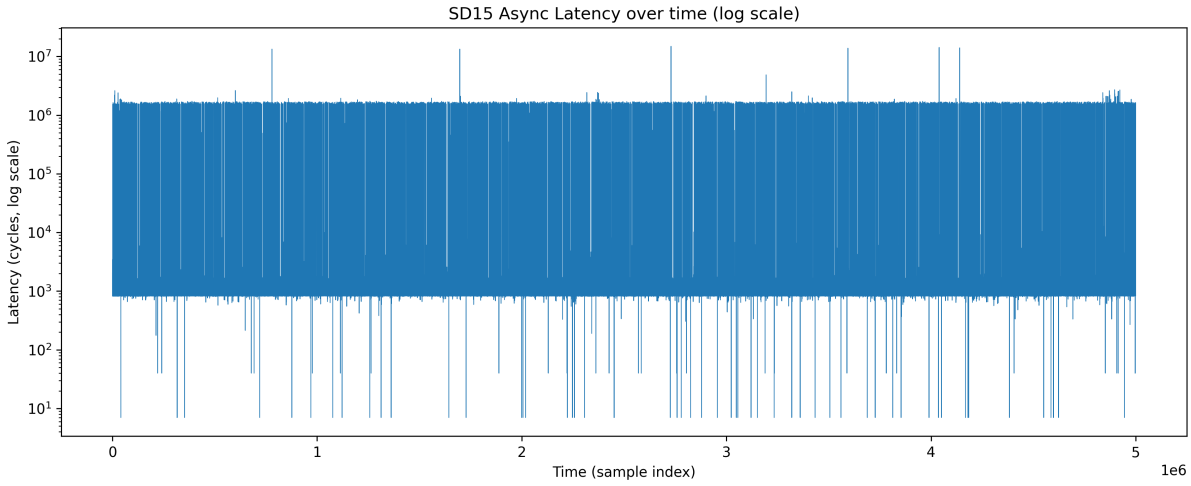


(a) Training execution.

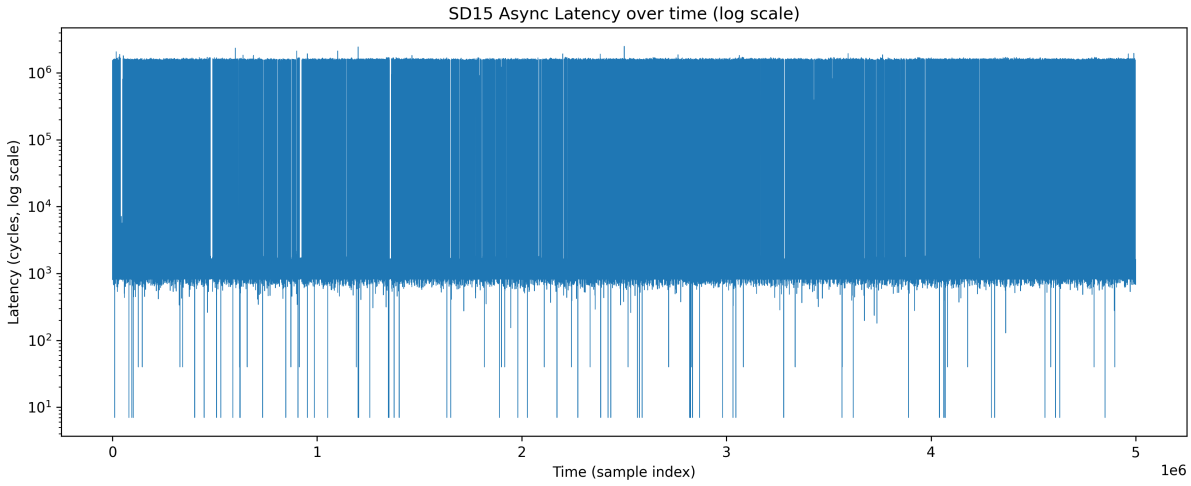


(b) Inference execution

Figure 5.8: Latencies measured on the Async pipeline for the ResNet-18 model running in training and inference on an NVIDIA RTX 4070 with the *cifar100* dataset.

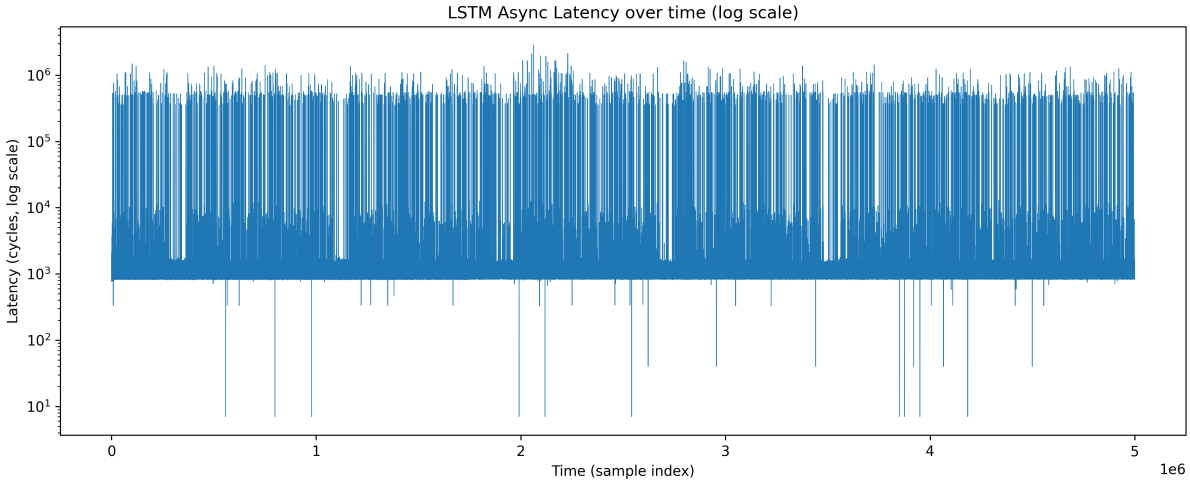


(a) Training execution.

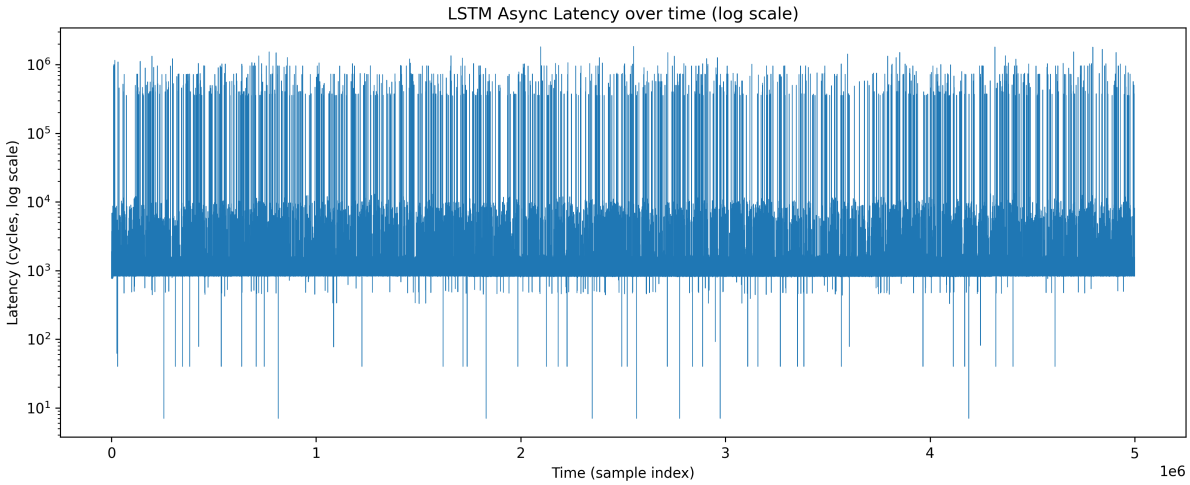


(b) Inference execution

Figure 5.9: Latencies measured on the Async pipeline for the Stable-Diffusion-v1-5 model running in training and inference on an NVIDIA RTX 4070 with the *naruto-blip-captions* dataset.

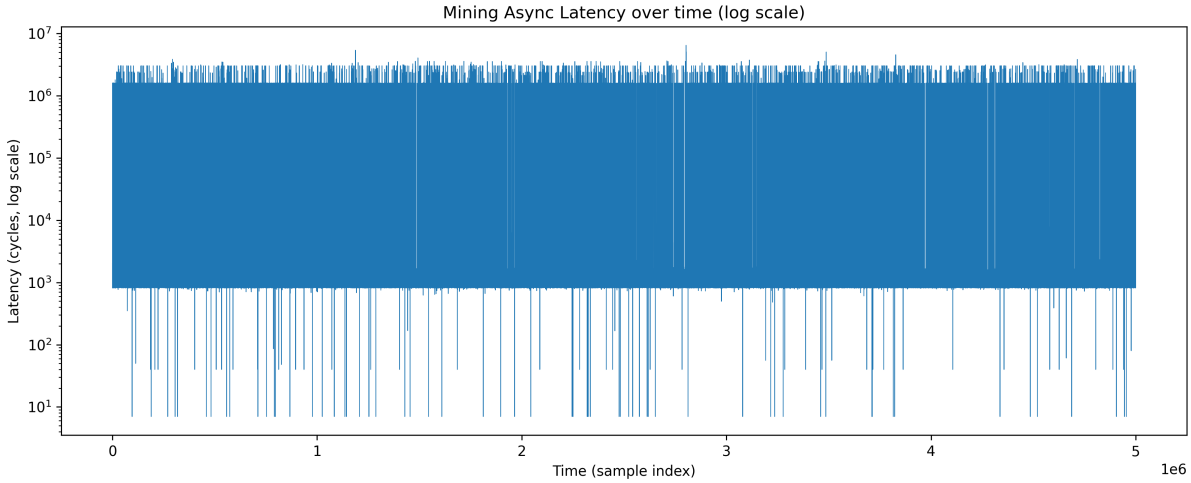


(a) Training execution.

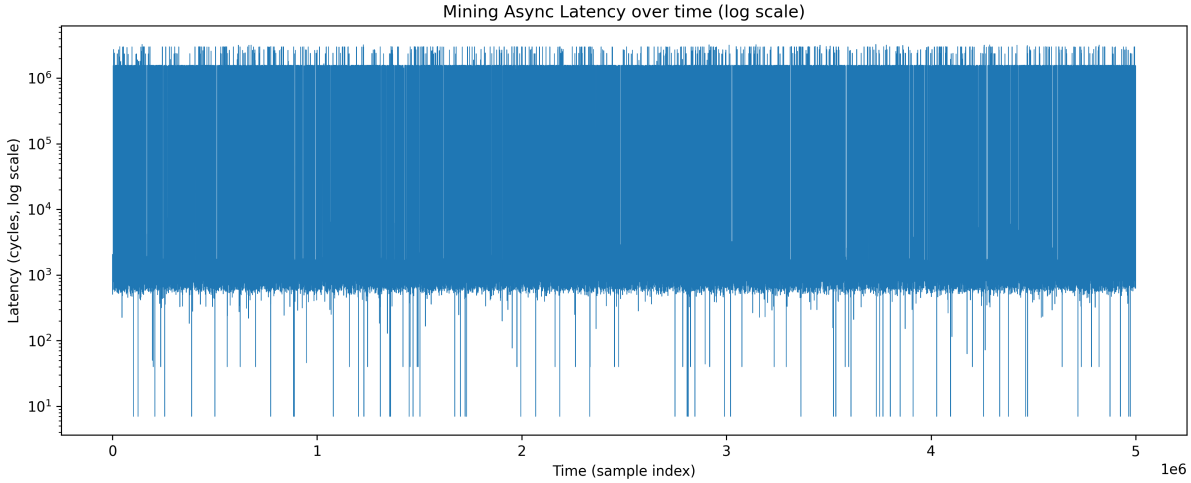


(b) Inference execution

Figure 5.10: Latencies measured on the Async pipeline for the LSTM model running in training and inference on an NVIDIA RTX 4070 with the *mnist_seq* dataset.



(a) DaggerHashimoto mining execution.



(b) NeoScrypt mining execution.

Figure 5.11: Latencies measured on the Async pipeline for the mining algorithms `DaggerHashimoto` and `NeoScrypt` on an NVIDIA RTX 4070.

To establish a direct comparison, a separate single-stream BiLSTM classifier was evaluated under the exact same conditions of Section 5.3.1, but trained exclusively on the Async latency traces. This setup measures the isolated fingerprinting capability of the asynchronous copy pipeline across the same four workload categories (CNN, Transformer, RNN, and Mining), keeping the Diffusion category excluded. The Async-based classifier reaches a lower weighted F1 score of approximately 0.70 on the validation set. This performance drop indicates that while the asynchronous copy pipeline signal remains inherently informative, it is considerably noisier and less discriminative than the TEX channel when utilized in isolation. The training curves for this configuration are displayed in Figure 5.12.

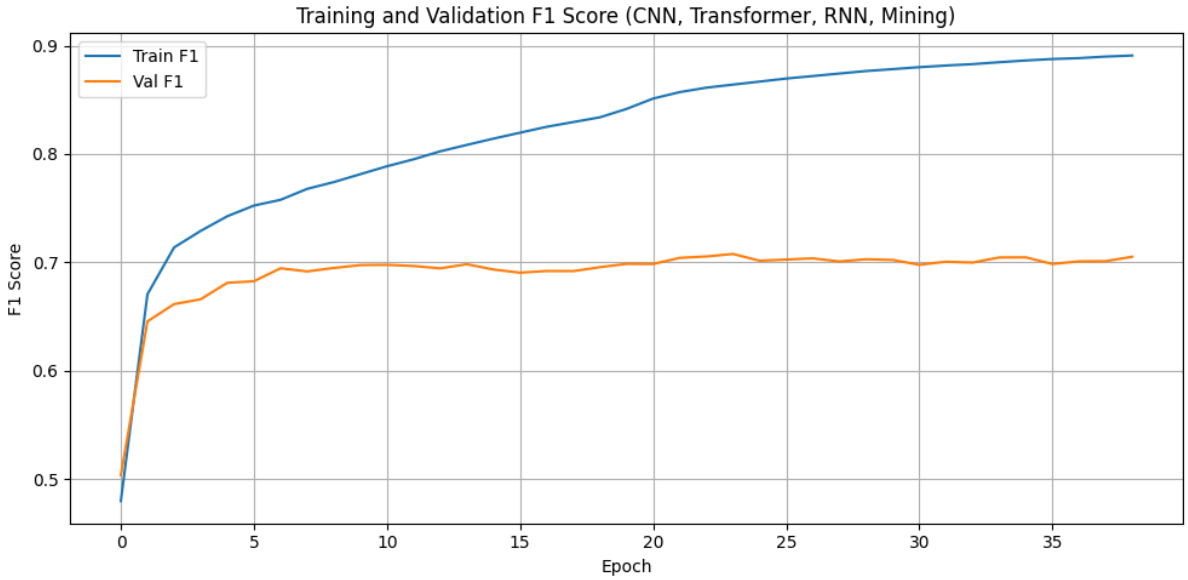


Figure 5.12: F1 score for the single-stream BiLSTM classifier trained on Async latencies.

5.4. TEXAS Assessment

Building upon both the distinct visual patterns observed in the raw latency profiles and the quantitative F_1 performance metrics achieved by the independent single-stream experiments (detailed in Figure 5.6 and Figure 5.12), we introduce **TEXAS**. This framework is designed to systematically combine the complementary side-channel information leaked by both execution pipelines, enabling the classification of workloads at multiple levels of detail and granularity based on the fusion of their temporal and statistical features of the latency traces. All experimental evaluations, architectural benchmarks, and downstream classification results presented from this point forward refer exclusively to the NVIDIA RTX 4070 platform.

Furthermore, the evaluation boundaries for the cryptocurrency mining domain must be distinctly defined due to its peculiar execution behavior under profiling. Cryptocurrency mining workloads consistently and completely saturate the GPU compute resources regardless of the specific underlying hashing algorithm. While this total hardware saturation obliterates the fine-grained execution variances necessary to distinguish further among individual sub-variants of miners, it provides a highly stable and identifiable macro-footprint. Consequently, the Mining category is retained exclusively for macro-level classification to distinguish between different overall workload families (i.e., identifying a generic trace as Mining versus CNN, Transformer, RNN, or Diffusion), while being entirely omitted from any subsequent fine-grained intra-family profiling.

The latency traces collected by the concurrent spy kernels constitute continuous time series of cycle-count measurements. In terms of data preprocessing, each raw trace is segmented into fixed-length windows of 256 measurements, with a sliding stride of 16 samples between consecutive windows to ensure adequate data density and sequence overlap. Each window is independently normalized by subtracting its mean and dividing by its standard deviation. This Z-score normalization makes the model invariant to absolute latency baselines and highly sensitive only to the structural shape and micro-variability of the intercepted signal.

To fully exploit the complementary information provided by the two concurrent side channels, the TEXAS framework implements a dedicated dual-stream BiLSTM classifier. BiLSTMs are uniquely suited for this scenario, as their dual-directional recurrent layers process the latency sequences both forward and backward in time, successfully capturing immediate latency overhead spikes alongside their subsequent cache and queue recovery phases. In this multi-stream architecture, two entirely independent parallel BiLSTM branches process the TEX trace and the Async trace simultaneously. Each branch consists of two stacked BiLSTM layers with a hidden size of 64 units per direction, effectively outputting a 128-dimensional feature vector at the last timestep. The final output vectors generated by each branch are subsequently concatenated into a comprehensive 256-dimensional joint representation. This fused vector is then passed through a dedicated fusion block consisting of a fully connected layer, a Batch Normalization layer to stabilize internal covariate shift, a Dropout layer to mitigate overfitting, and a final output layer that yields the definitive class scores.

The dual-stream network is trained using the Adam optimizer minimizing standard Cross-Entropy loss, incorporating an early stopping criterion based on the validation set to prevent over-parameterization. The compiled dataset is partitioned into training and validation sets using a strict 80/20 ratio. To robustly avoid data leakage — which occurs

when contiguous sliding windows from the same execution block span across both the training and validation splits — a separate, completely held-out set of independent CSV files originating from entirely separate execution runs is reserved exclusively for final inference evaluation.

To assess whether combining the two signals yields tangible classification benefits and improves discrimination across distinct granularities, we evaluate the dual-stream classifier on all five remaining core categories. Benefiting from the simultaneous processing of both hardware pipelines, the dual-stream TEXAS model achieves a validation F_1 score of approximately 0.94 across the extended five-category set, as illustrated in Figure 5.13. This represents a significant architectural milestone: the fusion framework successfully matches the high classification accuracy of the isolated TEX-only single-stream baseline while seamlessly scaling the output space to incorporate Diffusion as an additional, fully separated fifth class.

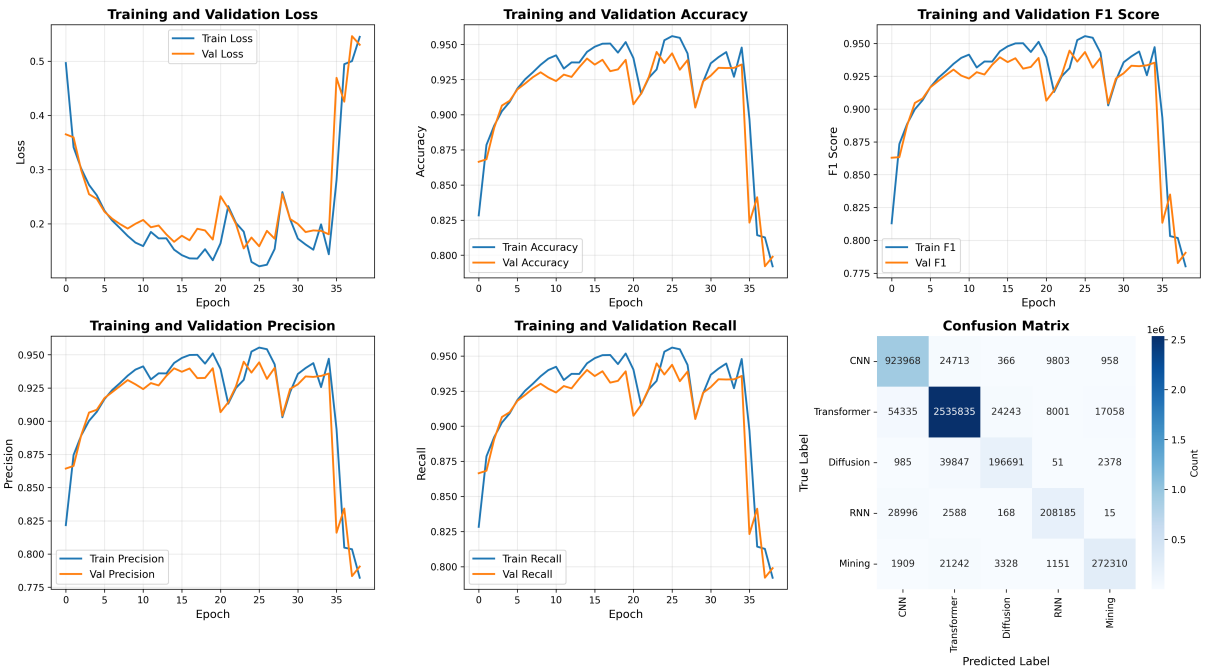


Figure 5.13: Training metrics and confusion matrix for the dual-stream BiLSTM classifier evaluated on all five workload categories.

An in-depth analysis of the resulting multi-class confusion matrix reveals that the most frequent remaining misclassifications occur specifically between the Transformer and Diffusion categories. This operational blending is highly consistent with the underlying structural relationship connecting the two domains. Modern generative diffusion models, such as Stable Diffusion v1-5, rely heavily on core transformer-based cross-attention

blocks to process conditioning tokens and map latent features. As a consequence, their underlying hardware execution footprint, memory access routines, and cache-line contention signatures closely mirror those generated by pure Transformer models, defining a clear architectural explanation for the higher similarity captured in the dual-stream side-channel traces.

Having established that the dual-stream classifier reliably identifies the macro-level workload category, we now evaluate its capability to perform fine-grained, intra-family analysis within the specific domains. For each targeted category, specialized dual-stream BiLSTM networks are instantiated. While for the CNN domain the framework aims to both identify the specific model topology and regress its structural parameters, for the remaining categories the analysis focuses exclusively on distinguishing individual models within each family. These models adopt the core joint-embedding pipeline and Z-score preprocessing methodology established in Section 5.4. Section 5.4.1 through Section 5.4.4 detail the empirical results and architectural insights obtained across these specific granular evaluations.

5.4.1. CNN Classification

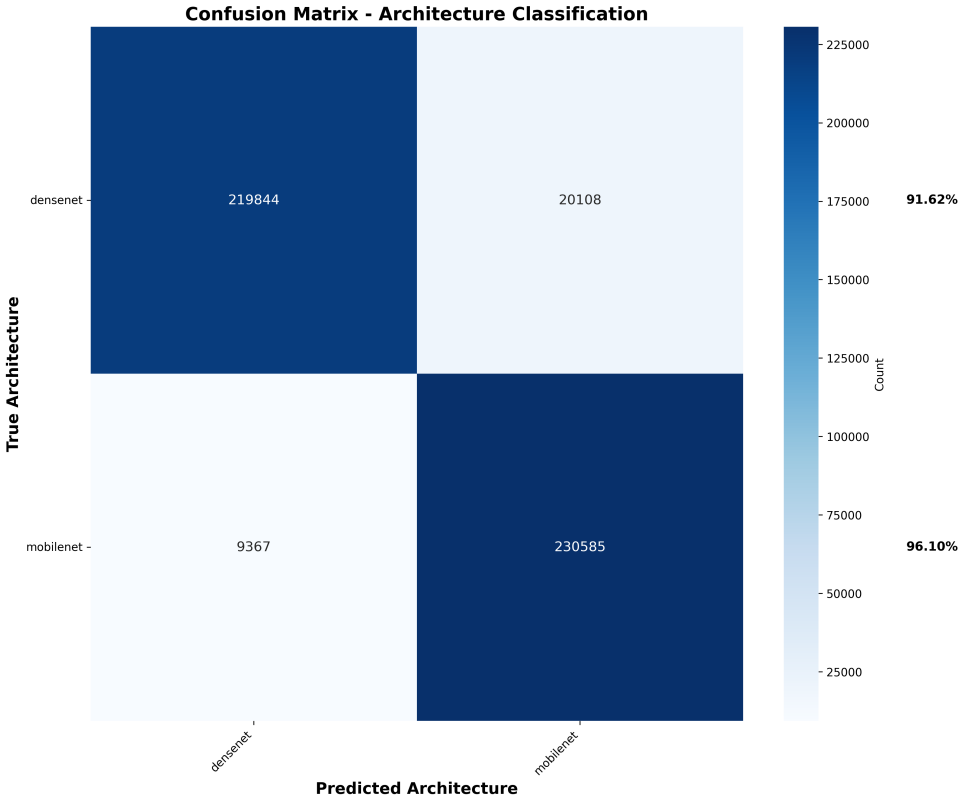
Convolutional Neural Network (CNN) workloads present highly structured and periodic execution signatures, characterized by deterministic memory access patterns heavily dominated by dense GEMM (General Matrix Multiply) operations and sliding-window convolutions. To thoroughly map the side-channel information density leaked by these pipelines, our evaluation within the CNN family is bifurcated into two distinct granular tasks: discrete model architecture identification and continuous structural attribute regression.

5.4.1.1. Model Classification

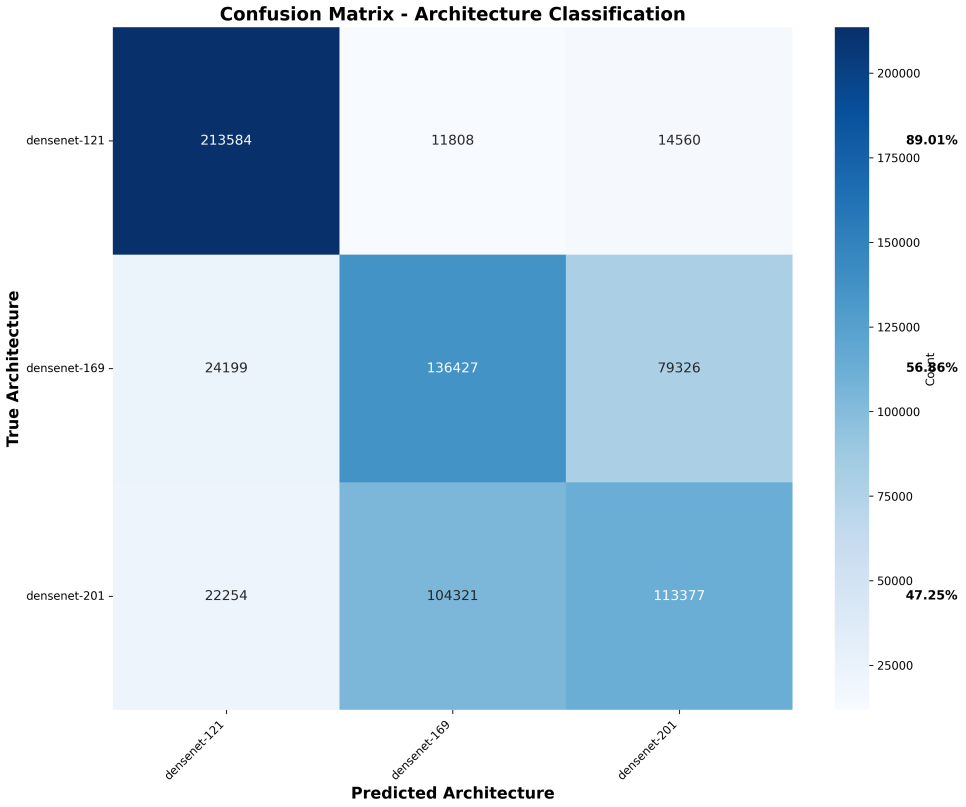
In this phase, the dual-stream classifier is tasked with identifying the exact convolutional model topology currently executing on the hardware. Even when processing identical input datasets, different convolutional architectures employ distinct layer depths, residual connection routing, and channel-width scaling factors. These structural differences directly translate into unique kernel execution boundaries, varying streaming multiprocessor (SM) occupancy rates, and distinct cache eviction cycles. By simultaneously observing the texture pipeline contention via the TEX stream and scheduling bottlenecks via the Async stream, the model effectively isolates these sub-family hardware footprints. The classification results demonstrate that the joint embedding space can reliably separate distinct network topologies with high precision, mapping the micro-architectural temporal

variances back to the specific deep learning model definition.

Within the CNN category, classification is structured across two levels. At the first level, a classifier distinguishes between DenseNet and MobileNet architectures, achieving a weighted F_1 score of **0.93**. At the second level, a dedicated classifier distinguishes between the three DenseNet variants (DenseNet-121, DenseNet-169, and DenseNet-201), achieving a weighted F_1 score of **0.63**. The confusion matrices for both levels are shown in Figure 5.14.



(a) DenseNet vs MobileNet classifier.



(b) DenseNet variant classifier.

Figure 5.14: Confusion matrices for the CNN model-level classifiers.

5.4.1.2. Architecture Regression

Going beyond discrete classification, we evaluate the capability of the TEXAS framework to infer continuous architectural hyperparameters directly from the CNN side-channel latency profiles. For this specific regression task, the categorical Softmax output layer of the fusion block is replaced with a linear activation head optimized via Mean Squared Error (MSE) loss. The network is trained to predict structural model hyperparameters, specifically the exact number of convolutional layers and pooling layers within the executing model. The empirical performance of this regression setup proves that the dual-stream latency measurements scale effectively to continuous inference, leaking granular structural dimensions of the target CNN workload without requiring access to any architectural metadata.

Specifically, we train two separate dual-stream BiLSTM regressors — one for the number of convolutional layers and one for the number of pooling layers — using the same dual-stream architecture described above. The only difference with respect to the classification setting is the output layer, which produces a single scalar value instead of class scores, and the use of Mean Squared Error (MSE) as the training loss. A ReLU activation is applied to the output to ensure non-negative predictions.

The regressors are trained on latency traces collected from a set of CNN models with known architectures, including ResNet-18, ResNet-34, ResNet-101, DenseNet-169, DenseNet-201, and MobileNet-V2, as well as PaliGemma and Mining traces used as low-layer-count references. The model is then evaluated on a held-out set of DenseNet-121 traces, which were not seen during training.

The training metrics for the convolutional layer regressor are shown in Figure 5.15. The model achieves a validation R^2 score of approximately 0.76 and a Mean Absolute Error of approximately 20 layers, stabilizing after roughly 30 epochs. The predicted vs actual plot confirms that the regressor learns a meaningful correlation between the latency signal and the layer count, though with considerable spread around the true value.

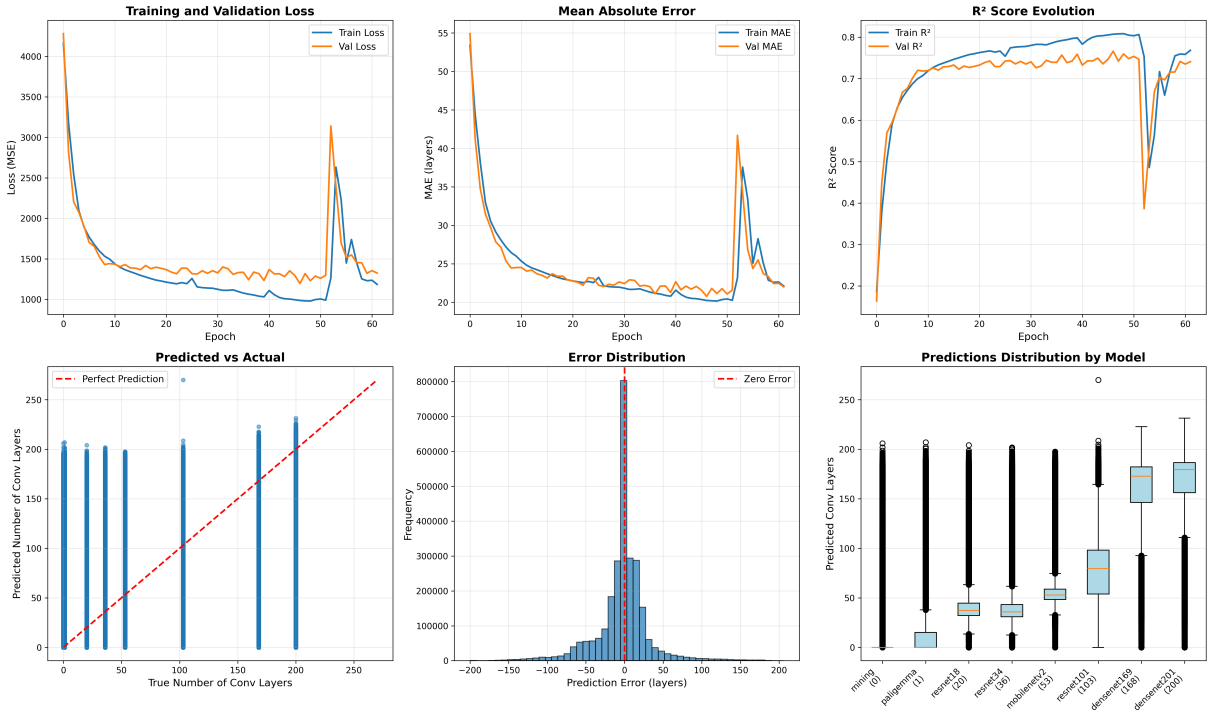


Figure 5.15: Training metrics for the convolutional layer count regressor, including loss, MAE, R^2 score, predicted vs actual values, error distribution, and per-model prediction spread.

Figure 5.16 shows the distribution of predictions on DenseNet-121 traces across different datasets and execution phases — both while the model is running in training mode and in inference mode (labeled as *test* in the figure). The true value is 120 convolutional layers. The mean prediction ranges between approximately 80 and 107 layers depending on the dataset and phase, with absolute errors between 11% and 33%. The best result is obtained on the CIFAR-10 training traces, where the mean prediction of 106.6 layers corresponds to an error of only 11.2%, while inference traces generally yield higher errors, suggesting that the training phase produces a stronger and more consistent contention signal on the measured pipelines.

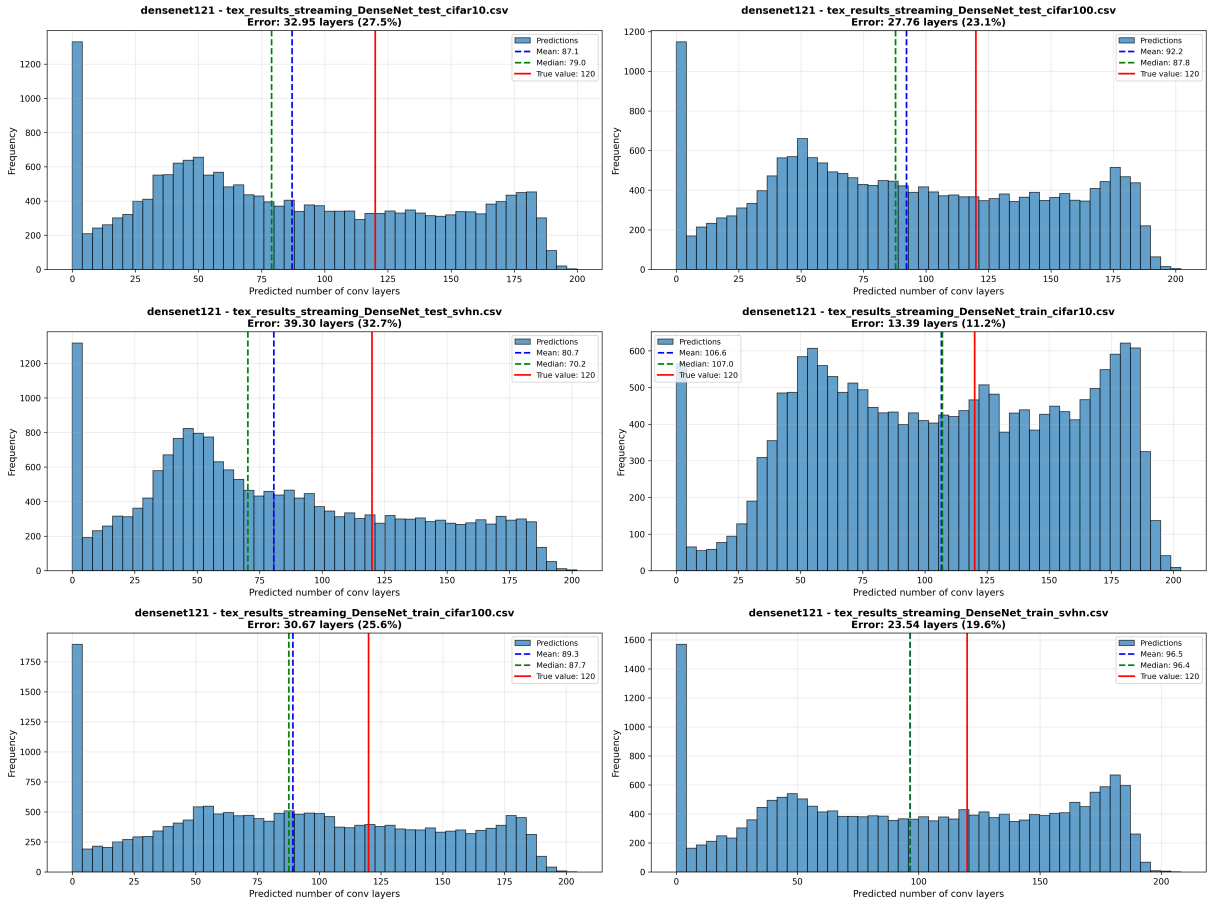


Figure 5.16: Distribution of predicted convolutional layer counts for DenseNet-121 across different datasets and execution phases.

Figure 5.17 summarizes the aggregated predictions against the true value, showing a mean absolute error of 27.9 layers across all inference traces.

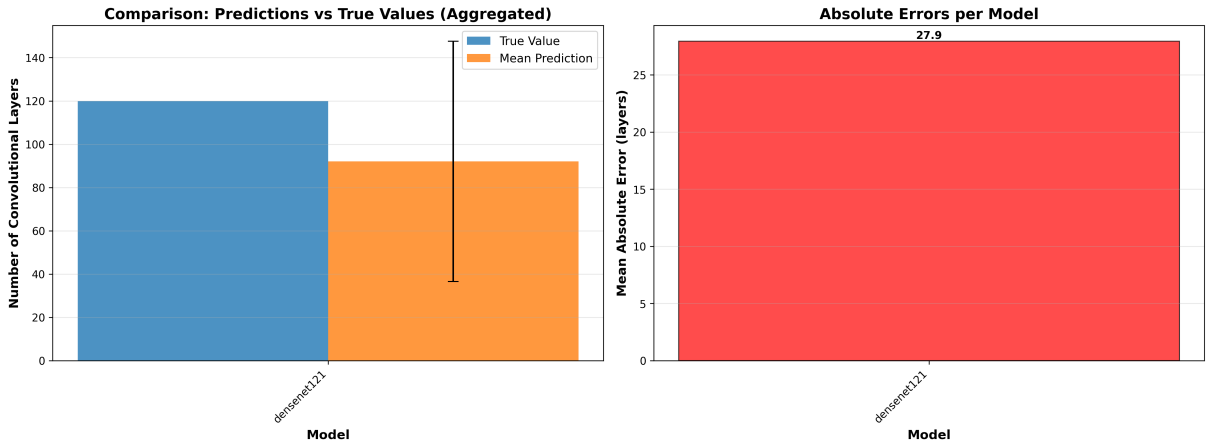


Figure 5.17: Aggregated comparison of predicted vs true convolutional layer count for DenseNet-121, with mean absolute error of 27.9 layers.

These results suggest that the latency traces captured by the dual-stream classifier carry information about the architectural complexity of CNN models. While the regression does not achieve the precision levels of the classification experiments, it demonstrates that side-channel signals can be leveraged to approximate structural properties of the victim model without any direct access to its code or parameters. A similar experiment was conducted for pooling layer count, yielding comparable results with a mean absolute error of 1.6 pooling layers on DenseNet-121, whose true value is 5; the corresponding figures are provided in Appendix A. We regard this as a proof-of-concept contribution, and leave more precise architectural inference as a direction for future work.

5.4.2. Transformer Classification

The Transformer category is the most diverse in our benchmark, comprising models from four distinct families: BERT, Deepseek, Gemma, and GPT-2. Given this diversity, classification is structured hierarchically across two levels.

At the first level, a classifier is trained to distinguish between the four families. It achieves a weighted F_1 score of **0.89** on the validation set. Figure 5.18 shows the corresponding confusion matrix.

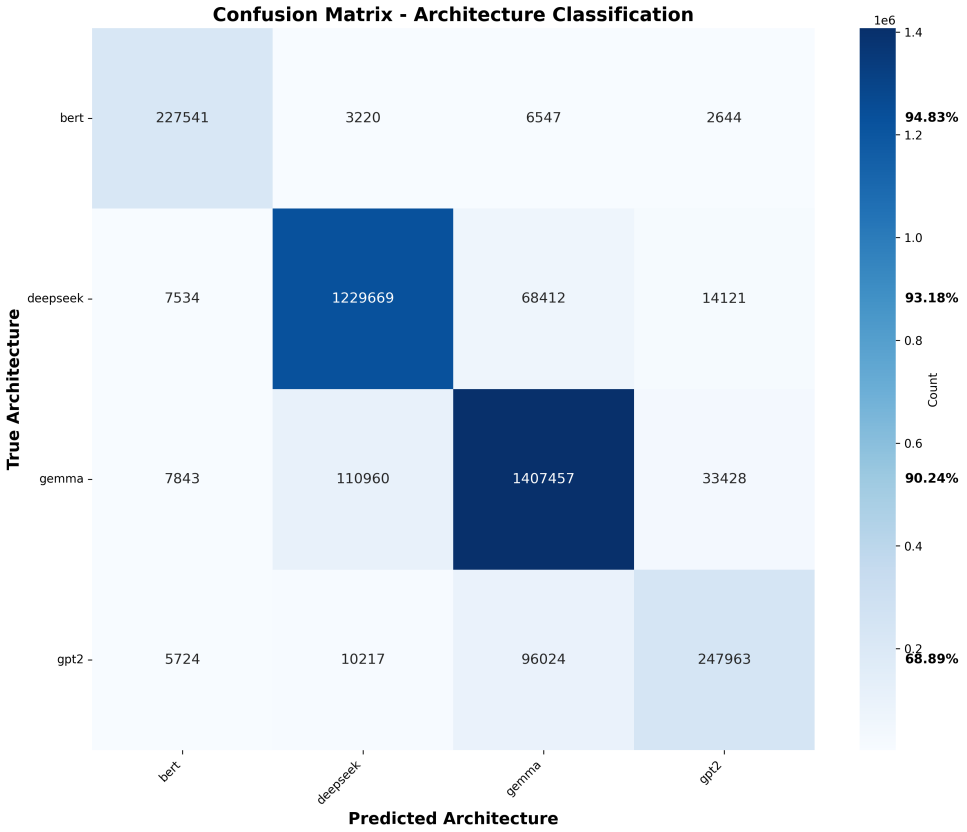
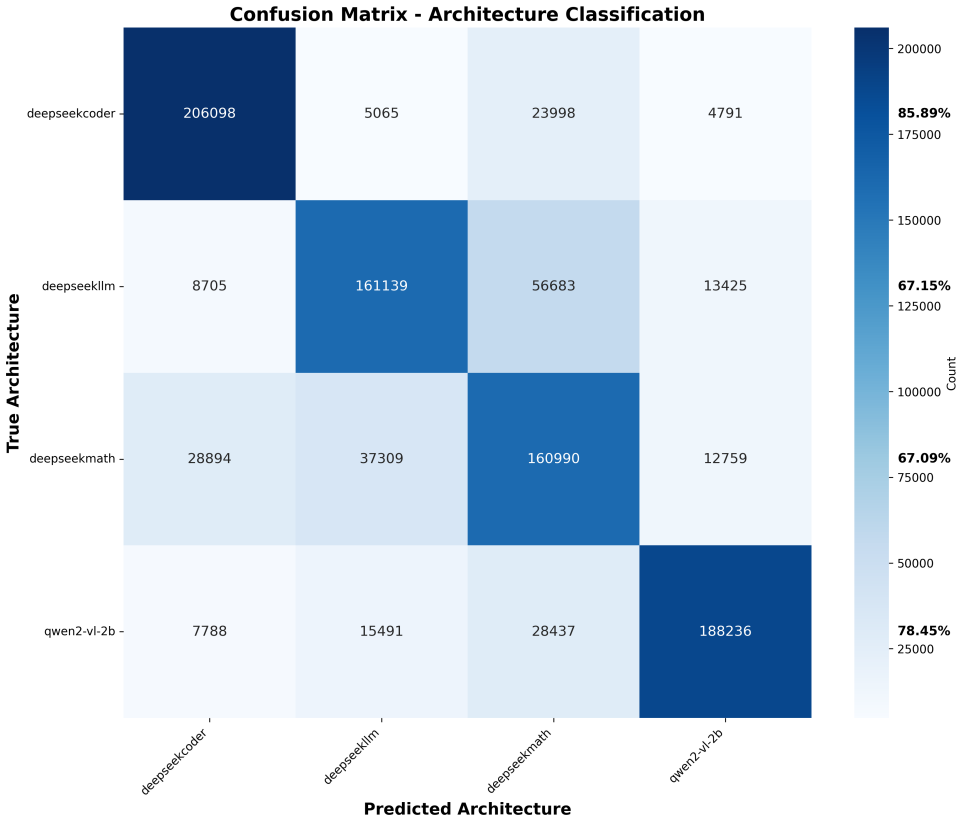
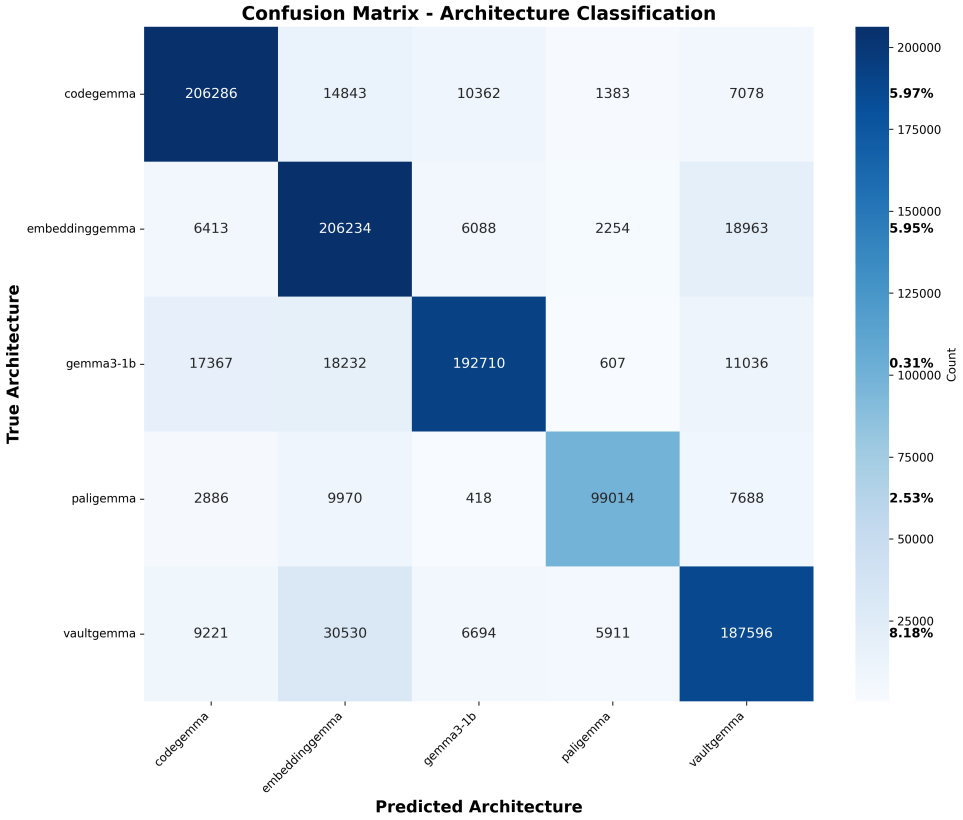


Figure 5.18: Confusion matrix for the Transformer family-level classifier.

At the second level, separate classifiers are trained for the two families that contain multiple models: Deepseek and Gemma. The Deepseek classifier distinguishes between DeepSeek-Coder, DeepSeek-Math-7b, DeepSeek-LLM-7b, and Qwen2-VL-2B, achieving a weighted F_1 score of **0.74**. The Gemma classifier distinguishes between five models within the family: CodeGemma, EmbeddingGemma, Gemma-3-1b, PaliGemma, and VaultGemma, achieving a weighted F_1 score of **0.82**. The confusion matrices for both classifiers are shown in Figure 5.19.



(a) Deepseek and Qwen model-level classifier.



(b) Gemma model-level classifier.

Figure 5.19: Confusion matrices for the Transformer model-level classifiers.

5.4.3. RNN Classification

Within the RNN category, a single classifier is trained to distinguish between GRU and LSTM architectures, achieving a weighted F_1 score of **0.96**. The confusion matrix is shown in Figure 5.20.

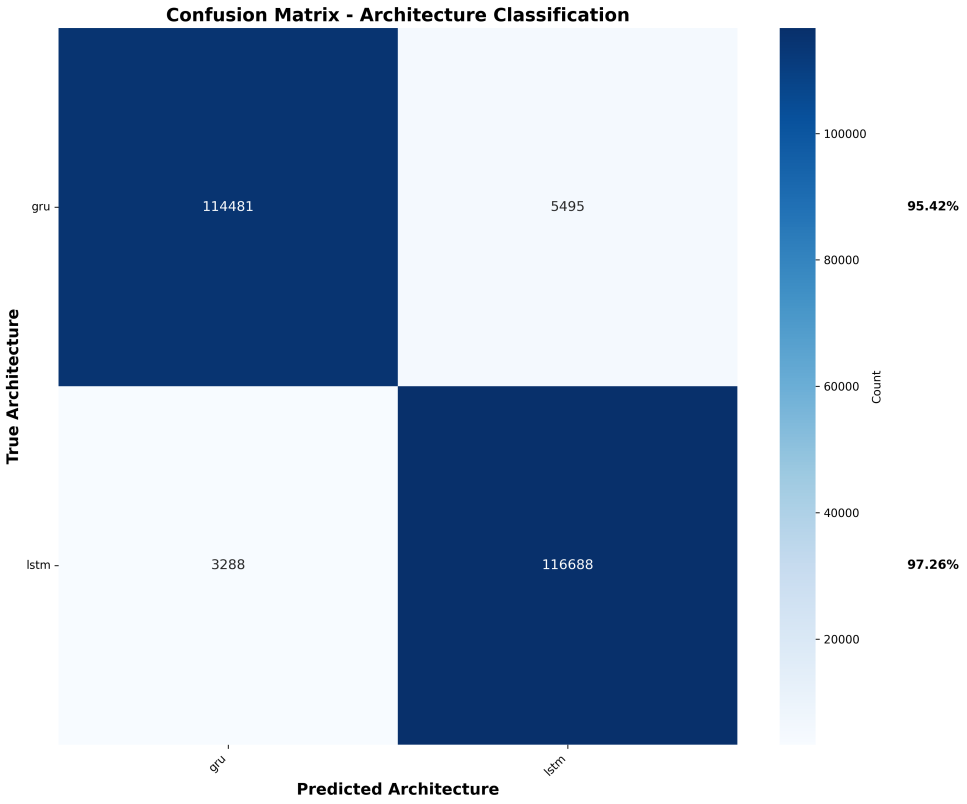


Figure 5.20: Confusion matrix for the RNN model-level classifier.

5.4.4. Diffusion Classification

The generative diffusion domain introduces highly complex, multi-stage hardware execution behaviors that distinguish it from standard discriminative deep learning models. Rather than relying on a single static feed-forward pass, diffusion pipelines execute long, iterative denoising loops that dynamically alternate between textual token conditioning, cross-attention mappings, and latent noise predictions. This subsection explores intra-family classification within this advanced generative domain.

The dual-stream classifier is evaluated on its ability to perform fine-grained model classification within this domain. Specifically, a classifier is trained to distinguish between Stable Diffusion v1.5 and tiny-sd, achieving a weighted F_1 score of **0.63**. The confusion matrix is shown in Figure 5.21.

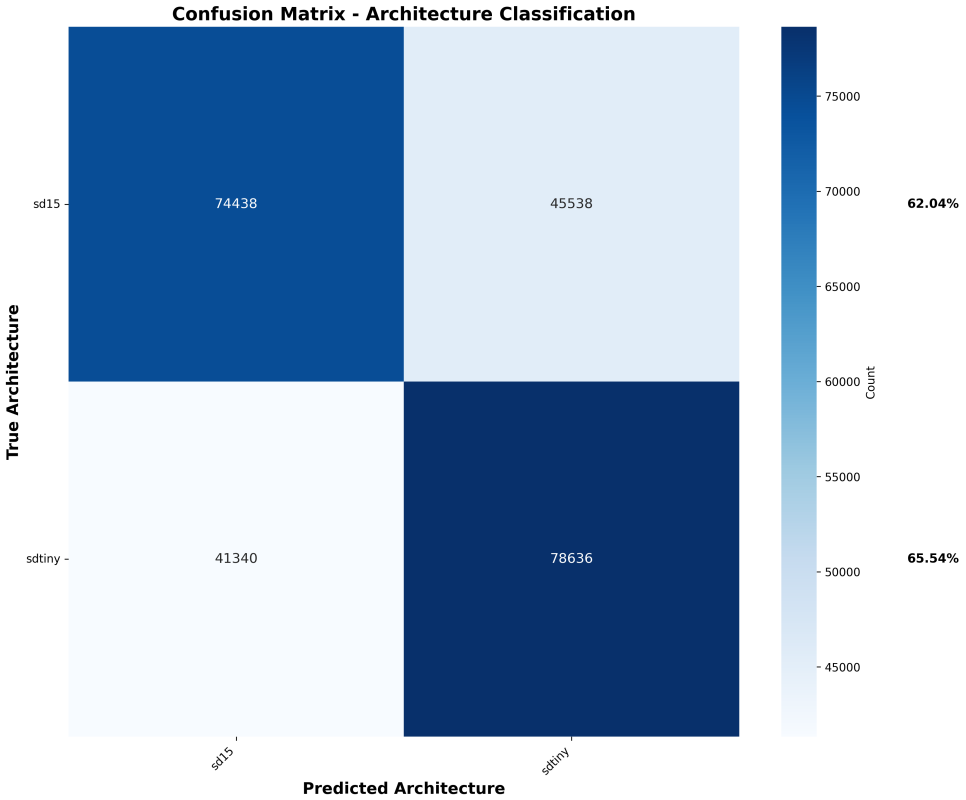


Figure 5.21: Confusion matrix for the Diffusion model-level classifier.

5.5. Performance Comparison Across Different GPUs

To assess the generalizability of the proposed attack across different GPU architectures, a representative subset of the workloads evaluated on the RTX 4070 was also tested on the NVIDIA RTX 3050 and RTX 5070. The subset was selected to include at least one model per category, ensuring coverage of all workload types. Additionally, the choice of models was constrained by the hardware limitations of the RTX 3050, whose reduced VRAM capacity prevented the execution of larger models that ran successfully on the RTX 4070 and RTX 5070. Mining workloads were excluded from the validation set, as no model-level classification was attempted for this category. Table 5.4 lists the models and datasets used in this validation phase.

It is worth recalling that, as discussed in Section 4.1, the classifiers used in this validation phase were retrained from scratch on data collected from each respective GPU. Since the absolute latency values measured by the spy kernels depend on the specific hardware, a classifier trained on RTX 4070 traces cannot be directly applied to traces collected from a different GPU model. Each GPU therefore requires its own set of trained classifiers.

5.5.1. Category-Level Classification

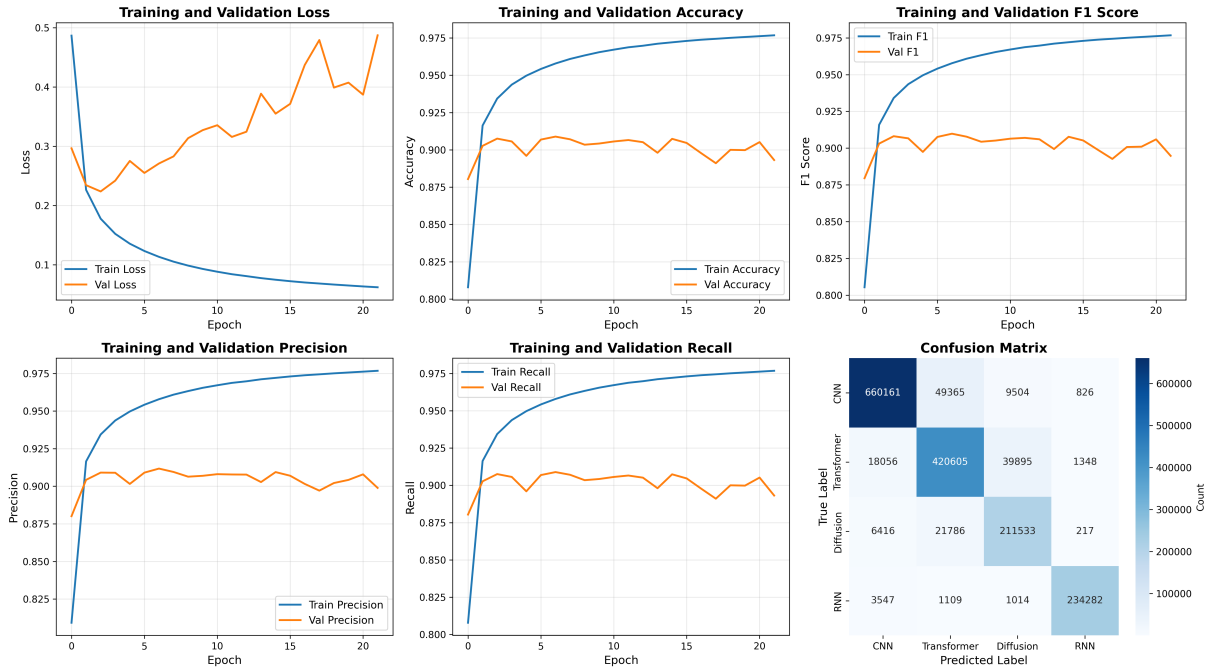
The dual-stream classifier was evaluated on four workload categories — CNN, Transformer, Diffusion, and RNN — on both the RTX 3050 and the RTX 5070. The overall results are summarized in Table 5.2.

Table 5.2: Category-level classification results across the three GPUs.

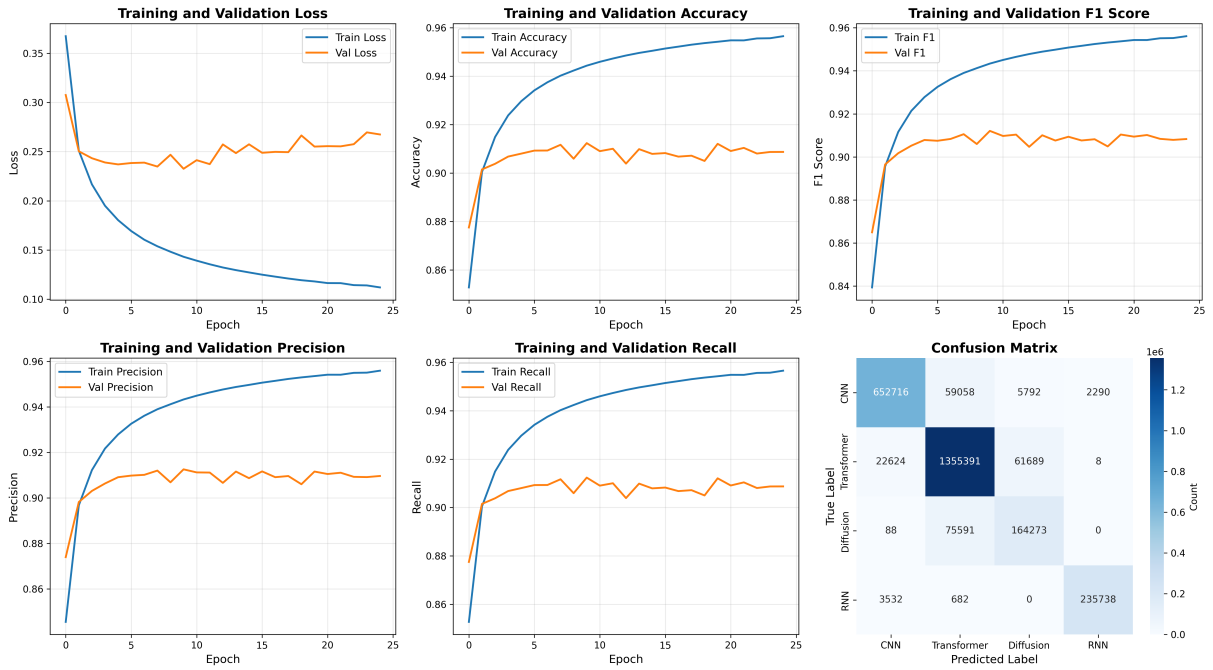
GPU	Accuracy	Weighted F1	Categories
RTX 4070	0.94	0.94	5 (incl. Mining)
RTX 3050	0.91	0.91	4
RTX 5070	0.91	0.91	4

The results are consistent across all three GPUs, confirming that the dual-stream classifier generalizes well to different hardware configurations. The slight difference with respect to the RTX 4070 is partly attributable to the reduced number of categories — Mining, which is among the easiest to classify, is not included in the validation experiments.

Looking at the per-class results, CNN and RNN achieve high F1 scores on both validation GPUs (CNN: 0.94 on RTX 3050, 0.93 on RTX 5070; RNN: 0.98 on both), consistent with the strong performance observed on the RTX 4070. The Transformer class achieves F1 scores of 0.86 and 0.93 on the RTX 3050 and RTX 5070 respectively. The Diffusion category shows the most variability across GPUs, with an F1 of 0.84 on the RTX 3050 and 0.70 on the RTX 5070, a result that is consistent with the systematic confusion between Diffusion and Transformer already observed on the RTX 4070, and which we attribute to the architectural overlap between the two categories. The training metrics and confusion matrices for both GPUs are shown in Figure 5.22.



(a) RTX 3050 dual-stream classifier.



(b) RTX 5070 dual-stream classifier.

Figure 5.22: Training metrics and confusion matrices for the dual-stream BiLSTM classifier evaluated on the validation GPUs.

5.5.2. Model-Level Classification

Model-level classifiers were also trained and evaluated on the RTX 3050 and RTX 5070, following the same hierarchical structure described in Section 5.4. Given the reduced workload subset used in the validation phase, only a subset of the model-level classifiers trained on the RTX 4070 could be replicated. Table 5.3 summarizes the weighted F1 scores obtained on each GPU for each classifier, alongside the corresponding results on the RTX 4070 for reference.

Table 5.3: Model-level classification results across the three GPUs.

Category	Classifier	RTX 4070	RTX 3050	RTX 5070
Transformer	Family-level	0.89	0.98	0.81
CNN	DenseNet vs MobileNet	0.93	0.82	0.93
RNN	GRU vs LSTM	0.96	0.93	0.89
Diffusion	SD v1.5 vs tiny-sd	0.63	0.76	0.81

The model subsets available on the validation GPUs differ from those used on the RTX 4070, and the classifiers were adapted accordingly. Overall, the model-level results on the RTX 3050 and RTX 5070 are broadly consistent with those obtained on the RTX 4070, confirming that the attack generalizes across hardware configurations when the classifier is retrained on data collected from the target GPU.

Table 5.4: Complete list of models and datasets used in the validation on the RTX 3050 and RTX 5070.

Model	Datasets	Phase
CNN		
DenseNet121	CIFAR-10 CIFAR-100 SVHN	Train Inference

Model	Datasets	Phase
DenseNet201	CIFAR-10 CIFAR-100 SVHN	Train Inference
MobileNet_V2	CIFAR-10 CIFAR-100 SVHN	Train Inference
Transformer		
BERT-base-uncased	IMDB Yelp Polarity AG News	Train Inference
GPT-2	Wikitext-2 IMDB Yelp Polarity AG News	Train Inference
Diffusion		
Stable-Diffusion-v1-5	sdxl-0.9 naruto-blip-captions	Train Inference
Tiny-sd	sdxl-0.9 naruto-blip-captions	Train Inference
RNN		
LSTM	mnist_seq tiny_shakespeare	Train Inference
GRU	mnist_seq tiny_shakespeare	Train Inference

5.6. Discussion

The experimental results presented in this chapter demonstrate that contention-based side-channel attacks targeting the TEX and asynchronous copy pipelines of NVIDIA GPUs can reliably fingerprint co-resident workloads across a range of model categories, families, and hardware configurations. We now discuss the key patterns emerging from the results, the cases where the attack is less effective, and the practical implications for real-world deployment scenarios.

5.6.1. Overall Classification Performance

At the category level, the dual-stream classifier consistently achieves weighted F1 scores above 0.90 across all three GPUs tested, confirming that the combination of TEX and Async signals provides a robust fingerprinting primitive. The comparison between the single-stream and dual-stream classifiers at the category level shows that the TEX pipeline alone is already highly discriminative (F1 $\approx$ 0.93), while the Async pipeline in isolation performs considerably worse (F1 $\approx$ 0.70). This asymmetry suggests that the texture fetch pipeline is more sensitive to workload differences at the category level, likely because different workload categories exhibit substantially different spatial memory access patterns that interact differently with the texture cache. The Async pipeline, while less discriminative on its own, contributes complementary information that becomes relevant when finer-grained distinctions are needed.

At the model level, performance varies considerably across categories. The RNN classifier achieves the highest F1 score (0.96 on the RTX 4070), reflecting the fact that GRU and LSTM architectures produce markedly different contention patterns despite belonging to the same category. At the other extreme, the DenseNet variant classifier and the Diffusion classifier both achieve F1 scores of 0.63, indicating that distinguishing between structurally similar models within the same family is substantially harder. The Transformer classifiers occupy an intermediate range, with the family-level classifier achieving 0.89 and the model-specific classifiers between 0.74 and 0.82.

5.6.2. Analysis of Difficult Cases

The most persistent source of misclassification across all GPUs and classification levels is the confusion between the Transformer and Diffusion categories. As noted in Section 5.4, this is consistent with the architectural overlap between the two: modern diffusion models rely heavily on transformer based attention mechanisms, and their memory access patterns

at the pipeline level are therefore difficult to distinguish from those of pure transformer workloads. This confusion is observable even on the validation GPUs, where the Diffusion class consistently shows lower per-class F1 scores than the other categories.

The low F1 score of the DenseNet variant classifier (0.63) reflects a different challenge: the three DenseNet variants (DenseNet-121, DenseNet-169, DenseNet-201) differ primarily in depth, and the contention signatures they produce on the measured pipelines are quantitatively similar, making them hard to separate based on timing alone. This observation is consistent with the results of the CNN architecture regression, where the regressor learns a meaningful trend between layer count and latency but with considerable spread, confirming that depth differences produce gradual rather than sharp changes in contention behavior.

The inability to distinguish between mining algorithms represents a fundamental limitation of the contention-based approach, rooted in a lack of discriminative information in the collected traces. All mining algorithms saturate GPU memory bandwidth with highly repetitive access patterns regardless of the specific hashing algorithm, leaving little variation in the timing traces of either pipeline that could serve as a distinguishing feature. This defines one of the boundary conditions of contention-based fingerprinting: workloads that are too homogeneous resist classification, and useful discrimination is only possible where workloads produce consistent but distinguishable contention signatures.

5.6.3. Practical Security Implications

The results of this work have concrete implications for the security of shared GPU environments. An unprivileged attacker who can launch CUDA kernels on the same physical GPU as a victim can, without any elevated privileges or modifications to the victim's execution environment, determine with high confidence which category of workload the victim is running, and in many cases identify the specific model family or model being executed. In practical terms, this means that an attacker sharing a research server or a multi-user workstation with a victim could infer whether the victim is running a large language model, a CNN, or a diffusion model, and potentially narrow down the specific architecture. This information could be used to inform more targeted attacks, to infer the value of a proprietary model being deployed, or to reconstruct aspects of a victim's research or commercial activity.

The CNN architecture regression further extends this threat surface: even without identifying the exact model, an attacker can approximate the architectural complexity of an unknown CNN by estimating its layer count from timing traces alone. While the current

precision is limited, this demonstrates that side-channel leakage extends beyond model identity to structural properties that are typically considered proprietary.

5.6.4. Limitations

Several limitations constrain the applicability of the proposed attacks in practice. First, the classifiers are hardware-specific: since the absolute latency values measured by the spy kernels depend on the microarchitectural characteristics of the target GPU, a classifier trained on one GPU model cannot be directly applied to another. The attacker must therefore have prior access to the same GPU model as the victim to collect training data, which is a realistic but non-trivial assumption.

Second, the attack assumes that the victim is running a workload that belongs to one of the categories represented in the training set. An entirely novel workload type that was not seen during training would likely be misclassified rather than flagged as unknown, which limits the attack's ability to generalize to arbitrary victim applications.

Third, the data collection phase requires the victim to execute the same workload for a sufficient duration to collect a meaningful number of latency measurements. Short-lived or intermittent workloads may not produce enough contention signal for reliable classification, particularly at the model level where finer-grained distinctions require more data.

Fourth, the attack is passive and does not actively interfere with the victim's execution, but it does consume GPU resources during the spy kernel's operation. On systems with strict resource accounting or monitoring, the presence of an anomalous long-running CUDA kernel could potentially be detected by a sufficiently attentive system administrator.

Fifth, it is worth noting that the classifiers and regressors presented in this work were not subject to systematic hyperparameter optimization. The results presented should therefore be interpreted as a lower bound on the attack's potential effectiveness rather than as an optimized upper bound.

Finally, although outside the primary scope of this work, we conducted preliminary experiments to assess whether the framework could be extended to interactive 3D graphics applications — workloads that drive real-time rendering pipelines on the same GPU hardware targeted by TEXAS. These experiments revealed that 3D applications produce latency profiles that are fundamentally incompatible with the current classification approach: the contention signatures vary drastically from frame to frame and from title to title, driven

by continuously changing rendering demands, and no stable feature representation could be extracted. The framework is therefore unable to fingerprint this class of workloads in its current form. This limitation is however of limited practical relevance in the threat scenarios considered by this work: shared research servers and multi-user workstations are overwhelmingly used for compute workloads such as deep learning training and inference, and interactive 3D applications are rarely if ever executed in such environments.

Taken together, these considerations highlight both the practical threat posed by the proposed attacks and the directions along which future work could improve upon the results presented here, as discussed in Chapter 7.

6 | Countermeasures

TEXAS exploits shared microarchitectural resources that are, by design, accessible to any unprivileged CUDA process running on the same GPU. Defending against contention-based side-channel attacks is therefore fundamentally a problem of resource isolation and signal obfuscation. In this chapter, we survey existing countermeasures that are relevant to the attack surface described in this work, and we discuss potential mitigations that could be applied specifically to the TEX and Async pipelines targeted by the proposed attack. Unless otherwise noted, the effectiveness of the countermeasures discussed in this chapter has not been empirically verified against TEXAS, and their discussion should be interpreted as an analysis of their potential applicability rather than a confirmed evaluation.

6.1. Existing Countermeasures

6.1.1. Hardware Isolation: MPS and MIG

NVIDIA provides two mechanisms for sharing a single GPU among multiple users or processes: the Multi-Process Service (MPS) and Multi-Instance GPU (MIG). MPS allows multiple CUDA contexts to share the same GPU concurrently by funneling them through a single CUDA context on the device, reducing context-switch overhead and improving utilization. Since MPS does not partition the underlying hardware pipelines, co-resident processes would likely still compete for the same shared resources — including the TEX and asynchronous copy pipelines — and contention may remain observable in principle. However, MPS may alter the scheduling behavior of concurrent kernels in ways that could affect the contention signal, and whether the attack proposed in this thesis would remain effective under MPS has not been empirically verified.

MIG partitions a single physical GPU into multiple isolated instances, each with its own dedicated compute resources, memory, and cache. This provides a much stronger isolation guarantee than MPS, and it is plausible that contention on the pipelines targeted by this thesis would be significantly reduced or eliminated between processes running in different

MIG instances. However, as demonstrated by Zhang et al. [14], MIG does not partition the last-level TLB, leaving a residual attack surface even under this isolation scheme. Furthermore, MIG is only available on high-end data center GPUs such as the NVIDIA A100 and H100, and is not supported on the consumer-grade GPUs used in this thesis.

6.1.2. Cache Partitioning

In the CPU domain, Intel's Cache Allocation Technology (CAT) allows the operating system to partition the last-level cache among different processes or virtual machines, reducing the ability of co-resident workloads to interfere with each other through cache contention. Liu et al. [4] proposed CATalyst, a mechanism that leverages CAT to partition the LLC into a hybrid hardware-software managed cache, demonstrating that LLC side-channel attacks can be effectively defeated with negligible performance overhead. An analogous mechanism does not currently exist for GPU caches: the L2 cache and the texture cache are shared among all concurrently executing kernels without any partitioning capability exposed to the software stack. Introducing hardware support for GPU cache partitioning — similar to Intel CAT but adapted to the GPU's many-core architecture — could in principle reduce contention-based leakage on the texture pipeline, though it would not necessarily address contention on the asynchronous copy pipeline, which operates on global memory rather than the on-chip cache hierarchy.

6.1.3. Memory Coloring and Partitioning

Memory coloring is a software technique, originally developed for CPU NUMA systems, that assigns physical memory pages to different tenants in a way that minimizes sharing of memory controller resources and reduces contention on the memory bus. Applied to GPU environments, memory coloring could potentially reduce contention on the asynchronous copy pipeline by ensuring that different processes access distinct regions of the physical memory hierarchy. However, it is unclear whether memory coloring alone would be sufficient to eliminate the contention signal observed by the Async spy kernel, and it would not address contention on the TEX pipeline, which depends on the texture cache rather than global memory bandwidth.

6.1.4. Timer Restriction

Both spy kernels of TEXAS rely on the `clock64()` instruction to measure per-operation latency with cycle-level precision. In the CPU domain, several countermeasures against timing-based side-channel attacks have proposed degrading or restricting the resolution

of high-precision timers to make it harder for an attacker to measure small timing differences. Liu et al. [5] proposed on-demand time blurring, a hypervisor-level mechanism that masks the low-order bits of the timestamp counter with negligible performance overhead, demonstrating that this approach can effectively mitigate timing-based covert channels on CPUs. A similar approach applied to `clock64()` could potentially increase measurement noise and reduce the attack's classification accuracy. However, Zhang et al. [15] have shown that GPU cache attacks can be carried out even without a high-resolution timer through the Invalidate+Compare primitive, suggesting that timer restriction alone may not be sufficient to eliminate all timing-based leakage.

6.2. Proposed Countermeasures

6.2.1. Enforcing Non-Bare-Metal Access

The attack presented in this thesis was designed and evaluated in bare-metal configurations, where both the victim and the attacker have direct, unprivileged access to the same physical GPU without any intermediary layer. Introducing a virtualization or containerization layer between user processes and the GPU hardware could potentially reduce the effectiveness of the proposed attack, as such layers may restrict access to low-level hardware counters such as `clock64()` or alter the timing characteristics of shared pipelines.

6.2.2. Noise Injection into Pipeline Latencies

A targeted mitigation against the specific attack proposed in this thesis would be to introduce artificial noise into the latency of the TEX and Async pipelines as observed by unprivileged processes. Naghibijouybari et al. [7] demonstrated that rate limiting and granularity limiting of GPU memory usage reporting can significantly degrade the precision of GPU side-channel attacks in their specific attack context. A similar principle could potentially be applied to the pipeline latencies measured by `clock64()`: randomly delaying the completion of texture fetch operations or asynchronous copy synchronization points at the driver level could make it harder for a spy kernel to reliably measure the contention signal. The challenge with this approach is calibrating the amount of noise: too little noise would not meaningfully degrade classification accuracy, while too much noise would impact the legitimate performance of workloads that rely on these pipelines.

6.2.3. Access Control on Hardware Performance Counters

The `clock64()` instruction is available to any unprivileged CUDA process and returns a high-resolution cycle counter local to each Streaming Multiprocessor. Restricting access to this instruction to privileged processes only would prevent unprivileged spy kernels from obtaining the precise timing measurements required by the proposed attack. In the CPU domain, Intel processors expose a Time Stamp Disable (TSD) bit in the CR4 control register that restricts execution of the `rdtsc` instruction to privilege level 0, providing a hardware-enforced mechanism for controlling timer access. An analogous mechanism for CUDA's `clock64()` does not currently exist, but implementing such a restriction at the driver or hardware level could represent a targeted and low-overhead mitigation. As noted above, the effectiveness of this countermeasure would depend on whether alternative timing mechanisms remain accessible to unprivileged code.

6.2.4. Pipeline Scheduling Randomization

A more fundamental mitigation would be to introduce randomization into the scheduling of operations on shared hardware pipelines, so that the latency observed by a spy kernel is less correlated with the activity of co-resident workloads. Random delays in the TEX pipeline or the asynchronous copy pipeline completion could increase the variance of the spy's measurements and reduce the signal-to-noise ratio of the collected traces. Unlike noise injection at the driver level, hardware-level scheduling randomization could be harder for an attacker to compensate for by simply collecting more measurements, as the variability would be intrinsic to the pipeline operation. The main concern with this approach is the potential performance impact on legitimate workloads that depend on predictable pipeline latency, and empirical evaluation would be needed to assess the trade-off between security and overhead.

7 | Conclusions and Future Developments

This thesis has presented TEXAS, a contention-based side-channel attack on NVIDIA GPUs that exploits two distinct hardware pipelines through dedicated spy kernels: the Texture Units pipeline (TEX spy kernel) and the asynchronous global-to-shared memory copy pipeline (Async spy kernel), introduced in NVIDIA's Ampere architecture. Both spy kernels operate entirely from unprivileged user-level code, requiring no elevated permissions, no access to the victim's memory or execution environment, and no modifications to the system configuration.

The central contribution of this work is the demonstration that timing signals from two independent hardware pipelines can be fused within TEXAS into a unified dual-stream classification framework that outperforms either spy kernel used in isolation. The dual-stream BiLSTM classifier achieves weighted F1 scores above 0.90 at the category level across three different NVIDIA GPU models — the RTX 3050, RTX 4070, and RTX 5070 — confirming that TEXAS generalizes across hardware configurations when the classifier is retrained on data collected from the target GPU. At finer granularity, the hierarchical classification scheme successfully distinguishes between model families and, in many cases, individual models within each category, with F1 scores ranging from 0.63 for structurally similar models to 0.96 for architecturally distinct ones. Beyond classification, the work demonstrates as a proof of concept that timing traces can also be used to approximate architectural properties of CNN models — specifically the number of convolutional and pooling layers — through regression on the collected latency traces.

These results establish that GPU side-channel leakage is not limited to a single hardware component, and that combining signals from multiple independent pipelines within TEXAS provides a more complete and robust fingerprinting primitive than any single-source approach. TEXAS is passive, requires no interaction with the victim, and exploits hardware resources that are shared by design and cannot easily be partitioned without significant performance impact.

7.1. Future Developments

Several directions remain open for future investigation.

Hyperparameter optimization and architecture search. The classifiers and regressors presented in this work were not subject to systematic hyperparameter tuning. The architectures, training procedures, and data preprocessing choices were selected based on reasonable defaults and preliminary experimentation, but no exhaustive search over hyperparameter configurations was conducted. Exploring alternative sequence model architectures — such as transformer-based classifiers or models with attention mechanisms — as well as optimizing window size, stride, hidden dimensions, and training procedures could yield meaningfully higher classification accuracy and regression precision than those reported here. The results presented in this thesis should therefore be interpreted as a lower bound on the attack's potential effectiveness rather than as an optimized upper bound.

Architectural inference for unknown CNN models. The CNN architecture regression presented in this thesis is limited to estimating the layer count of models represented in the training set. A natural and impactful extension would be to develop a framework capable of inferring the architecture of entirely unknown or proprietary CNN models — including custom architectures not seen during training — by learning a more general mapping between latency signatures and structural properties. This would significantly increase the practical threat posed by the attack, as it would no longer require the attacker to enumerate the victim's possible architectures in advance, enabling inference on black-box models deployed in shared environments.

Extension to other hardware pipelines. This work focuses on two specific shared resources: the TEX pipeline and the asynchronous copy pipeline. Other GPU hardware components — such as Tensor Cores, the L2 cache, or the memory controller — may provide additional or complementary side-channel signals. Incorporating measurements from a broader set of pipelines into the classification framework could improve accuracy for the cases where the current two-pipeline approach struggles, such as the Transformer/Diffusion confusion or the distinction between DenseNet variants.

Extension to cloud and virtualized environments. The experiments in this thesis were conducted in bare metal settings without virtualization or containerization. Investigating whether the proposed attack remains effective under GPU virtualization frameworks, containers, or cloud-based GPU sharing mechanisms such as NVIDIA MIG would be an important step toward assessing the real-world threat surface in cloud deployments.

Cross-GPU transfer learning. Currently, the classifiers must be retrained from scratch for each target GPU model. Exploring transfer learning or domain-adaptation techniques to reduce the amount of data required to adapt a classifier trained on one GPU to a new hardware target would lower the attacker's data collection burden and broaden the attack's applicability.

Empirical evaluation of countermeasures. As discussed in Chapter 6, several potential mitigations could be applied to reduce the effectiveness of the proposed attack. A systematic empirical evaluation of their impact — including the trade-off between the degree of protection and the performance overhead introduced — remains an open problem. In particular, assessing whether timer restriction, noise injection, or pipeline scheduling randomization can reduce classification accuracy below a practically useful threshold without unacceptable performance degradation would be a valuable contribution toward understanding how to defend shared GPU environments against this class of attacks.

Bibliography

[1] J. Ahn, J. Kim, H. Kasan, L. Delshadtehrani, W. Song, A. Joshi, and J. Kim. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 565–577, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480093. URL https://doi.org/10.1145/3466752.3480093.

[2] I. Baddour, S. K. Sanadhya, and D. S. Banerjee. An improved micro-architectural covert-channel attack on gpus. *Journal of Hardware and Systems Security*, 9(3): 149–162, 2025. doi: 10.1007/s41635-025-00170-0.

[3] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker. Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589080. URL https://doi.org/10.1145/3579371.3589080.

[4] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016. doi: 10.1109/HPCA.2016.7446082.

[5] W. Liu, D. Gao, and M. K. Reiter. On-demand time blurring to support side-channel defense. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 210–228, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66399-9.

[6] Y. Miao, Y. Zhang, D. Wu, D. Zhang, G. Tan, R. Zhang, and M. T. Kandemir. Veiled pathways: Investigating covert and side channels within gpu uncore. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1169–1183, 2024. doi: 10.1109/MICRO61859.2024.00088.

[7] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2139–2153, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243831. URL https://doi.org/10.1145/3243734.3243831.

[8] R. Nazaraliyev, Y. Zhang, S. B. Dutta, A. Marquez, K. Barker, and N. Abu-Ghazaleh. Not so refreshing: Attacking {GPUs} using {RFM} rowhammer mitigation. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 5641–5660, 2025.

[9] NVIDIA Corporation. NVIDIA Ada GPU Architecture. Technical report, NVIDIA Corporation, 2022. URL https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf. Accessed: March 2026.

[10] NVIDIA Corporation. *CUDA C++ Programming Guide*, 2025. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: March 2026.

[11] NVIDIA Corporation. *Parallel Thread Execution ISA Reference*, 2025. URL https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: March 2026.

[12] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32648-9.

[13] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[14] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. 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*, CCS '23, page 960–974, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3616672. URL https://doi.org/10.1145/3576915.3616672.

[15] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao. Invalidate+Compare: A Timer-Free GPU cache attack primitive. In *33rd USENIX Security Symposium*

(USENIX Security 24), pages 2101–2118, Philadelphia, PA, Aug. 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-zhenkai.

A | Appendix A

This appendix contains the training metrics and inference results for the pooling layer count regressor, complementing the convolutional layer regression results presented in Section 5.4.1.2.

The pooling layer regressor achieves a validation R^2 score of approximately 0.78 and a Mean Absolute Error of approximately 0.4 pooling layers on the validation set, stabilizing after roughly 20 epochs. When evaluated on the held-out DenseNet-121 traces, the mean absolute error rises to 1.6 pooling layers out of a true value of 5, with errors ranging between 23% and 39% depending on the dataset and execution phase.

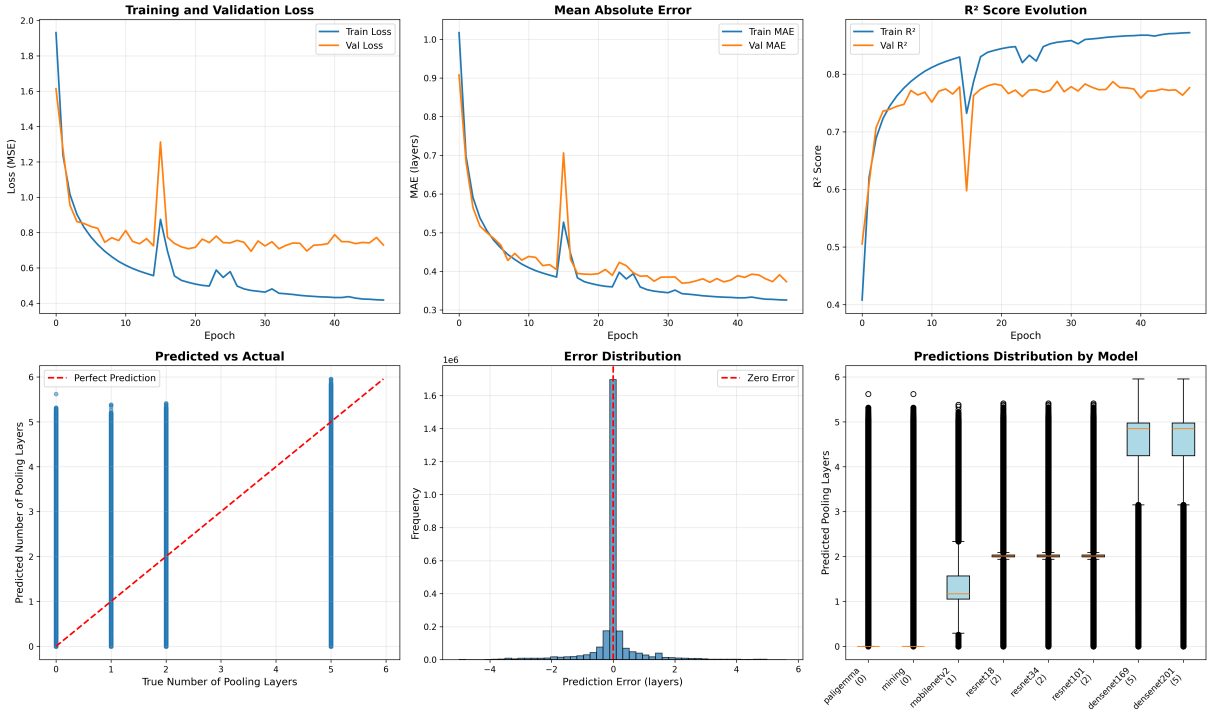


Figure A.1: Training metrics for the pooling layer count regressor.

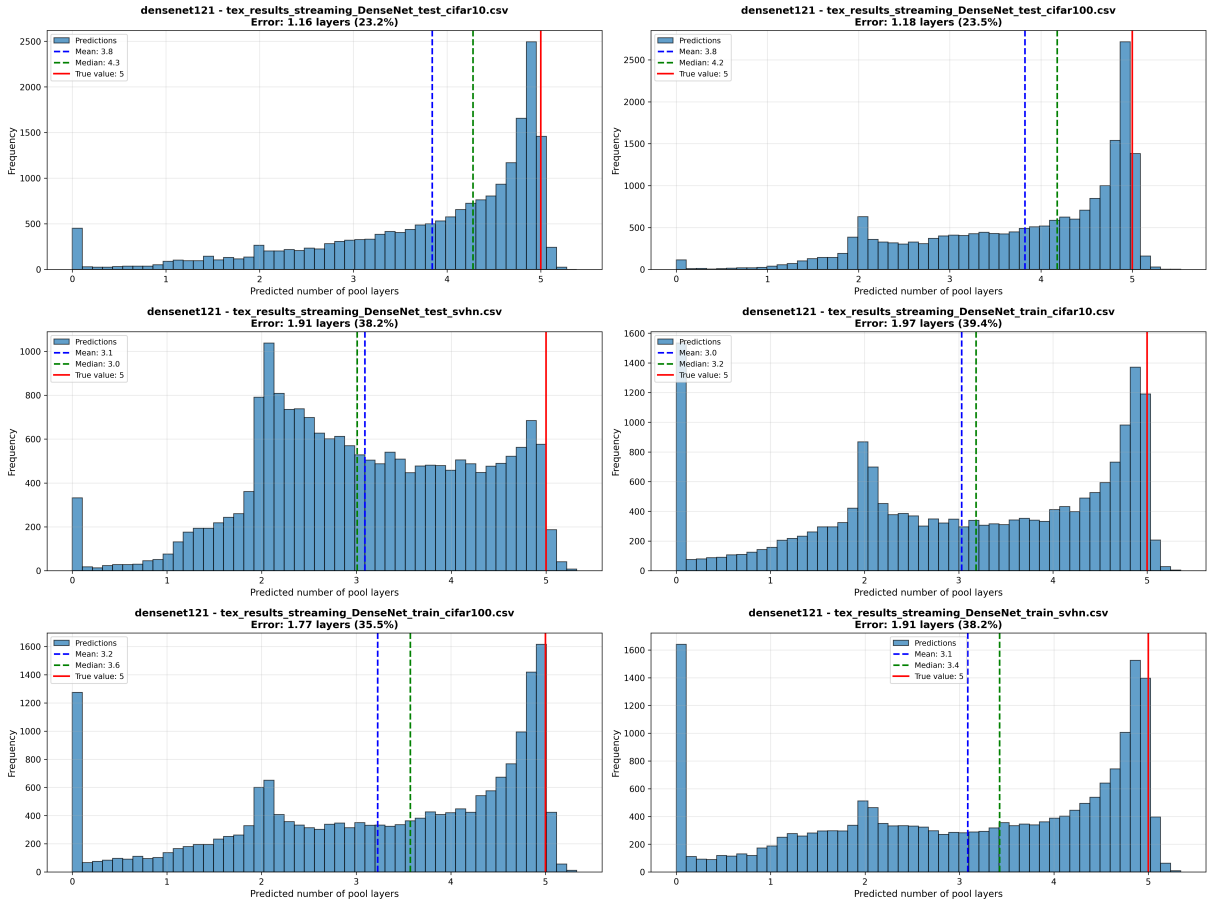


Figure A.2: Distribution of predicted pooling layer counts for DenseNet-121 across different datasets and execution phases.

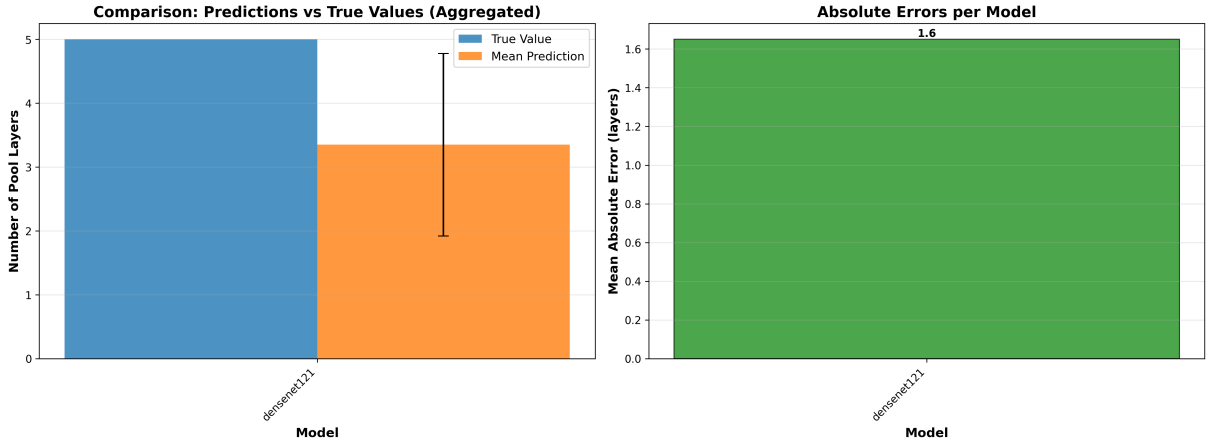


Figure A.3: Aggregated comparison of predicted vs true pooling layer count for DenseNet-121.

List of Figures

2.1 High-level architecture of an NVIDIA Ada Lovelace GPU 6
2.2 Internal architecture of a Streaming Multiprocessor (SM) in the NVIDIA Ada Lovelace architecture 7

5.1 TEX latency comparison for GPT-2 model on NVIDIA RTX 4070 33
5.2 TEX latency comparison for ResNet-18 model on NVIDIA RTX 4070 . . . 34
5.3 TEX latency comparison for Stable-Diffusion-v1-5 model on NVIDIA RTX 4070 35
5.4 TEX latency comparison for LSTM model on NVIDIA RTX 4070 36
5.5 TEX latencies for `DaggerHashimoto` and `NeoScrypt` on NVIDIA RTX 4070 37
5.6 F1 score for the single-stream BiLSTM classifier trained on TEX latencies. 38
5.7 Async latency comparison for GPT-2 model on NVIDIA RTX 4070 40
5.8 Async latency comparison for ResNet-18 model on NVIDIA RTX 4070 . . 41
5.9 Async latency comparison for Stable-Diffusion-v1-5 model on NVIDIA RTX 4070 42
5.10 Async latency comparison for LSTM model on NVIDIA RTX 4070 43
5.11 Async latencies for `DaggerHashimoto` and `NeoScrypt` on NVIDIA RTX 4070 44
5.12 F1 score for the single-stream BiLSTM classifier trained on Async latencies. 45
5.13 Training metrics and confusion matrix for the dual-stream BiLSTM classifier evaluated on all five workload categories. 47
5.14 Confusion matrices for the CNN model-level classifiers. 50
5.15 Training metrics for the convolutional layer count regressor, including loss, MAE, R^2 score, predicted vs actual values, error distribution, and per-model prediction spread. 52
5.16 Distribution of predicted convolutional layer counts for DenseNet-121 across different datasets and execution phases. 53
5.17 Aggregated comparison of predicted vs true convolutional layer count for DenseNet-121, with mean absolute error of 27.9 layers. 54
5.18 Confusion matrix for the Transformer family-level classifier. 55
5.19 Confusion matrices for the Transformer model-level classifiers. 56

5.20 Confusion matrix for the RNN model-level classifier. 57
5.21 Confusion matrix for the Diffusion model-level classifier. 58
5.22 Training metrics and confusion matrices for the dual-stream BiLSTM classifier evaluated on the validation GPUs. 60

A.1 Training metrics for the pooling layer count regressor. 79
A.2 Distribution of predicted pooling layer counts for DenseNet-121 across different datasets and execution phases. 80
A.3 Aggregated comparison of predicted vs true pooling layer count for DenseNet-121. 80

List of Tables

5.1 Complete list of models and datasets used in the evaluation on the RTX 4070. 28
5.2 Category-level classification results across the three GPUs. 59
5.3 Model-level classification results across the three GPUs. 61
5.4 Complete list of models and datasets used in the validation on the RTX 3050 and RTX 5070. 61

List of Symbols

Symbol	Description	Unit
W	sliding window length for trace segmentation	samples
s	stride between consecutive windows	samples
x_i	raw latency measurement at position i within a window	cycles
$\tilde{x}_i$	Z-score normalized latency at position i	—
μ_w	mean of a latency window	cycles
σ_w	standard deviation of a latency window	cycles
ϵ	numerical stability constant	—
C	number of target classes	—
$\mathbf{x}_{\text{tex}}$	TEX latency window fed to the classifier	—
$\mathbf{x}_{\text{async}}$	Async latency window fed to the classifier	—
$\mathbf{f}_{\text{tex}}$	feature vector extracted by the TEX BiLSTM branch	—
$\mathbf{f}_{\text{async}}$	feature vector extracted by the Async BiLSTM branch	—
$\mathbf{h}$	fused hidden representation after the fusion block	—
$\hat{\mathbf{y}}$	output logits of the dual-stream classifier	—
$\mathbf{W}_f$, $\mathbf{b}_f$	weight matrix and bias of the fusion fully connected layer	—
$\mathbf{W}_o$, $\mathbf{b}_o$	weight matrix and bias of the output layer	—
η	learning rate of the Adam optimizer	—
p	Dropout probability	—
lfsr	state of the 32-bit Galois LFSR	—

Acknowledgements

Anche questo lungo percorso giunge infine alla sua conclusione e, se sono arrivato fin qui, dei ringraziamenti sono d'obbligo.

Desidero innanzitutto ringraziare il Prof. Luca Cassano e il mio correlatore Elia Lazzeri per l'opportunità di lavorare in un campo così stimolante e interessante, per i preziosi consigli ricevuti e per la loro grande disponibilità durante quest'anno di lavoro.

Ringrazio inoltre la mia famiglia, che ha sempre creduto in me e nella strada che ho scelto di intraprendere: senza di loro, il mio percorso universitario non sarebbe neanche iniziato.

Un pensiero va infine a tutti coloro che, in un modo o nell'altro, hanno fatto parte di questo percorso — amici, colleghi e compagni di studio — con cui ho condiviso momenti difficili e soddisfazioni. La loro presenza ha reso questo cammino più leggero e più ricco.

A tutti voi, grazie.