UNIVERSITY OF CALIFORNIA,
IRVINE

Securing Processors from Hardware Exploits: A Real-Time Domain Adaptation for Resilient Attack Detection

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Jaya Keshava Chandra Kotha

Dissertation Committee:
Distinguished Professor Jean-Luc Gaudiot, Chair
Professor Nader Bagherzadeh
Professor Chen-Yu (Phillip) Sheu

2026

DEDICATION

To my family and friends, for your unwavering belief in me.

To all the teachers in my life, for showing me the profound joy of learning something new.

And to my students—past, present, and future: this dissertation is a promise kept. I will never stop pursuing knowledge and trying new things, so that I may become a teacher who inspires you to do the same.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Distinguished Professor Jean-Luc Gaudiot, for his unwavering support, invaluable guidance, and continuous mentorship throughout my doctoral studies. His vision and dedication have been instrumental in shaping this research and my development as a scholar. I also extend my sincere thanks to my dissertation committee members, Professor Nader Bagherzadeh and Professor Chen-Yu (Phillip) Sheu, for their insightful feedback, constructive suggestions, and the valuable time they dedicated to reviewing this work.

I would like to extend my gratitude to my colleagues and friends at the PASCAL lab for their friendship, timely comments, and suggestions during my work. Within the lab, I am deeply thankful to Dr. Congmiao Li for her foundational contributions to this field and his direct support on our previous collaborative work. Special thanks also go to Chad Kobe Wong for his instrumental help and collaboration in designing and testing the machine learning models.

I would also like to express special thanks to Adam Boswell, my manager at Amazon prior to the start of my doctoral studies, for his immense support and encouragement during my transition into the PhD program.

Finally, I would like to thank my family and friends for their continuous support and encouragement throughout this journey.

The text of Section 5.2 of this dissertation is an adaptation of the material as it appears in J. Kotha and J.-L. Gaudiot, "A Data-Driven Approach Using Hardware Performance Counters to Detect Micro-Architectural Attacks," in *Proceedings of the Nineteenth IEEE International Workshop on Security, Trust, and Privacy for Software Applications (STPSA 2025) at COMPSAC 2025*, Toronto, Canada, July 2025, pp. 2301–2306, used with permission from IEEE.

This work was supported in part by the National Science Foundation (NSF) under Grant CNS-2026675. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

Declaration of AI Usage: During the preparation of this dissertation and the underlying research, the author utilized artificial intelligence tools, including Google Gemini, ChatGPT, GitHub Copilot, and Grammarly, to support various stages of the project. Specifically, these tools were utilized as programming copilots to assist in the design and development of the experimental codebase, to search for and synthesise information regarding complex technical topics, and to aid in the structural organization, copyediting, and LaTeX formatting of the final manuscript. After utilizing these tools, the author thoroughly reviewed, validated, and refined all generated code and written content, assuming full and sole responsibility for the final outcome and ensuring its academic integrity.

VITA

Jaya Keshava Chandra Kotha

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2026 (Expected)**
University of California, Irvine *Irvine, California*

Master of Science in Electrical and Computer Engineering **2019**
University of California, Irvine *Irvine, California*

Bachelor of Technology in Electronics and Communication **2017**
Karunya University *Coimbatore, India*

RESEARCH EXPERIENCE

Graduate Research Assistant **2022–2023**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2024–2026**
University of California, Irvine *Irvine, California*

PREPRINTS

Characterizing the Variance Envelope: A Multi-Dimensional Analysis of Spectre Telemetry Across Architectures and Workloads **2026**
arXiv preprint arXiv:2608.13920

REFEREED CONFERENCE PUBLICATIONS

A Data-Driven Approach Using Hardware Performance Counters to Detect Microarchitectural Attacks **Jul 2025**
IEEE STPSA (co-located with IEEE COMPSAC)

SOFTWARE & DATASETS

mdme-framework `github.com/jayakeshav/mdme-framework`
The Multidimensional Microarchitectural Evaluation (MDME) framework is an automated orchestration and data collection tool designed to systematically evaluate microarchitectural telemetry across multidimensional interference matrices and adversarial pacing modes. GitHub Link

RT-MAD `github.com/jayakeshav/RT-MAD`
Real-Time Microarchitectural Attack Detection (RT-MAD) is a decoupled client-server architecture designed for live hardware performance counter (HPC) telemetry acquisition and adaptive microarchitectural threat detection. GitHub Link

VizLab `github.com/jayakeshav/vizlab`
A data visualization tool designed to efficiently explore and analyze microarchitectural datasets. It features an integrated dataset as a submodule, streamlining the setup process by eliminating the need for separate data downloads. GitHub Link

Microarchitectural Attack Detection Dataset `doi.org/10.21227/xxx6-v468`
A comprehensive dataset of hardware performance counter (HPC) telemetry for Spectre V1, V2, and benign workloads. Officially published via IEEE DataPort, with a mirror available on GitHub.

ABSTRACT OF THE DISSERTATION

Securing Processors from Hardware Exploits: A Real-Time Domain Adaptation for Resilient Attack Detection

By

Jaya Keshava Chandra Kotha

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2026

Distinguished Professor Jean-Luc Gaudiot, Chair

Speculative execution features allow Spectre attacks to leak sensitive data. Using Hardware Performance Counters (HPCs) to detect these threats at runtime is promising, but standard machine learning models fail in real-world use because they break down due to hardware differences, background noise, and traffic shaping. To address this, our dissertation presents a unified framework combining physical hardware analysis with adaptive machine learning, systematically mapping how Spectre signatures vary across different leakage channels and processors like Intel, AMD, and ARM.

To resolve these hardware-induced domain shifts, we designed a hybrid model that pairs a Domain-Adversarial LSTM (DA-LSTM) to track time-based patterns with an XGBoost classifier for precise decision boundaries, leveraging Source-Free Domain Adaptation (SFDA) so the system can automatically adjust to new, unlabeled hardware without original training data. Live client-server pipeline evaluations prove that this approach restores high-recall detection across diverse systems and matches traditional accuracy while cutting computational retraining time by roughly 85%, turning theoretical detection into a practical blueprint for microarchitectural defense.

Chapter 1

Introduction

While the sharing of hardware resources in modern computer systems drives performance, it fundamentally complicates the strict isolation of security domains. Attackers increasingly exploit this shared microarchitecture through transient execution (Spectre) vulnerabilities, leveraging underlying hardware flaws to compromise otherwise secure software. Because these attacks leave minimal architectural traces, traditional software-level defenses are often insufficient. To resolve this critical gap, this dissertation applies hardware performance counters alongside adaptive machine learning to detect microarchitectural attack footprints, with the primary goal of developing practical, real-time defenses for vulnerable systems.

1.1 Motivation

Modern electronic devices process sensitive information, making data leakage through side channels a major security risk. Attackers exploit variations in timing [17], electromagnetic signals [8], and power consumption [18]. For example, microarchitectural attacks like Spectre [17] and Rowhammer [16] use speculative execution and hardware vulnerabilities to bypass software security. Defending against these attacks usually requires hardware changes

or slow software patches, which are impractical for legacy or performance-critical systems.

To detect these attacks without modifying hardware, researchers use Hardware Performance Counters (HPCs) to monitor system activity at runtime. As new vulnerabilities emerge, defenders deploy Machine Learning (ML) models to analyze HPC data and identify the specific signatures of an attack.

Previous studies [21, 23, 24, 43, 44] report high detection accuracy in isolated lab settings. However, these static ML models regularly fail in real-world systems. This failure occurs because prior work usually evaluates a single attack on a single hardware architecture with no background activity. In practice, different attack methods, background system noise, and hardware differences constantly change the HPC data.

Furthermore, attackers use different side-channel methods, which makes detection more difficult. Most detectors focus only on *Flush+Reload* [42], which measures the timing of shared memory lines, or *Prime+Probe* [28, 31], which measures disturbances across the cache. Because these two methods interact with the hardware differently, a model trained on one method often fails to detect the other.

Recent studies show that current detectors fail under realistic conditions because they are trained in limited environments [20]. The field currently lacks a clear understanding of how Spectre attacks affect HPCs among different workloads and hardware. Without this understanding, models cannot distinguish an attack from normal system activity. To build adaptable models, we must first map how attack signatures change under different operating conditions.

1.2 Threat Model

This dissertation uses a threat model based on realistic conditions for modern shared computer systems. Establishing these boundaries is essential for evaluating the effectiveness of real-time detection pipelines:

- **Attacker Capabilities:** The attacker can run standard, unprivileged applications on the target machine. They do not have administrative or kernel access. This represents a typical scenario, such as a malicious user in a cloud environment sharing physical hardware with a victim.

- **Hardware and OS Assumptions:** The operating system and hardware are secure against traditional software attacks and enforce proper memory isolation. However, the hardware remains fundamentally vulnerable to speculative execution flaws. We assume that completely patching these flaws through software would cause unacceptable performance drops.

- **Leakage Channels:** We focus on cache-based side channels, specifically *Flush + Reload* [42] and *Prime + Probe* [28]. Attackers use these methods to steal sensitive data by measuring timing differences across the shared hardware cache.

1.3 Research Contributions

This dissertation overcomes the limitations of static side-channel detection by proposing an adaptive monitoring framework. This work establishes a clear framework to understand how hardware telemetry changes under different conditions and uses these insights to build a real-time protection system. Ultimately, this approach translates theoretical anomaly detection into a practical, low-overhead solution designed for real-world deployment across diverse hardware. The core contributions are:

- **Analyzing Hardware Variations:** We use a data-driven methodology to analyze how hardware telemetry shifts across three attack mechanisms (Spectre V1 and V2 [17]) and two leakage channels (Flush+Reload [42] and Prime+Probe [28]). We evaluate these shifts across Intel, ARM, and AMD processors, accounting for differences in how each architecture defines hardware counters and the limits of cross-device observation.

- **Measuring System Noise and Attacker Evasion:** We evaluate how detection signals degrade under heavy background workloads (Idle, CPU, Memory, and Mixed) to identify the most noise-resistant metrics. We also quantify how attacker evasion tactics, such as rate-limiting or burst execution, affect the stability of the hardware telemetry.

- **Designing an Adaptive Machine Learning Model:** We design a machine learning model that adapts to changing system environments. This model combines a Domain-Adversarial Long Short-Term Memory (DA-LSTM) network to track time-based features with an eXtreme Gradient Boosting (XGBoost) classifier for precise detection. By utilizing Source-Free Domain Adaptation (SFDA/SHOT), this architecture practically eliminates the costly requirement of collecting large, labeled datasets every time the system encounters new hardware.

- **Engineering a Live Client-Server Detection Pipeline:** We build and validate a real-time detection pipeline that separates data collection from analysis. A lightweight client collects data on the target system, while a dedicated server handles the heavy machine learning inference. This architecture provides accurate runtime detection with minimal performance overhead on the monitored host.

- **Demonstrating Computational Efficiency and Practical Viability:** We prove the real-world usability of this framework by evaluating its operational cost. Our adaptive retraining method maintains or exceeds traditional detection accuracy while

achieving an approximate 85% reduction in computational retraining time compared to full-dataset retraining, making continuous, real-time protection highly viable in production environments.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows:

- **Chapter 2** provides the theoretical foundation for Spectre variants, microarchitectural side channels, and hardware performance counters (HPCs), alongside a review of existing detection methods.

- **Chapter 3** details the unified experimental methodology and real-time system architecture. This includes the automated data orchestration workflow, time-series data alignment, background interference generation, and the design of the decoupled, asynchronous client-server pipeline for telemetry acquisition and adaptive inference.

- **Chapter 4** explains how attack signatures change under different conditions. It analyzes these variations across different hardware architectures, temporal footprints, and environmental noise levels.

- **Chapter 5** transitions from static classification to adaptive machine learning. It details the mathematical formulation and structural design of the hybrid DA-LSTM and XGBoost model.

- **Chapter 6** presents the evaluation of our detection framework. It verifies the model's effectiveness in offline mode across varying domains and demonstrates the system's ability to adapt in real-time scenarios.

- **Chapter 7** summarizes the contributions of this research, acknowledges the limits of

our intra-device validation, and suggests future research directions.

Chapter 2

Background and Related Work

This chapter introduces the foundational concepts needed to understand Spectre vulnerabilities and hardware-based monitoring. It also reviews the evolution of Hardware Performance Counter (HPC) security applications and existing detection methods.

2.1 Background

To understand why static detection models fail across different computing environments, we must first establish how Spectre attacks generate observable telemetry. This section details the distinct phases of a speculative attack—from the initial branch misprediction to cache-based data exfiltration—and explains how underlying architectural differences affect the resulting Hardware Performance Counter (HPC) signatures.

2.1.1 Speculative Execution and Side-Channel Attacks

Speculative execution is a microarchitectural optimization that allows modern processors to improve performance by predicting and executing instruction paths before confirming them. While this optimization improves speed, it introduces severe security risks by leaving

observable microarchitectural side effects, such as changes in the cache state. The branch predictor forecasts likely execution paths based on historical patterns. If a prediction is incorrect, the processor flushes the pipeline and rolls back the architectural state. However, it does not revert the residual microarchitectural changes, which attackers can then exploit.

Spectre, as documented by Kocher *et al.* [17], exploits speculative execution to leak sensitive information. While Intel later identified numerous variants (collectively termed Spectre-NG [40], including attacks from virtual machines and return stack buffer exploits [29]), the two primary attack vectors rely on manipulating different predictors.

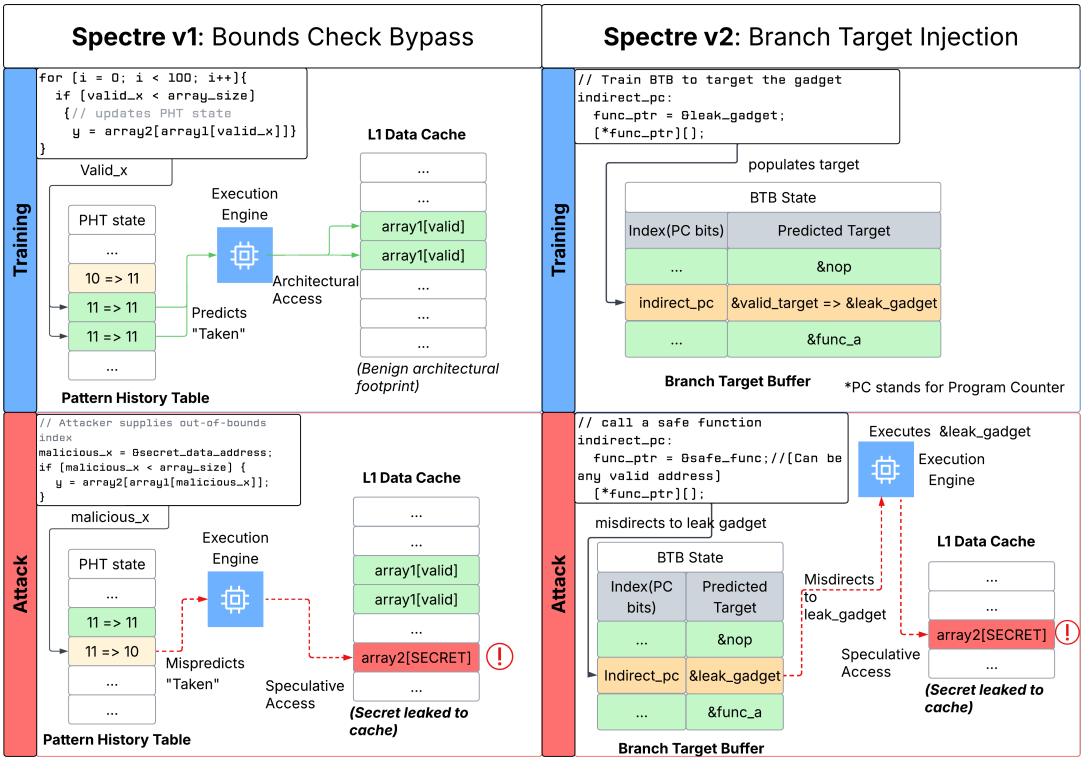


Figure 2.1: Microarchitectural hardware schematic contrasting primary Spectre variants: Spectre V1 (left) and Spectre V2 (right).

Spectre V1 (Bounds Check Bypass) [17, 1, 30] manipulates conditional branch mispredictions, forcing the processor to speculatively execute instructions past a bounds check. As illustrated in Figure 2.1 (left), the attacker trains the Pattern History Table (PHT) to trigger speculative access to a secret. This access appears as an anomalous spike in conditional

branch mispredictions. Conversely, Spectre V2 (Branch Target Injection) [17, 19] poisons indirect branch predictors, such as the Branch Target Buffer (BTB). As detailed in Figure 2.1 (right), this misdirects execution to a leak gadget based on past patterns, shifting the microarchitectural footprint toward metrics associated with indirect branch speculation.

2.1.2 Microarchitectural Leakage Channels

Once a processor speculatively accesses sensitive data, the attacker must transmit it via a covert channel. While all Spectre-family attacks use speculative execution, their microarchitectural footprints differ significantly depending on the chosen exfiltration channel.

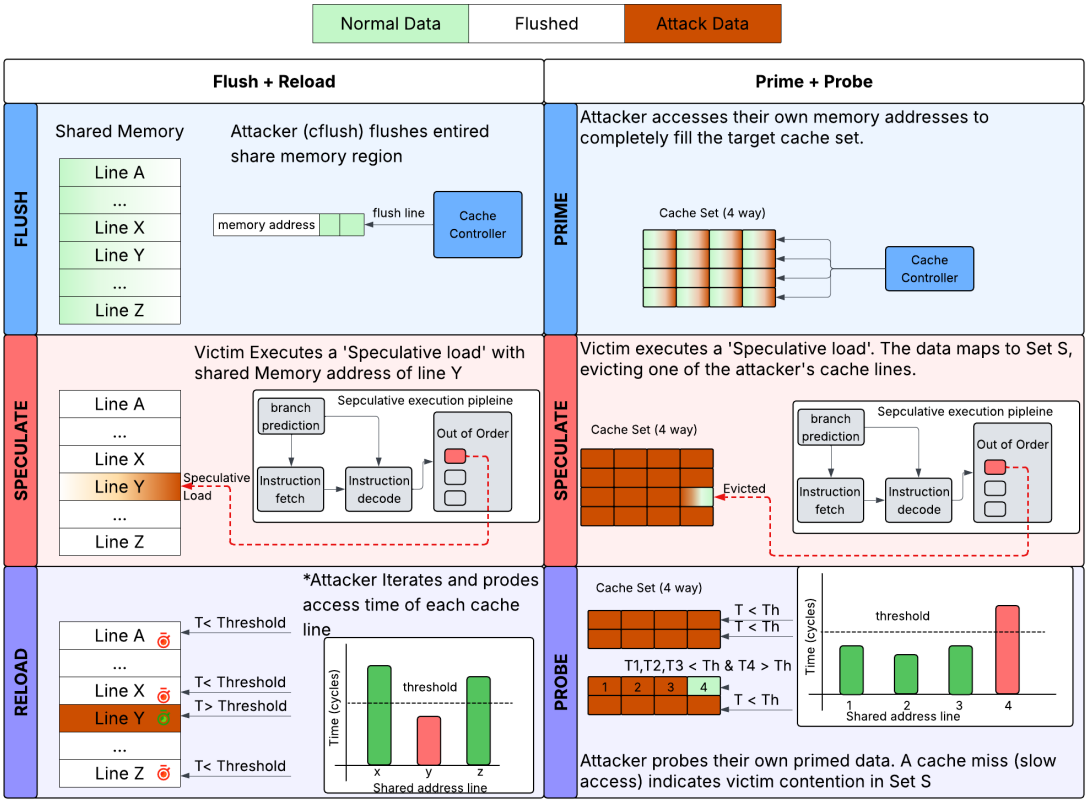


Figure 2.2: Comparison of cache side-channel leakage mechanisms: Flush+Reload (left) and Prime+Probe (right).

The *Flush+Reload* channel [42, 11] relies on the timing differences of shared memory lines. As shown in Figure 2.2 (left), the attacker flushes memory (e.g., using the `cflush` instruction),

waits for the victim's speculative execution to load the target memory, and measures the reload time to determine if the secret is in the cache. This generates a distinct signature of cache hits and misses. In contrast, *Prime+Probe* [33, 28, 38] infers data by measuring set-based disturbance across the cache hierarchy. As depicted in Figure 2.2 (right), the attacker primes a cache set with their own data and later probes it; a cache miss indicates that the victim's speculative execution evicted the primed data. This reliance on eviction set construction and broader cache contention significantly alters the resulting telemetry profile.

2.1.3 Hardware Performance Counters (HPCs)

HPCs are specialized registers in modern microprocessors that count hardware-related activities. Originally designed for debugging and performance tuning, recent studies [6, 24] show that these counters are also valuable for security monitoring. Malicious activities leave dynamic, recognizable patterns in HPC telemetry. For example, attackers who flush caches to manipulate data create artificially high cache miss rates, while those manipulating branch predictors directly increase misprediction rates.

However, because HPCs are microarchitectural features, their semantics depend heavily on the underlying hardware. A major factor affecting side-channel observability is the cache inclusion policy [41]. In inclusive hierarchies (common in many Intel processors), the Last Level Cache (LLC) acts as a directory for all lower-level lines. Saturating an LLC set triggers an automatic "back-invalidation" that forces data out of the L1 cache, creating a large, easily detectable HPC signature [24]. In contrast, alternative topologies (such as the banked L2 structures in some AMD architectures) enforce strict physical memory routing. This routing misaligns standard eviction sets and severely limits observability across different platforms.

2.2 Related Work in HPC-Based Detection

The application of Hardware Performance Counters (HPCs) for security has evolved through distinct research phases, transitioning from general malware classification to the current recognition of signature fragility.

2.2.1 Malware Classification and Speculative Defense

Early feasibility studies demonstrated that HPCs can identify anomalous execution patterns without the overhead of software instrumentation. Tang *et al.* [37] pioneered this approach, using unsupervised machine learning to classify general malicious code signatures. Following the disclosure of Spectre vulnerabilities, researchers adapted these techniques for Spectre and Meltdown. Li *et al.* [25] established foundational proof-of-concept thresholds, showing that hardware vulnerabilities leave distinct footprints in cache and branching metrics. Kotha *et al.* [21] expanded on this work, proving the feasibility of high-accuracy Multi-Layer Perceptrons (MLPs) for detecting unthrottled speculative bursts. While these early works proved the viability of ML-based detection, they primarily evaluated models under isolated, ideal conditions. These controlled environments fail to account for signal deterioration caused by real-world system noise.

To address the limitations of single-device evaluations, subsequent research explored cross-platform telemetry. Larson *et al.* [23] conducted foundational cross-platform measurements for edge devices, establishing that HPC behavior diverges across architectures. However, their work focused on general malware and did not map the specific evasion space of Spectre vulnerabilities or evaluate adversarial adaptation.

2.2.2 The SoK Era: Recognizing Fragility

Recognizing that sophisticated attackers attempt to evade static thresholds, Li *et al.* [24] introduced the concept of "Adversarial Pacing," proving that attackers can bypass detec-

tion by artificially limiting their execution throughput. More recently, a Systematization of Knowledge (SoK) by Kosasih *et al.* [20] analyzed 50 prior works, concluding that most proposed HPC detectors remain critically vulnerable. The SoK identified a systemic failure in the literature to account for domain shifts, highlighting the lack of realistic threat models that combine architectural variance, environmental noise, and adversarial pacing.

2.2.3 Research Gap and Contributions of this Work

As summarized in Table 2.1, this dissertation directly answers the SoK's critique by addressing a fundamental research gap: the absence of a multidimensional threat model evaluation. We provide the first unified empirical characterization that simultaneously maps cross-ISA drift, multi-variant signatures, system noise interference, and adversarial pacing. Furthermore, we quantify the detection collapse of standard models under these conditions, providing the realistic threat model evaluation missing from the current state of the art. Finally, to overcome this degradation, we introduce an adaptive machine learning architecture designed to dynamically maintain detection resilience against these environmental and adversarial shifts.

Table 2.1: Comparison of the Characterization Matrix Against Prior HPC Security Literature

Reference	**CI**	**MV**	**SN**	**AP**	**AM**	**Primary Objective**
Tang *et al.* [37]	✗	✗	✗	✗	✗	General anomaly classification
Li *et al.* [25]	✗	✗	✗	✗	✗	Proof-of-concept ML thresholds
Larson *et al.* [23]	✓	✗	✗	✗	✗	Basic cross-platform measurement
Kotha *et al.* [21]	✗	✗	✓	✗	✗	High-accuracy MLP feasibility
Kosasih *et al.* [20]	✗	✓	✗	✗	✗	Systematization of ML failures
Li *et al.* [24]	✗	✗	✓	✓	✗	Attacker throughput vs. detection
This Work	✓	✓	✓	✓	✓	**Empirical characterization and real-time adaptive mitigation**

Note: CI = Cross-ISA, MV = Multi-Variant, SN = System Noise, AP = Adversarial Pacing, AM = Adaptive Modeling.

Chapter 3

Unified Experimental Methodology

To assess the microarchitectural variations of Spectre attacks and build an effective detection pipeline, we require a highly controlled yet extensive experimental framework. Instead of evaluating a single threat on one architecture, this dissertation uses a unified experimental matrix spanning multiple attack variants, leakage channels, hardware architectures, and environmental noise factors. We evaluate three distinct attack profiles (Spectre V1 with Flush+Reload, Spectre V2 with a Flush+Reload-like timing leak, and Spectre V1 with Prime+Probe) across a multidimensional matrix of hardware topologies, environmental interference, and adversarial traffic shaping.

3.1 Target Threat Variants and Implementation

To test how well our models detect different hardware attacks, we built three proof-of-concept (PoC) programs. These represent the primary methods Spectre attacks use to steal data. We adapted them from standard research code and modified them to support the attack speeds described in Section 3.2. All programs use the same command-line tools to control their speed (such as burst sizes, batch intervals, and characters per second), ensuring timing

stays consistent across all tests.

3.1.1 Spectre V1 (Bounds Check Bypass) with Flush+Reload:

Our baseline Spectre V1 attack tricks the processor's branch predictor to steal data using a "Flush+Reload" method. First, it clears a shared array (`array2`) from the cache. Then, it trains the processor with safe memory accesses before tricking it into reading a secret memory location out of bounds. The processor temporarily uses this secret to access a specific part of the shared array, loading it into the cache. The attacker then times how long it takes to read the array again; a fast read indicates a cache hit, revealing the secret. We tuned the threshold for a "fast" read for each processor (for example, 200 cycles on the ARM testbed).

3.1.2 Spectre V1 with Prime+Probe:

We also built a "Prime+Probe" version to see how different cache techniques affect detection. Unlike Flush+Reload, this method does not require shared memory. Instead, the attacker fills (primes) specific parts of the cache with its own data. When the tricked processor reads the secret, it accidentally overwrites some of the attacker's primed data. The attacker then checks (probes) the data again. A longer read time indicates a cache miss, revealing which part of the cache the victim used and exposing the secret.

3.1.3 Spectre V2 (Branch Target Injection):

Our Spectre V2 attack targets a different part of the processor: the indirect branch predictor. The attacker trains the Branch Target Buffer (BTB) to jump to a specific malicious code snippet (a "gadget"). When the victim runs its normal code, the poisoned BTB tricks it into jumping to the attacker's gadget instead. This gadget hides the secret data in the cache before the processor catches the mistake and stops. The attacker then uses timing analysis to recover the secret data.

3.1.4 Cross-Architecture Portability and System-Level Mechanics:

While the core concepts of these attacks remain consistent across processors, the specific instructions depend on the hardware. The x86-64 versions (for Intel and AMD) use specific instructions like `_mm_clflush` to clear the cache and `__rdtscp` for precise timing.

When migrating the Flush+Reload attack to the ARM Cortex-A76 processor, we made several system-level changes:

- **Cache Maintenance:** We replaced the x86 `clflush` with the ARM `dc_civac` instruction to clear the cache. To ensure memory operations finish before training the processor, we added a Data Synchronization Barrier (`dsb_ish`).

- **Cycle Measurement:** ARM Linux systems usually restrict access to high-precision timers. To resolve this, our code uses the Linux kernel's performance monitor. We use a `perf_event_open` system call to count hardware CPU cycles. The attacker turns this counter on and off around the cache probe using `ioctl` commands and reads the exact cycle count using `read()`.

- **Pacing Constraints:** Because system calls for timing require more overhead than direct hardware instructions, the speed modes (Burst-All and Batch-Rate) operate differently on ARM. For example, we limited ARM bursts to 20 characters per cycle to keep the system stable and prevent pipeline stalls, whereas x86 can handle larger bursts.

In spite of these hardware differences, the overall logic, speeds, and memory operations stay uniform across all platforms. This ensures the data we collect from the Hardware Performance Counters (HPC) is valid for comparison

3.2 Adversarial Pacing and Traffic Shaping

To simulate a sophisticated attacker attempting to hide within normal system traffic, we control the speed of the attacks. We mix four different speed modes (pacing modes) to evaluate how well our models detect attacks that change their frequency and duration:

- **Unthrottled Mode:** The attack runs as fast as possible without any limits. This serves as our baseline, showing the maximum impact the attack can have on the hardware.

- **Constant Rate Mode:** The attack steals a set number of characters per second. It balances the speed by adding a tiny delay (measured in microseconds) after every secret read. This spreads out the attack traffic to make it appear more normal.

- **Burst-All Mode:** The attack steals all the target data as fast as possible in one quick burst, and then sleeps for the rest of the second. This tests if our models can detect a sudden spike of activity followed by silence.

- **Batch-Rate Mode:** The attack breaks the reads into small, fixed-size batches. It spaces these batches out evenly to match a specific speed target. This imitates the regular, repeating patterns of normal background programs.

3.3 Hardware Testbed Matrix

We deploy our attack implementations across four distinct microarchitectures, representing a spectrum of speculative depths, cache replacement policies, and instruction sets:

- **Intel Core i7-3537U (Ivy Bridge):** Serves as our primary baseline representing deep-speculation pipelines with strictly inclusive Last-Level Caches (LLC). *(Referred to hereafter and in figures as Intel 1).*

- **Intel Core i5-7200U (Kaby Lake):** Acts as a secondary inclusive-cache baseline to isolate and quantify intra-ISA (generational) telemetry variance. *(Referred to hereafter and in figures as Intel 2).*

- **ARM Cortex-A76:** Represents modern RISC architectures featuring a *non-inclusive* L3 victim cache, allowing the study of cross-ISA telemetry shifts in decoupled hierarchies.

- **AMD A4-5000 (Jaguar):** Represents embedded x86-64 microarchitectures that only feature a cache hierarchy up to L2. This strictly banked, inclusive L2 cache topology acts as the foundation for our hardware-intrinsic mitigation analysis.

Table 3.1 summarizes the exact hardware specifications, cache topologies, and software environments for all evaluated platforms.

Table 3.1: Experimental Testbed Specifications

Processor	**Micro - Architecture**	**Cores / Threads**	**Cache (L1d / L2 / L3)**	**OS & Kernel**
Intel Core i7-3537U	Ivy Bridge (22nm)	2C / 4T	32KB / 256KB / 4MB (Inclusive)	Kali 2025.3 (Linux 6.12)
Intel Core i5-7200U	Kaby Lake (14nm)	2C / 4T	32KB / 256KB / 3MB (Inclusive)	Kali 2025.3 (Linux 6.12)
ARM Cortex-A76	ARM v8.2-A (7nm)	4C / 4T	64KB / 512KB / 2MB (Non-Inclusive)	Debian 12 (Linux 6.1)
AMD A4-5000	Jaguar (28nm)	4C / 4T	32KB / 2MB (Inclusive [4-Bank]) / None	Kali 2025.3 (Linux 6.16)

Hardware Controls and Scalability. We selected these platforms as controlled baselines to isolate unfiltered hardware behavior. Modern processors frequently include hidden microcode patches (such as branch history flushing) that mask raw hardware signals. To accurately map the unmitigated footprint of the attacks, we require systems completely free

of these black-box filters. Most importantly, the primary architectural difference we evaluate—inclusive versus non-inclusive cache design—remains highly relevant today. Modern enterprise servers (such as Intel Xeon Scalable and AMD Zen) have shifted toward non-inclusive or exclusive caches to manage massive memory capacities. Therefore, the hardware dynamics we isolated on our test machines apply directly to today's high-performance processors.

3.4 Environmental Noise and System Load Simulation

To prevent the machine learning models from overtraining on a pristine environment, we captured telemetry under multiple levels of synthetic system load. To neutralize scheduling artifacts, our background load engine uses NumPy and SciPy. These libraries explicitly release the CPython Global Interpreter Lock (GIL), allowing background tasks to execute as unthrottled concurrent threads across separate physical cores to induce four distinct architectural states:

- **Idle:** The baseline condition runs with no active interference threads, serving as the optimal, quiet environment for the attacker.
- **CPU-Intensive:** A continuous, multi-threaded workload that computes dot products and calculates eigenvalues on random floating-point matrices (base size of 200×200). This introduces severe arithmetic and scheduling contention.
- **Memory-Pressure:** An array-manipulation workload that continuously allocates, sorts, and computes Fast Fourier Transforms (FFTs) on large floating-point arrays (10^6 elements). The generator maintains a rolling buffer of the 10 most recent arrays to force continuous cache-line replacement and thrash the LLC.
- **Mixed Workload:** A highly variable workload intended to model realistic, multi-

faceted behavior. It rapidly cycles through pseudo-inverse matrix calculations, normal distribution statistics (mean and standard deviation), and 1D FFT signal processing.

Benign Baseline Workloads: When establishing the non-malicious "Benign" execution baselines evaluated in later sections, we avoid using simple idle sleep states. Instead, the framework runs a diverse suite of standard system utilities to model legitimate application telemetry. These workloads include `stress-ng` for targeted CPU and virtual memory pressure, `openssl sha256` for cryptographic hashing, `gcc` for source compilation, and `tar`/`dd` for heavy file I/O operations.

3.5 Data Collection and Telemetry Infrastructure

3.5.1 Performance Counter Selection and Sampling

To capture the microarchitectural footprint of the attacks, we deployed the `perf` tool on our primary testbed, designated as *intel1*. In our initial methodologies, we utilized `perf record` in event sampling mode at 5-millisecond intervals; however, this approach incurred significant system overhead and precluded the ability to stream telemetry data in real-time. Each recorded row represents 20 total metrics (including temporal and environmental data) monitored across the experimental runs published in our first work[21].

To support live data ingestion for the real-time detection pipeline, we transitioned to using `perf stat`. We recorded aggregate hardware counter metrics at 50-millisecond intervals. This optimized interval substantially reduces observational overhead whilst maintaining sufficient temporal resolution for live analysis.

Table 3.2 lists the 16 core global hardware events monitored during this phase, evenly categorized into cache and branch metrics. These global metrics offer a basic description of processor behavior: the cache events track memory hierarchy contention (e.g., L1/L2 data

load misses and Last Level Cache responses), while the branch events monitor speculation patterns (e.g., conditional branch execution and retired mispredictions).

Table 3.2: Monitored Events: Initial Cache and Branch Metrics considered on the intel1 device.

Cache Events	**Branch Events**
Cache References	Branch Instructions
Cache Misses	Branch Misses
L1 Data Cache Loads	Executed Branch Instructions
L1 Data Cache Load Misses	Executed Conditional Branches
L2 All Demand Data Reads	Executed Branch Mispredictions
L2 Demand Data Read Hits	Retired Branch Mispredictions
Offcore All Data Reads LLC Any Response	Retired Near-Taken Branch Mispredictions
Offcore All Data Reads LLC Miss to DRAM	Retired Conditional Branches

3.5.2 Automated Orchestration Workflow (`batch_auto`)

To ensure repeatability and strict time synchronization, we coordinated the data collection using a custom automation package called `batch_auto`. As illustrated in Figure 3.1, this pipeline executes a complete, hands-off workflow divided into distinct phases to enforce strict isolation between experimental runs.

Phase 1: Configuration. The workflow begins with a declarative JSON configuration file where the researcher defines the experiment's state space, including attack targets, input parameters, and interference matrices. The framework then performs Cartesian parameter expansion to generate every possible combination of these variables for testing systematically.

Phase 2: Initialization. Before execution, the pipeline verifies the hardware target against a centralized device profile to ensure hardware performance counter compatibility. It then initializes a master log CSV file to track the progress and outcome of all subsequent runs

Phase 3: Active Loop. For each parameter combination generated in Phase 1, the pipeline executes a strictly controlled run-level loop . First, it enforces a microarchitectural reset by clearing CPU states (such as residual cache lines and branch predictor entries) to prevent data leakage from previous runs, followed by a mandatory stabilization sleep. Next, it spawns a noise thread, utilizing an `InterferenceManager` to inject deterministic background pressure (e.g., CPU-intensive matrix multiplication or memory-thrashing FFTs).

Once the system stabilizes under the targeted noise, the pipeline executes the attack binary and captures telemetry across sequential batches using `perf stat`. After the attack completes, the framework terminates the background noise, logs the run outcome to the master log, and enforces a cooldown gap before looping to the next parameter combination. This rigid coupling guarantees that the sampled HPC telemetry perfectly corresponds to the specific combination of attack family, pacing model, and system load.

3.5.3 Data Alignment and Postprocessing

To prepare the raw time-series telemetry for machine learning ingestion, the data undergoes **Phase 4: Post-Processing**. This multi-stage alignment pipeline bypasses hardware performance counter multiplexing limits by utilizing a Multi-Run Serial Profiling model, where each experiment executes three times sequentially to monitor non-overlapping subsets of events

During post-processing, the pipeline first parses the raw text outputs from `perf stat` into standard tabular formats. Next, it performs horizontal index merging. Instead of relying on absolute timestamps—which vary between sequential runs—the pipeline merges the three batches by row index. This arranges the data according to the program's execution progress, forming a unified high-dimensional time-series vector as if all events were recorded simultaneously

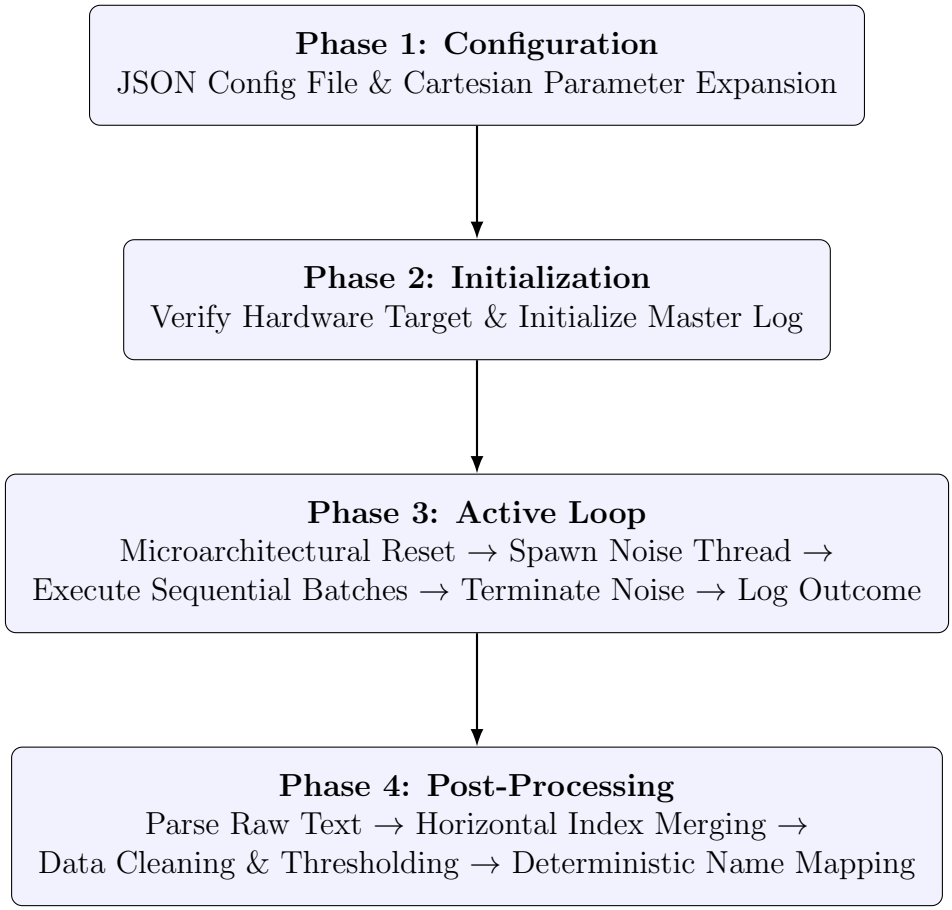


Figure 3.1: The four-stage execution pipeline of the `batch_auto` orchestration framework .

The datasets then undergo data cleaning and thresholding. The framework drops entirely zero-value columns and truncates sparse rows (where all counters are zero) to eliminate idle pre-run and post-run periods. Additionally, continuous memory state variables are binarized based on occupancy thresholds (e.g., converting continuous cache values into binary hit/miss indicators).

Finally, the pipeline applies deterministic name mapping. The cleaned files are renamed and saved into a standardized dataset folder, with physical metadata (such as interference mode, attack flag, and run number) automatically burned into the filename to prevent manual labeling errors and guarantee reproducibility

3.6 Real-Time Client-Server Detection Architecture

In order to enable sub-second threat classification without impeding host data acquisition, the detection pipeline utilizes a decoupled, asynchronous client-server model connected through WebSockets with Socket.IO. This design separates high-frequency telemetry collection on the host from resource-intensive machine learning inference conducted on an isolated server.

3.6.1 Client-Side Telemetry Acquisition

The client-side telemetry daemon acquires data using a non-blocking background thread, which enables uninterrupted monitoring. In live monitoring mode, it initiates a persistent Linux `perf stat` subprocess with 50-millisecond sampling intervals. The client parses the resulting output, aggregates hardware counters, and standardizes event terminology using token overlap matching to maintain compatibility with the server's expected feature space. Subsequently, the structured telemetry payloads are transmitted to the server.

To maintain data integrity and resilience, the client continuously monitors network health.

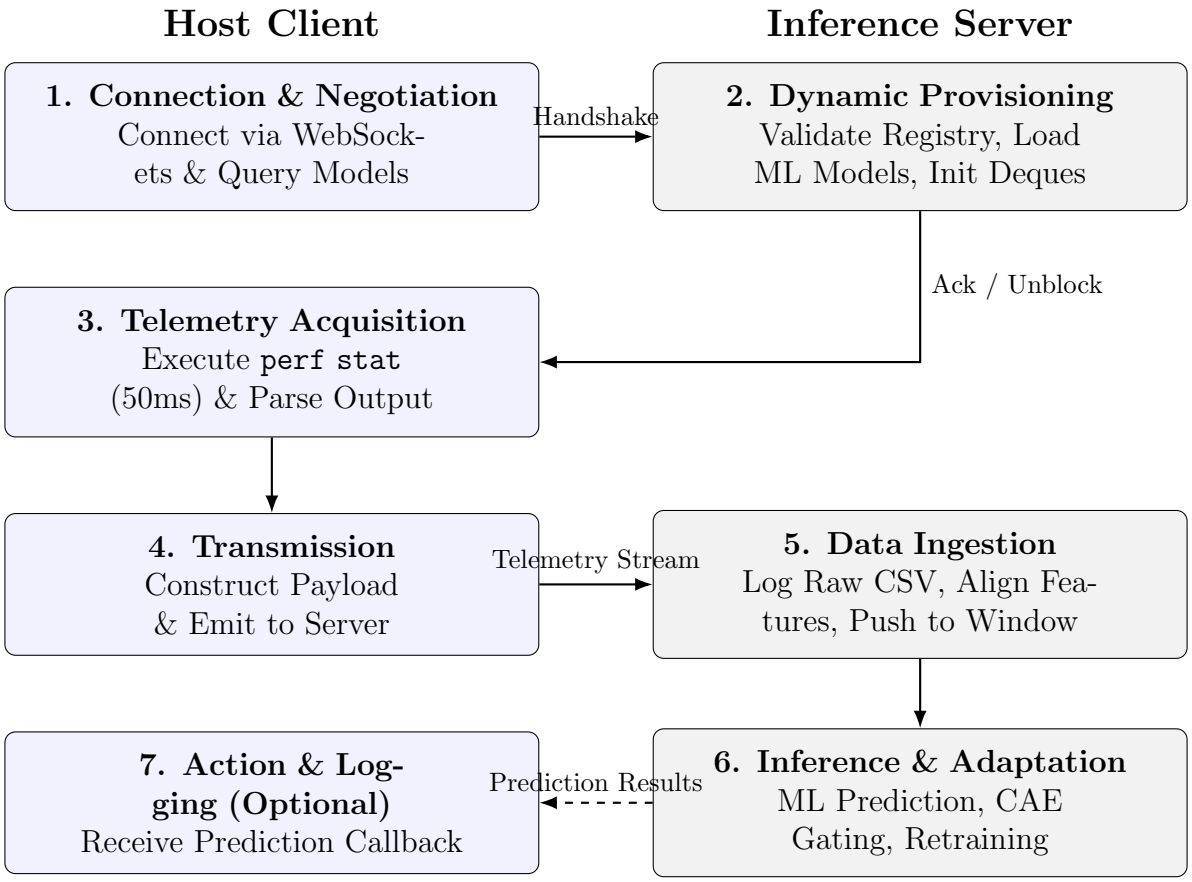


Figure 3.2: Execution workflow of the asynchronous client-server architecture. The server broadcasts inference results back to the host client for optional mitigation or local logging.

If a Socket.IO transmission exception is detected, the client buffers data locally and subsequently transmits the cached telemetry samples in chronological order once connectivity is restored. Additionally, the client supports a CSV replay mode for simulation or offline testing, emitting payloads at a user-defined delay interval.

3.6.2 Server-Side Ingestion and Inference

After establishing a connection and verifying the client against a centralized device registry, the Flask-SocketIO inference server dynamically provisions the session. The server then loads the requested machine learning model (such as XGBoost, LSTM, Domain Adaptation, or Hybrid) and allocates sliding window memory structures using double-ended queues (`collections.deque`).

As raw telemetry data streams in, the server logs the data to disk and aligns the payload features to match the model's expected canonical order. When the sliding window deque reaches its configured capacity, the server extracts the sequence, shapes the feature matrix, and initiates the classification routine. The resulting predictions, which include labels, confidence probabilities, and inference speed, are immediately broadcast to the host client, saved to a runtime prediction log, and published to a live web dashboard channel. Following prediction, the server continuously removes the oldest elements by the configured stride size to advance the sliding window.

3.7 Assessment Criteria

To evaluate the classification performance of our detection models, we utilize five standard statistical metrics derived from the confusion matrix: Accuracy, Precision, Recall, the F1-Score, and the Area Under the Curve (AUC). Because monitoring microarchitectural telemetry requires balancing high detection rates against operational disruption, these metrics help assess both the reliability and the correctness of the anomaly classifier.

- **Accuracy:** Measures the overall proportion of correct predictions (both true positives for active attacks and true negatives for benign states). It provides a baseline indicator of overall system correctness:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

- **Precision:** Quantifies the exactness of the classifier by calculating the ratio of true attack detections out of all instances flagged as malicious. High precision means fewer false alarms, which is critical for preventing unnecessary interference in production environments:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

- **Recall (True Positive Rate):** Measures the completeness of the detection model by determining the proportion of actual attack instances that are correctly identified by the system. High recall ensures that stealthy or pacing-optimized transient execution attacks are not missed:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.3}$$

- **F1-Score:** Represents the harmonic mean of Precision and Recall. This single metric provides a balanced evaluation, which is especially useful given the heavy class inequalities between normal system background noise and sporadic attack phases:

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

- **Area Under the Curve (AUC):** Evaluates how well the model distinguishes between classes across all possible thresholds. The Receiver Operating Characteristic (ROC) AUC computes the two-dimensional area under the curve plotted by the True Positive Rate against the False Positive Rate. It provides a robust, threshold-independent measure of the classifier's ability to separate malicious transient execution from benign baseline states.

Here, TP represents True Positives, TN represents True Negatives, FP represents False Positives, and FN represents False Negatives.

Chapter 4

The Variance Envelope: Multi-Dimensional Characterization

To establish a clear baseline for domain adaptation, we first examine the ground-truth Hardware Performance Counter (HPC) telemetry of Spectre attacks, defined here as the signatures generated by unthrottled exploits executing on an isolated, idle system.[1] By decoupling these operational layers, we isolate the specific variance induced by the hardware architecture, the leakage mechanism, and the adversary's defensive evasion strategy.

4.1 Attack Signatures

An attacker may alter their exploit strategy on a specific, fixed hardware target to bypass signature-based monitors. To quantify this intra-ISA variance, three distinct attack variants are evaluated on a single platform (the Intel Core i5-7200U) under idle conditions: Spectre V1 via Flush+Reload (`V1_fr`), Spectre V2 via Flush+Reload (`V2_fr`), and Spectre V1 via Prime+Probe (`V1_pp`). Because these variants manipulate different components of the

[1] Portions of this chapter are based on the preprint [22].

underlying microarchitecture, they generate distinct telemetry fingerprints.

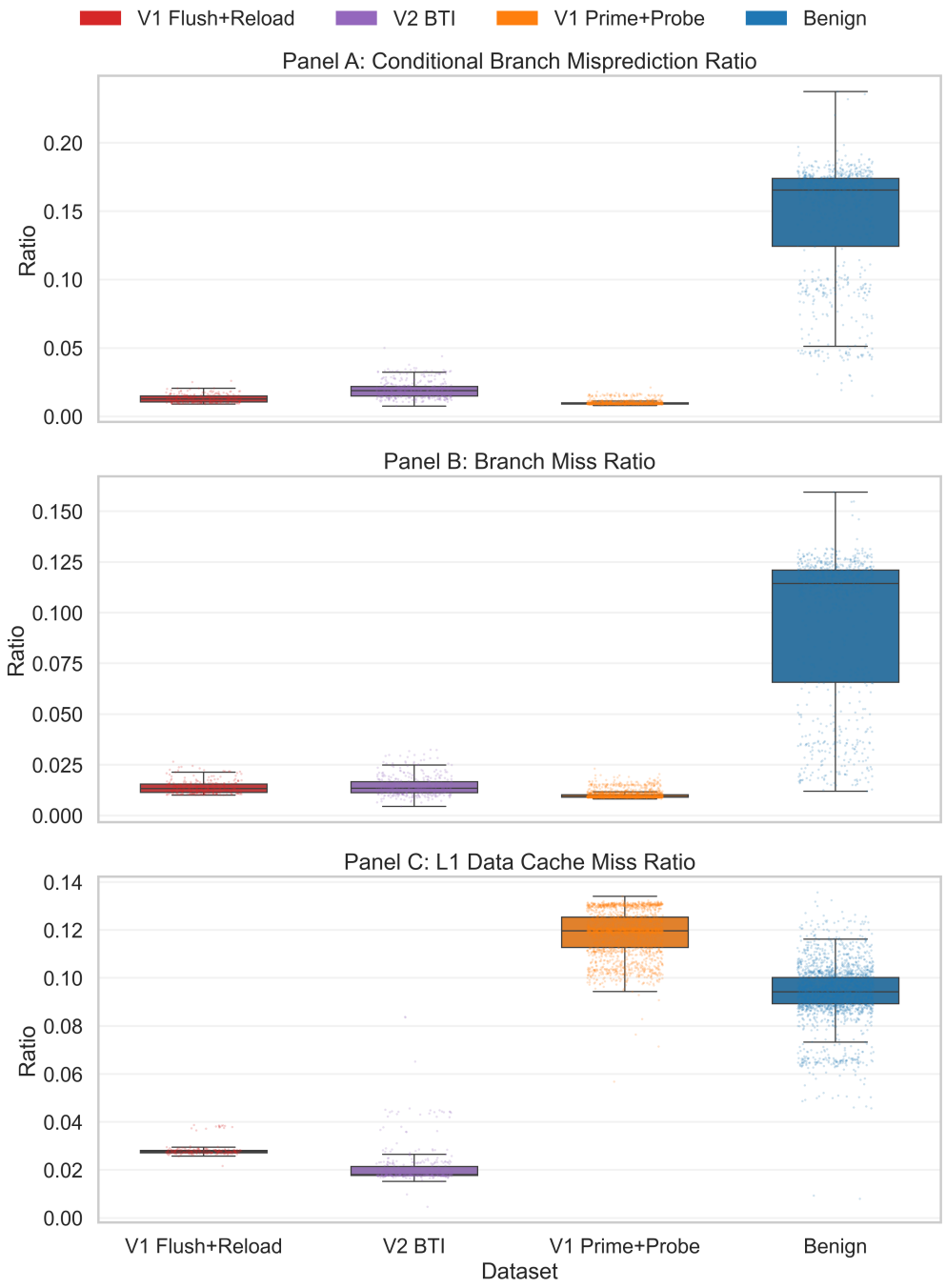


Figure 4.1: Baseline HPC fingerprints for attack variants on Intel Core i5-7200U.

As shown in Figure 4.1, Panels A and B isolate the impact of speculation triggers via the Conditional Branch Misprediction Ratio and Global Branch Miss Ratio. Whether utilizing a conditional bounds check bypass (`V1_fr`, `V1_pp`) or indirect branch target injection (`V2_fr`),

all three variants display remarkably similar fingerprints characterized by low-magnitude, tightly clustered ratios. Spectre V2 exhibits a slightly higher Conditional Branch Misprediction Ratio (Median: 0.018) compared to Spectre V1 variants (Median: 0.012), reflecting the additional overhead of indirect branch poisoning. However, this variation remains minor when compared to the elevated variance and median of the naturally noisy benign baseline (Median: 0.165). This consistency occurs because unthrottled attack loops successfully train underlying predictors to favor the speculative path, resulting in a stable, low-entropy signature.

In contrast, Panel C of Figure 4.1 highlights divergence across leakage channels. While `V1_fr` and `V1_pp` share a similar speculation trigger, Prime+Probe (`V1_pp`) infers data by measuring set-based disturbance across the cache hierarchy. This necessitates constructing eviction sets and thrashing the L1 cache, pushing its median L1 Data Cache Miss Ratio to 0.119. Meanwhile, targeted Flush+Reload implementations create a low-impact footprint (Median: 0.027), placing Prime+Probe's severe cache contention visibly above both Flush+Reload variants and the benign baseline (Median: 0.094).

While baseline microarchitectural footprints provide a recognizable foundational signature, treating these signatures as universal constants introduces critical classification errors. Initial telemetry assumes uniform hardware translation; however, physical counters are bound by vendor design philosophies, meaning the same logical attack induces divergent counter behaviors across architectures.

4.1.1 Baseline Cross-Architecture Timing Thresholds

Before analyzing the resulting HPC distributions, we first illustrate the fundamental side-channel leakage mechanism. Figure 4.2 presents the Flush+Reload timing curves for a leaked character ('T') across the Intel, AMD, and ARM testbeds. Across all platforms, the attack relies on exploiting a stark timing differential: a distinct low-cycle peak represents a rapid

cache hit for the successfully leaked character, while the clustered, higher-cycle noise represents slower main memory accesses for incorrect predictions. This specific timing gap visually demonstrates how the Flush+Reload channel deduces which character was speculatively accessed by Spectre. While this relative hit-to-miss mechanism remains constant, the absolute timing thresholds (measured in raw cycles or nanoseconds) differ significantly between vendors due to distinct cache access latencies and memory controller implementations. Consequently, an attacker must precisely calibrate these baseline timing thresholds for each specific architecture to ensure a successful exploit.

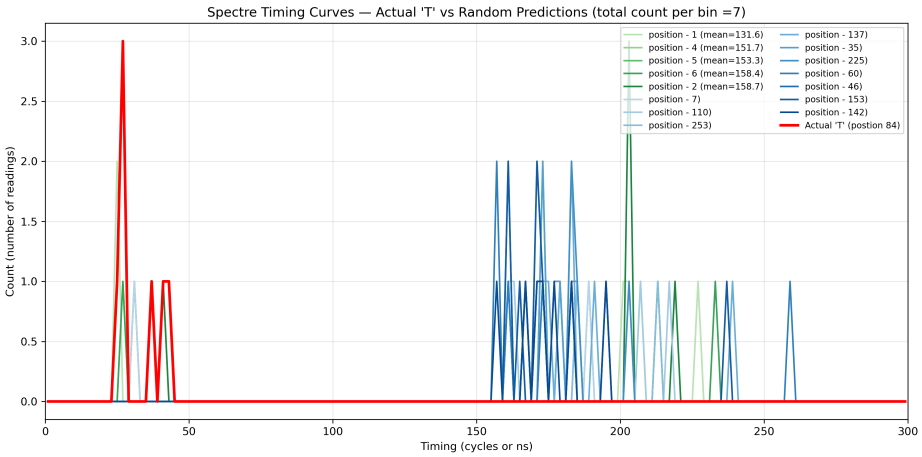
4.1.2 Zooming into Cache Miss Distributions: Intel 1 vs. ARM Cortex-A76

To expand our understanding of how microarchitectural boundaries manifest in telemetry distributions, we zoom into the cache behavior of two specific devices from our hardware matrix: the Intel Core i7-3537U (Intel 1) and the ARM Cortex-A76. Examining their raw distributional histograms exposes the fine-grained mechanics governing counter behavior.

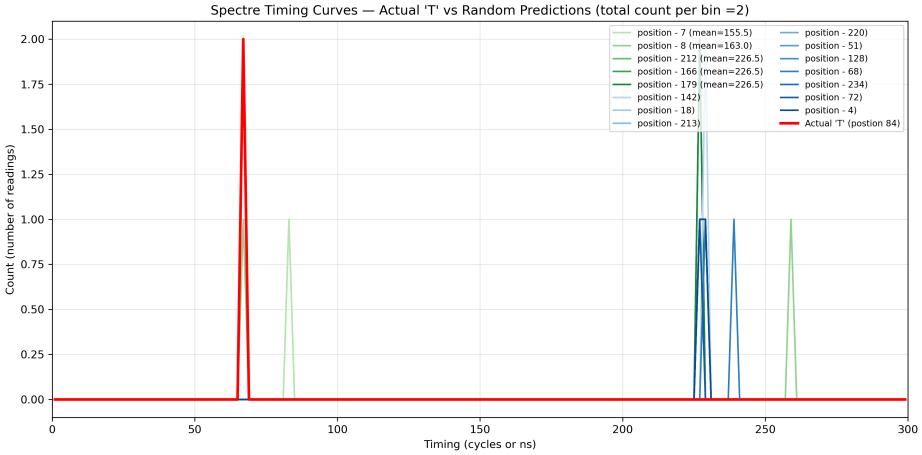
As captured in the cache miss rate histograms of Figure 4.3, the two architectures exhibit fundamentally inverted macroscopic profiles during attack execution. On Intel 1, the cache miss rate distribution undergoes a severe rightward shift during attacks due to inclusive Last-Level Cache (LLC) back-invalidation and speculative replay paths. Conversely, the ARM Cortex-A76 displays a concentrated attack cluster at lower ratios because its non-inclusive cache hierarchy and private-cache residency absorb speculative traffic without triggering extensive DRAM lookups.

4.1.3 Zooming into Branch Dynamics

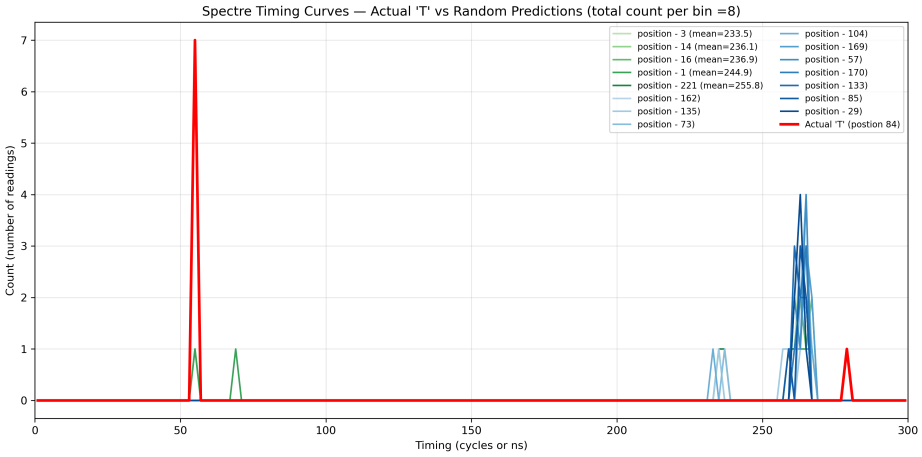
Extending our fine-grained analysis beyond memory mechanics, we examine how control-flow divergence differs across these architectures.



(a) Intel Testbed

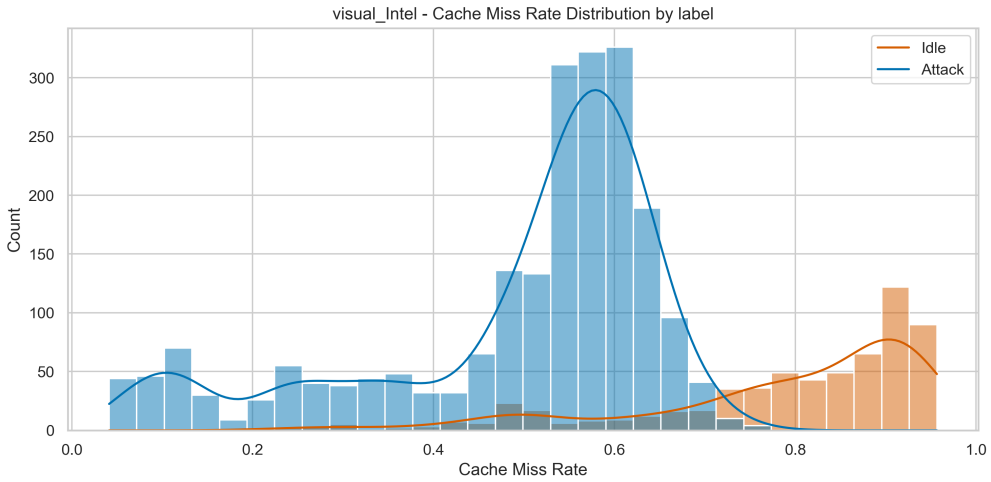


(b) AMD Testbed

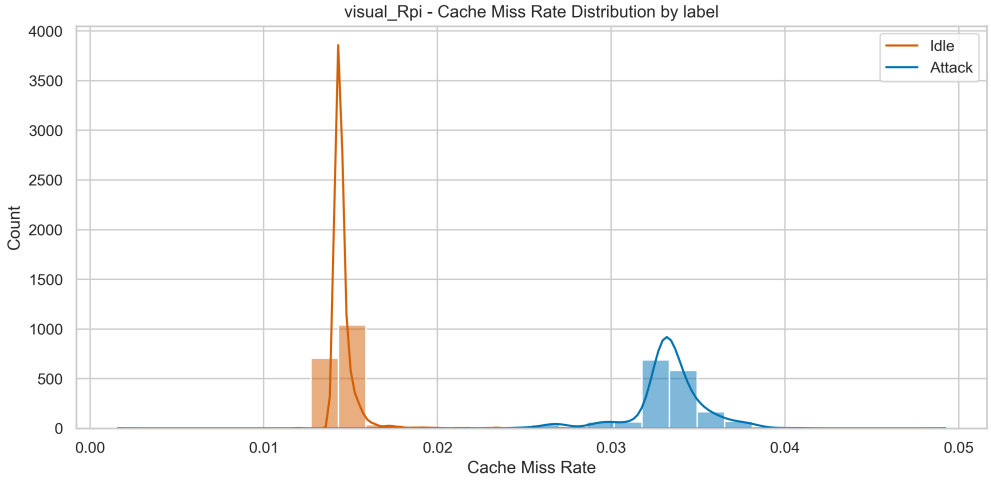


(c) ARM Cortex-A76

Figure 4.2: Baseline Flush+Reload cache hit and miss timing distributions across test architectures.

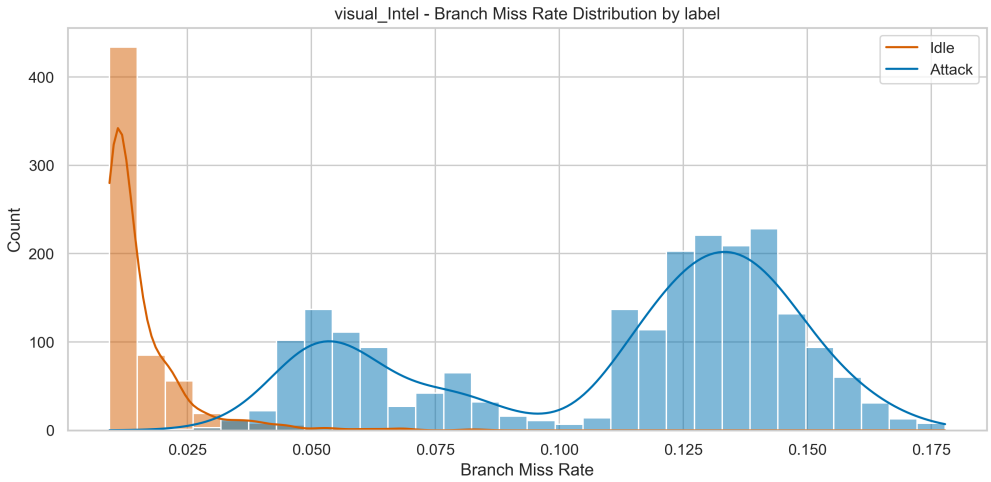


(a) Intel 1 Cache Miss Rate

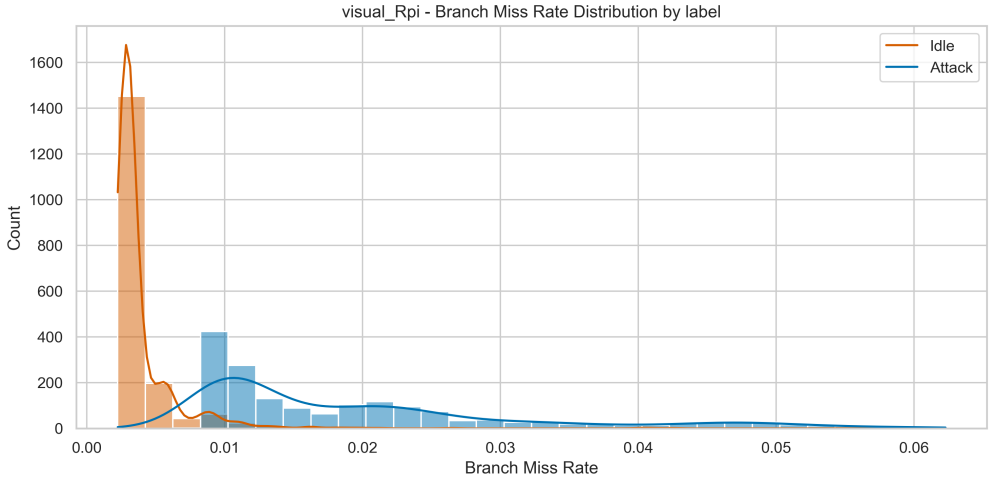


(b) ARM Cortex-A76 Cache Miss Rate

Figure 4.3: Cache miss rate distributions comparing Intel 1 and ARM.



(a) Intel 1 Branch Miss Rate



(b) ARM Cortex-A76 Branch Miss Rate

Figure 4.4: Branch miss rate distributions comparing Intel 1 and ARM.

Zooming into the branch dynamics via Figure 4.4 illustrates how control-flow divergence is recorded across vendors. Intel 1 exhibits a wide bimodal separation where the attack forms a tightly constrained lower-miss-rate cluster separated from benign stochastic noise. In contrast, the ARM platform compresses the branch miss spectrum, reflecting the distinct speculative path depth and predictor feedback mechanisms of the Cortex-A76 core.

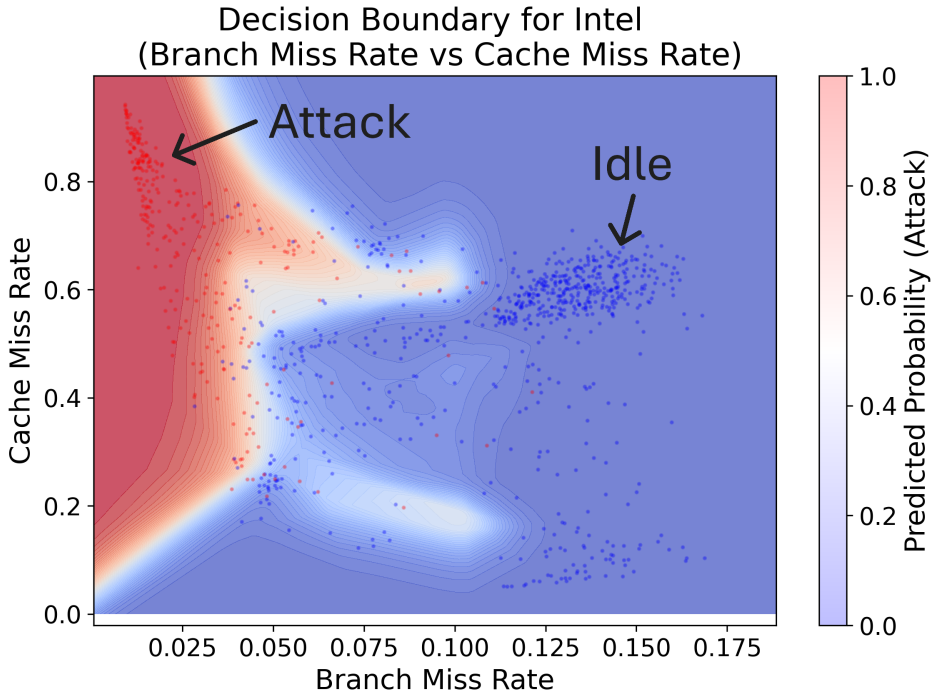
4.1.4 Zooming into Decision Boundaries

Finally, we examine how these hardware-specific distributions shape the classification spaces between benign and malicious states.

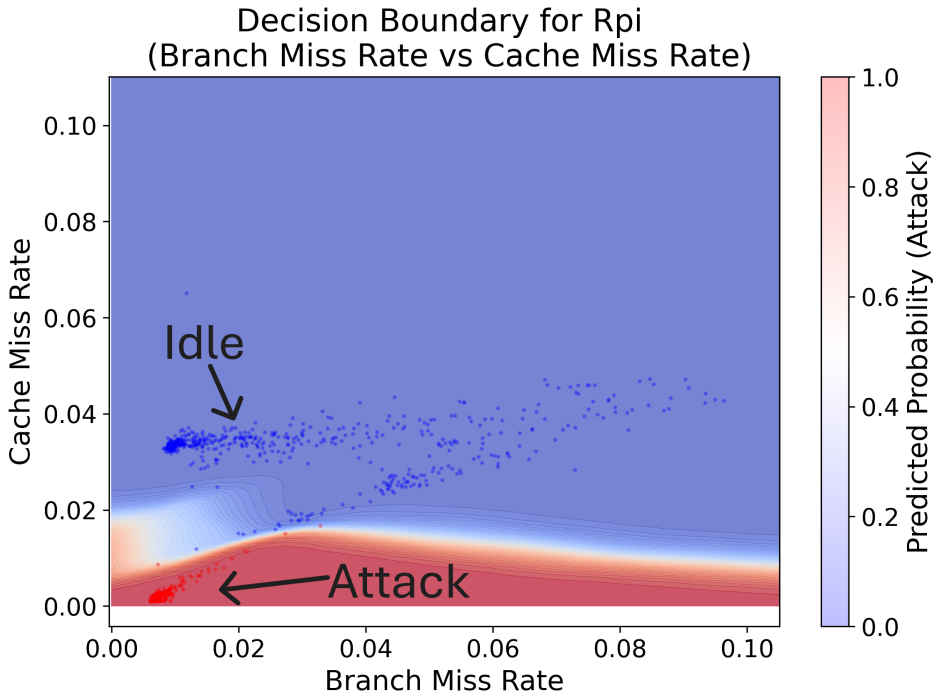
These distributional divergences directly dictate the geometric topology of the classification spaces. As visualized in the zoomed decision boundaries of Figure 4.5, a classifier trained on Intel 1 maps a complex, multi-regime contour bounded by high cache-miss thresholds, whereas the ARM boundary relies on a sharp, low-cache separation curve. This visual evidence confirms that static decision boundaries cannot be directly interchanged between Intel 1 and ARM architectures without incurring substantial classification errors.

4.2 Semantic Counter Discrepancies

Failed cache exploits are frequently misattributed to insufficient timer resolution. However, signature absence often indicates that the attack fails to induce microarchitectural state changes, exposing an inherent state of architectural resistance. A major factor in cross-platform signature variance is cache topology; for example, Intel x86 processors predominantly employ inclusive caching, whereas recent ARM processors utilize non-inclusive or exclusive caching policies. Inclusive caching guarantees that any memory address present in a higher-level cache tier is also retained in lower-level tiers, fundamentally altering how eviction events propagate through the memory hierarchy.



(a) Intel 1 Decision Boundary



(b) ARM Cortex-A76 Decision Boundary

Figure 4.5: Zoomed two-metric classification decision boundaries.

Furthermore, semantic discrepancies in performance counter definitions across Instruction Set Architectures (ISAs) severely limit the portability of static detection models. Vulnerable speculative execution capabilities span microprocessors from Intel, AMD, and ARM. Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels. However, the performance counter telemetry capturing these channels differs strictly by architecture.

On Intel x86 systems, the generic `cache-misses` event reported by Linux `perf` maps to `LONGEST_LAT_CACHE.MISS`, which counts references missing the last-level cache. Conversely, ARM Cortex processors utilize distinct event architectures for tracking last-level cache misses, relying on counters such as `LL_CACHE_MISS`. Moreover, structural differences across platforms further complicate telemetry interpretation; for instance, the Last-Level Cache (LLC) exposed on our AMD evaluation devices corresponds to the shared L2 cache rather than an L3 cache as found on other hardware profiles. A complete listing of all derived telemetry metrics and their architectural mappings is detailed in Appendix B.

Recognizing that cache topologies, hardware tiering, and counter definitions dictate telemetry availability, we can contextualize the limits of cross-platform detection. Because a semantic shift in an event fundamentally alters the underlying physical phenomenon recorded by the hardware, an attack signature optimized on one ISA cannot be directly ported to another. Generic thresholds calibrated on Intel-like inclusive architectures inevitably fail to map accurately to ARM's exclusive caching, AMD's L2-as-LLC hierarchy, and differing Performance Monitoring Units (PMUs). Consequently, semantic discrepancies serve as the primary catalyst for variance in cross-device observability, underscoring that portable transient-execution detection requires architecture-aware feature selection and normalization.

4.3 Cross-Device Observability

A core fallacy in portable hardware security is the assumption that identical attack software generates uniform telemetry across different processors. To quantify this cross-device variance, the unthrottled Spectre V1 Flush+Reload implementation is evaluated against a benign idle baseline across primary hardware profiles, including the Intel Core i7-3537U, ARM Cortex-A76, and AMD Jaguar. Abstracting telemetry into normalized ratios—such as the Cache Miss Ratio and Branch Miss Ratio—mitigates baseline bias from variations in clock frequencies and pipeline widths.

As illustrated in Figure 4.6, Panel A shows that the attack consistently forces a Cache Miss Ratio near 1.0 across all devices due to aggressive cache flushing inherent to the Flush+Reload channel. However, the benign baseline reveals pronounced architectural divergence. On the Intel Core i7 platform, the benign cache miss ratio spans a wide variance envelope, creating a measurable gap between normal execution and the attack. Conversely, the AMD Jaguar exhibits a naturally high and tightly clustered benign Cache Miss Ratio (Median: 0.92, IQR: 0.90–0.93). Compared to Intel's benign median of 0.69, this native architectural behavior drastically compresses the separability margin between benign and malicious execution, demonstrating the risk of cross-ISA false positives.

Panel B of Figure 4.6 demonstrates similar divergence in the Branch Predictor. Attack execution creates a highly predictable control flow, resulting in a low, tightly clustered Branch Miss Ratio across platforms. However, benign system noise varies considerably; the ARM Cortex-A76 maintains a low benign branch miss ratio, whereas the AMD Jaguar exhibits marked variance with an interquartile range extending far above other platforms. This baseline data confirms that class separation boundaries are strictly hardware-dependent.

Having mapped the variance envelope across spatial and architectural dimensions, it is evident that hardware heterogeneity distorts static attack signatures. However, Spectre attacks

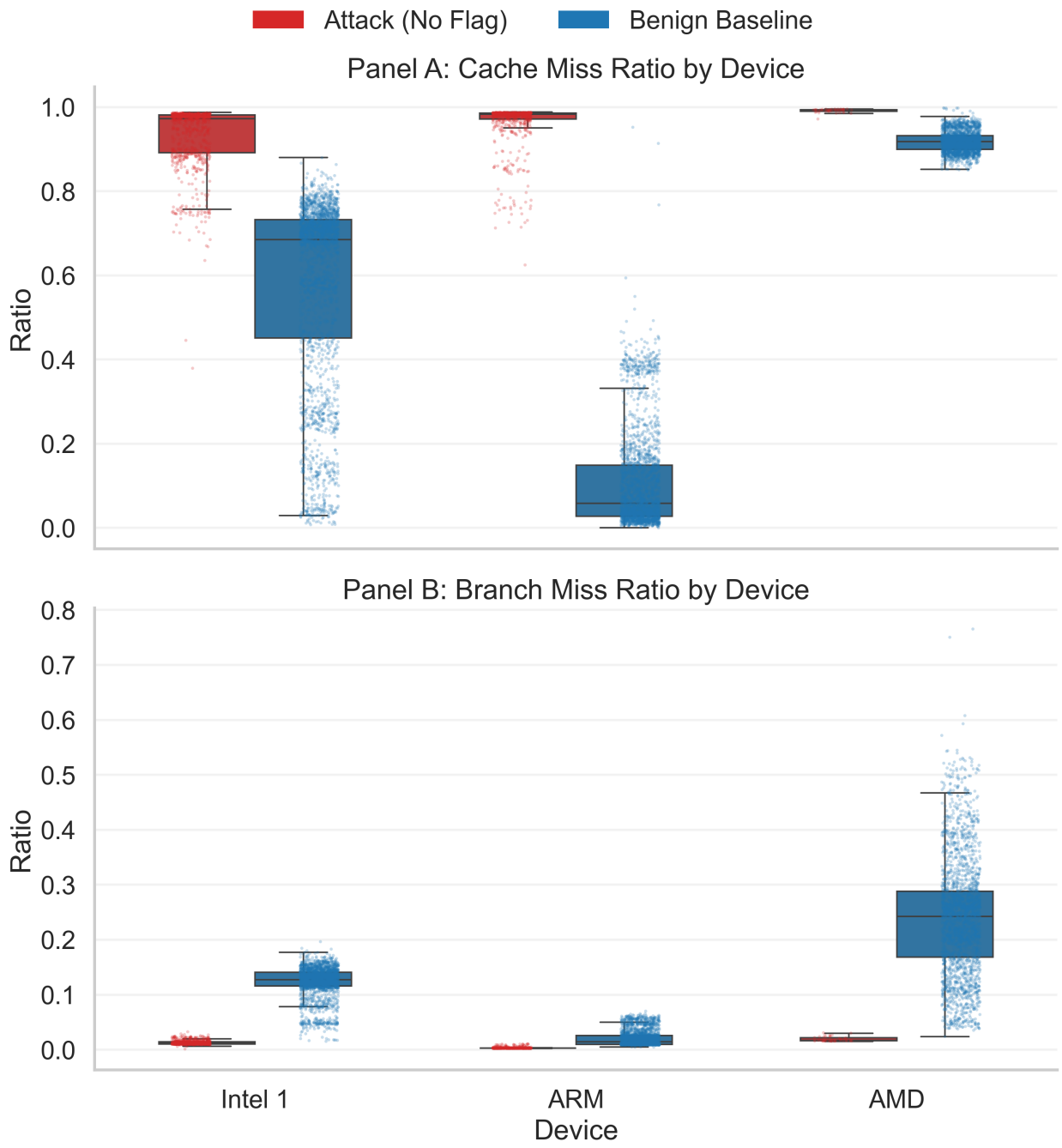


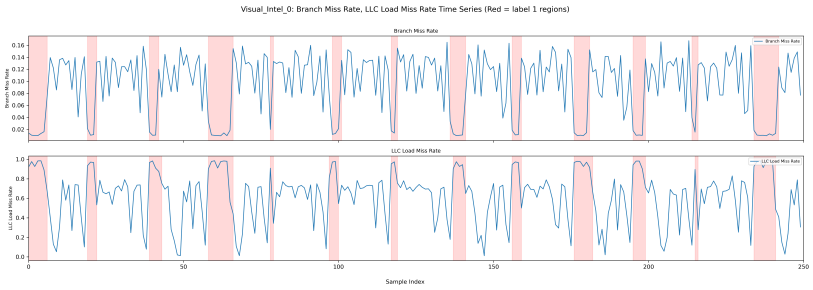
Figure 4.6: Baseline HPC telemetry comparing unthrottled Spectre V1 against benign idle execution.

are also acutely sensitive to execution rate. To fully characterize the operational variance envelope, analysis must shift from static execution footprints to evaluate how signatures behave dynamically over time under transient system noise and adversarial manipulation.

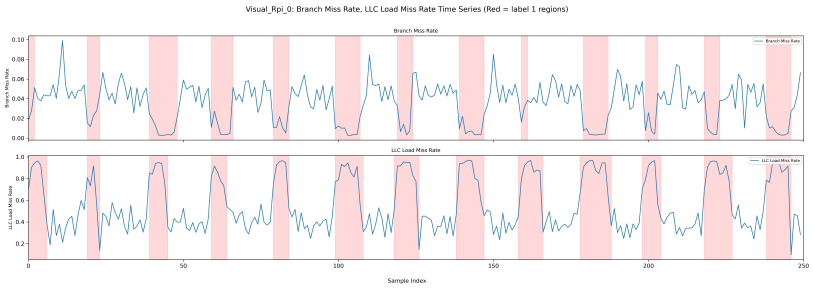
4.4 Temporal Dynamics

To complement our batch-level findings, this section examines how Spectre activity unfolds dynamically over time. This temporal perspective is essential for runtime detection systems and directly motivates leveraging sequential changes across consecutive sample windows for classification.

The analysis focuses on two representative ratio metrics: the Branch Miss Rate, which indicates speculative control-flow divergence, and the LLC Load Miss Rate, which captures long-latency speculative accesses to DRAM-backed probe arrays.



(a) Intel 1 Time Series



(b) ARM Cortex-A76 Time Series

Figure 4.7: Time-series of Branch Miss Rate and LLC Load Miss Rate on Intel (Top) and ARM (Bottom). Red bands mark Spectre-labeled regions.

Figure 4.7 contrasts the temporal behavior of these ratios during throttled Spectre executions. On Intel, attack regions appear as sharp transitions where the Branch Miss Rate drops toward zero and the LLC Load Miss Rate approaches saturation, displaying abrupt transitions at burst boundaries. On ARM, the corresponding attack windows are slightly wider and exhibit smoother transitions: the Branch Miss Rate decreases across labeled regions, while the LLC Load Miss Rate forms rounded peaks that remain distinct from the surrounding baseline. These architecture-shaped temporal dynamics provide discriminative signatures that can be leveraged by sequence-aware runtime classifiers operating on consecutive sampling windows.

4.5 Adversarial Pacing and Environment Noise

A sophisticated attacker can trade execution throughput for microarchitectural stealth via traffic shaping, modulating the frequency and duration of speculative bursts to distort resulting HPC distributions. This throughput-visibility trade-off relies on manipulating temporal dynamics to blend malicious execution into legitimate background patterns.

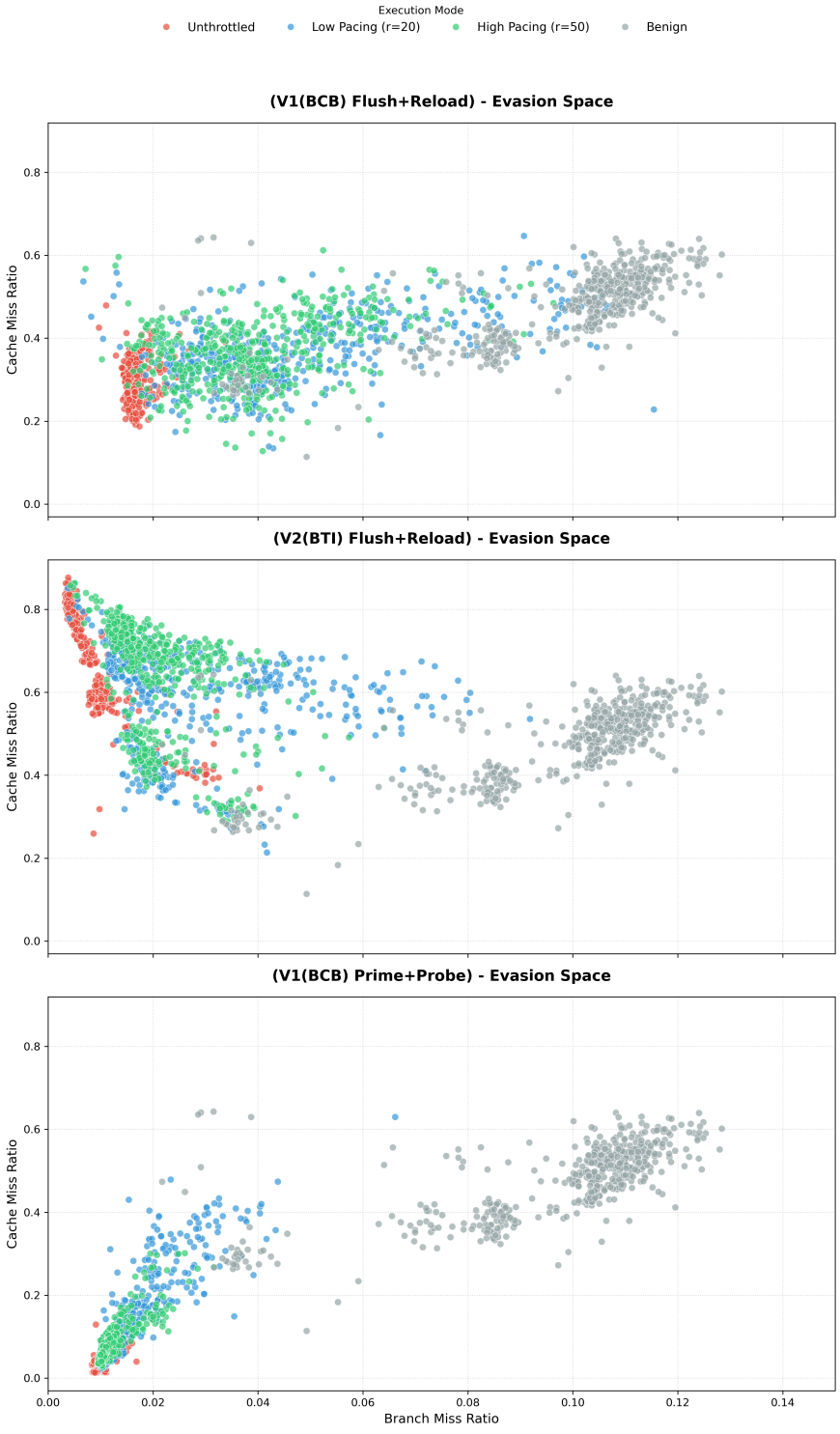


Figure 4.8: Evasion space trajectories for Spectre V1 and V2 attack variants.

As captured in Figure 4.8, empirical evaluation demonstrates distinct temporal evasion behaviors. `V1(BCB)` Flush+Reload exhibits horizontal entropy diffusion (top panel). As pacing increases, the signature maintains a high Cache Miss Ratio but slides horizontally, indicating that temporal shaping reduces cache eviction frequency while increasing microarchitectural entropy within the branch predictor, pushing signatures into high-variance benign clusters.

In contrast, `V2(BTI)` Flush+Reload demonstrates a precision retention strategy (middle panel), remaining pinned in the high-cache/low-branch-miss region even under heavy pacing ($r = 50$). This indicates that Branch Target Injection maintains high branch prediction accuracy during high-frequency flushes. The most elusive primitive, `V1(BCB)` Prime+Probe (bottom panel), follows a diagonal signal collapse trajectory where cache intensity and branching signatures degrade proportionally, rendering high-pacing executions statistically difficult to distinguish from low-intensity benign samples.

Beyond adversarial evasion, practical deployment occurs within multi-tenant environments where background operations compete for shared hardware resources. Subjecting the `V1` Prime+Probe attack and a benign baseline to a multi-dimensional interference matrix reveals a clear operational divergence between metrics that degrade gracefully and those that experience severe signal distortion.

As detailed in Figure 4.9, Panel A shows that the L1 Data Cache Miss Ratio suffers a severe loss of class separability under resource-intensive workloads (CPU and memory-bound tasks). This overlap occurs due to frequency-domain interference between benign cache thrashing and malicious eviction, where execution contention throttles attacker throughput. Under memory pressure, the benign L1 cache distribution expands significantly (Median: 0.094, Max: 0.134), substantially overlapping with the Prime+Probe median signature (Median: 0.108) and rendering static L1 thresholds ineffective.

Conversely, Panel B of Figure 4.9 demonstrates that speculative branching metrics degrade

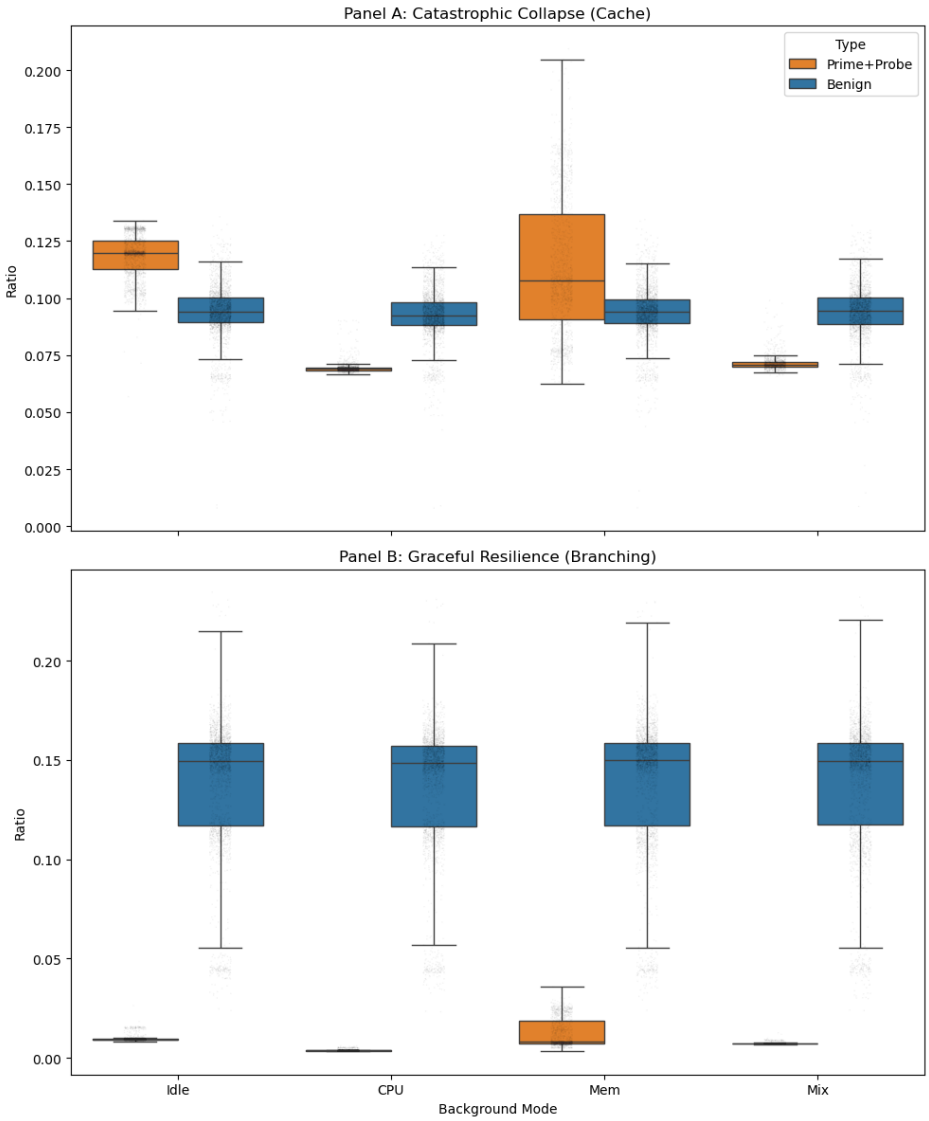


Figure 4.9: Telemetry degradation under multi-dimensional system noise.

gracefully. The Conditional Branch Misprediction Ratio maintains a distinct margin of separability from the noisy benign baseline (Median: 0.150) even under heavy memory contention (attack Median: 0.008, Max: 0.036). The dedicated hardware mechanisms of the Branch Prediction Unit remain sensitive to repetitive, tight-loop Spectre logic, preserving a distinct low-entropy branching signature.

Chapter 5

Modeling Speculative Anomalies: From Static to Adaptive

As established in the preceding analysis, the microarchitectural telemetry of Spectre attacks, such as Spectre [17] is fundamentally non-stationary. Recognizing that static thresholds are easily bypassed by adversarial traffic shaping and cross-architecture variance, this chapter details the transition from rigid heuristics to adaptive, machine learning-driven detection. We begin by formalizing the mathematical necessity for domain adaptation. Following this, we evaluate lightweight offline classifiers utilizing normalized feature ratios to establish baseline performance in constrained environments. Finally, we explore advanced temporal architectures, including Long Short-Term Memory (LSTM) networks [13] and Domain-Adversarial Neural Networks (DANN) [9], culminating in the introduction of a novel hybrid DA-LSTM and XGBoost [3] architecture designed for robust, cross-platform runtime classification.

5.1 The Case for Adaptation

The empirical characterizations detailed throughout Chapter 4 demonstrate that the microarchitectural footprint of a Spectre attack is not a fixed, deterministic signature. To formalize the necessity of advanced modeling, let $X \in \mathbb{R}^d$ represent the d-dimensional HPC telemetry feature vector and $Y \in \{0, 1\}$ denote the system state (benign or under attack). Under ideal, isolated conditions, a static thresholding system seeks to learn a rigid mapping function $f : X \to Y$ based on the assumption that the joint probability distribution $P(X, Y)$ is stationary [39].

However, our multi-dimensional evaluation proves that the true telemetry distribution is highly parameterized by the architectural domain $\mathcal{D}$, the environmental noise interference ϵ, and the adversarial pacing rate λ. Consequently, the variance envelope must be mathematically defined as a non-stationary, domain-dependent distribution $P(X, Y \mid \mathcal{D}, \epsilon, \lambda)$. Because the target distribution drifts continuously as these parameters shift in a live environment, any static decision boundary optimized for a specific, isolated configuration $(\mathcal{D}_i, \epsilon_i, \lambda_i)$ will inevitably suffer from severe covariate shift [36, 32] when deployed in a generalized, real-world setting $(\mathcal{D}_j, \epsilon_j, \lambda_j)$.

This mathematical reality invalidates the efficacy of rigid, heuristic thresholding. It necessitates a transition toward statistical modeling that can first capture this complex baseline variance and subsequently adapt to it in real-time, forming the direct motivation for the lightweight classifiers and advanced temporal architectures detailed in this chapter.

5.2 Lightweight Offline Classifiers

To establish a baseline for detection efficacy before introducing complex temporal architectures, we first evaluate the performance of standard, lightweight offline classifiers. A critical prerequisite for these models is the transformation of raw hardware telemetry into normalized

feature representations capable of mitigating baseline environmental noise.

5.2.1 Feature Representation and Normalization

Raw hardware performance counters exhibit massive variance depending on the specific microarchitecture and background workload. To address this, the input data is constructed using derived feature ratios rather than raw counter values [21]. These derived ratios normalize the raw telemetry, effectively eliminating variations caused by device-specific differences or background programs. This normalization focuses the model on the relative inefficiencies characteristic of speculative exploitation, enabling accurate binary classification while reducing noise [21].

The feature set utilizes 20 raw hardware events processed into 12 normalized ratios, divided into two categories [21]:

- **Cache Ratios:** Cache Miss Rate, L1 Data Cache Load Miss Rate, L2 Demand Data Read Hit Ratio, Offcore All Data Reads LLC Miss Rate, and Offcore Demand Data Reads LLC Miss Rate.

- **Branch Ratios:** Branch Miss Rate, Executed Branch Misprediction Rate, Retired Branch Misprediction Rate, Executed to Conditional Misprediction Rate, Retired Near Taken to Conditional Misprediction Rate, Retired to Executed Branch Rate, and Retired to Executed Branch Misprediction Rate.

To visually demonstrate the necessity and effectiveness of these specific derived ratios, Figure 5.1 illustrates the telemetry distribution of a Spectre attack compared to an idle baseline. It is important to note that all baseline experiments and data collection for these lightweight offline models were conducted exclusively on the Intel 1 hardware architecture.

As shown in the scatter plots, the microarchitectural footprint of the exact same attack

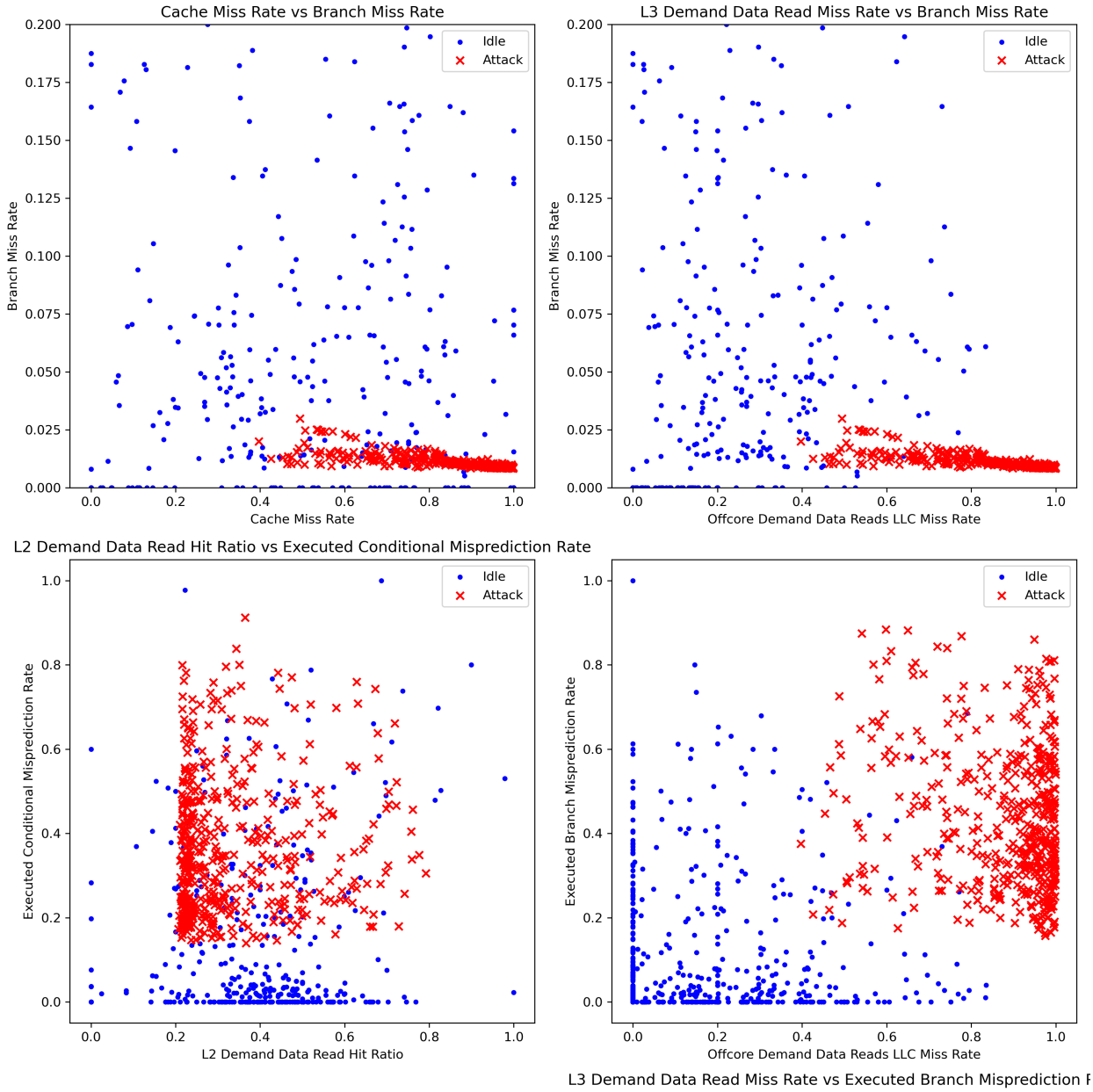


Figure 5.1: Scatter plots demonstrating the discriminative power of different normalized feature ratios during a Spectre attack (red crosses) versus an idle baseline (blue dots). Data was collected exclusively on the Intel 1 architecture.

differs significantly depending on the feature dimensions analyzed. While broader metric combinations, such as the overall Cache Miss Rate versus Branch Miss Rate (top-left), exhibit overlapping clusters with poorly defined decision boundaries, utilizing highly specific derived ratios yields a much clearer classification space. For instance, mapping the L3 Demand Data Read Miss Rate against the Branch Miss Rate (top-right), or isolating the L2 Demand Data Read Hit Ratio (bottom-left), provides a stark visual separation between malicious and benign execution. This variance in class separability across different feature projections underscores the critical importance of selecting highly discriminative ratios prior to training the baseline models.

5.2.2 Temporal Framing

While normalized ratios provide a stable spatial signature, Spectre attacks manifest over time. To capture temporal correlations without the immediate overhead of recurrent neural architectures, we implement a sliding window approach. The initial input data is formatted as 12 values per reading, with each reading representing a 5-millisecond window.

To improve temporal correlation capture, consecutive readings are grouped into frames of 5 readings, covering a total duration of 25 milliseconds. Consequently, each frame consists of 60 features (12 ratios $\times$ 5 readings). This structured input allows offline models to learn both intra-reading relationships and inter-reading temporal patterns, significantly enhancing detection accuracy [21].

5.2.3 Baseline Evaluation and the Limits of Static Models

Using this 60-feature framed input, several foundational models were evaluated: Logistic Regression, Elastic Net [45], Support Vector Machines (SVM) [5], Gaussian Processes [34], Radial Basis Function Neural Networks (RBFNN) [2], and a Multi-Layer Perceptron (MLP) [35].

As demonstrated in our prior evaluations, summarized in Table 5.1, the MLP achieved the highest baseline detection accuracy of 99.87% under isolated, idle conditions, requiring only 35.78 seconds of training time and 17.23 MB of memory [21].

Table 5.1: Model Performance, Resource Usage, and Error Rates [21]

Model	**Accuracy**	**Time (s)**	**Memory (MB)**	**FP + FN**
Logistic Regression	97.93%	0.206	11.57	415
Elastic Net	98.06%	1.325	1.52	388
SVM with RBF	99.57%	39.731	9.45	86
Gaussian Process	99.33%	225.749	345.33	135
RBFNN	99.06%	30.096	20.77	188
MLP (Single Input)	99.87%	35.780	17.23	27

However, when evaluating the MLP under shifting background workloads, the limitations of static classification become apparent. Table 5.2 details the model's performance degradation when transitioning from an idle state to intensive workloads. When tested directly on a combined CPU and memory load without adaptation (Configuration 1), the model's accuracy dropped to 89.75%, generating 205 combined false positives and false negatives [21].

Table 5.2: Detection Accuracy and Error Counts Across Modes [21]

Background Mode	**% Accuracy (Config 1 / 2)**	**FP + FN (Config 1 / 2)**
Idle Mode	99.95 / 99.47	1 / 10
High CPU Utilization	90.15 / 99.26	197 / 14
High Memory Utilization	71.55 / 99.79	569 / 4
Combined Load	89.75 / 99.42	205 / 11

To demonstrate the necessity of adaptation, a secondary evaluation (Configuration 2) was performed where the model was retrained using just 5% of the new attack data. As seen in Table 5.2, this minimal retraining recovered the detection accuracy to 99.42% under the combined load, reducing the error count by 94% (from 205 to 11 errors) [21]. While this proves that adaptive boundaries can successfully identify transient attacks amid extreme system noise, relying on continuous manual retraining of an MLP is not feasible for real-time, cross-device deployment. This directly motivates the development of the next-generation

temporal and domain-adversarial models detailed in Section 5.3.

5.3 Next-Generation Temporal and Adaptive Models

Time-series data requires a model to track sequential patterns and respond to changing data shifts. Traditional machine learning architectures often assume inputs are independent and identically distributed, an assumption that fails when handling time-related dependencies or domain shifts. The particular challenges of time-series data have given rise to various sequence-aware architectures, such as recurrent neural networks, temporal autoencoders, and domain-aware neural models. This section features both classical and state-of-the-art models within this domain. (The complete matrix of structural hyperparameters and optimization settings utilized to train these temporal and adaptive architectures is detailed in Appendix A.)

5.3.1 Recurrent Neural Networks & Memory Methods

Recurrent Neural Networks

Time-series data exhibits time-based dependencies. In other words, the prediction at time t, denoted as y_t, is not only dependent on x_t, but also on previous inputs $x_{t-1}, x_{t-2}, \ldots$. Traditional neural networks do not explicitly define temporal relationships between sequential elements. As a result, Recurrent Neural Networks (RNNs) were introduced by Elman [7] to address this limitation and capture the flow of information over time.

RNNs process sequential inputs step-by-step by maintaining an internal hidden state h_t, which acts as a memory of past inputs. At each time step t, the hidden state is updated based on the current input x_t and the previous hidden state h_{t-1}:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b_h) \tag{5.1}$$

Here, W_x and W_h are shared weight matrices across time steps, and b_h is the bias vector. The h_t update recursively compresses the entire input sequence history up to time t. Because the matrix W_h is shared across all time steps, the model detects consistent temporal patterns regardless of where they appear in the sequence. The hidden state h_t can then be mapped to an output prediction:

$$\hat{y}_t = g(W_y h_t + b_y) \tag{5.2}$$

Despite their design for time-series data, standard RNNs struggle to learn long-range patterns due to the vanishing gradient problem. During backpropagation, the loss gradient is computed through time using the hidden states h_t, which depend directly on the weight matrices W_x and W_h. If the elements of these weight matrices are small, the gradients decay exponentially toward zero as they travel backward through long sequences. Consequently, the network fails to update its parameters efficiently for early time steps and "forgets" long-term context.

Long Short-Term Memory (LSTM) Architecture

The Long Short-Term Memory (LSTM) model was introduced by Hochreiter and Schmidhuber [14]. It builds upon the recurrent neural network architecture by including a cell state c_t that updates additively rather than via repeated matrix multiplications. This additive update allows gradients to flow across many time steps largely unattenuated. LSTMs also

implement gating mechanisms to control what information is written to, retained in, or read from the model's memory.

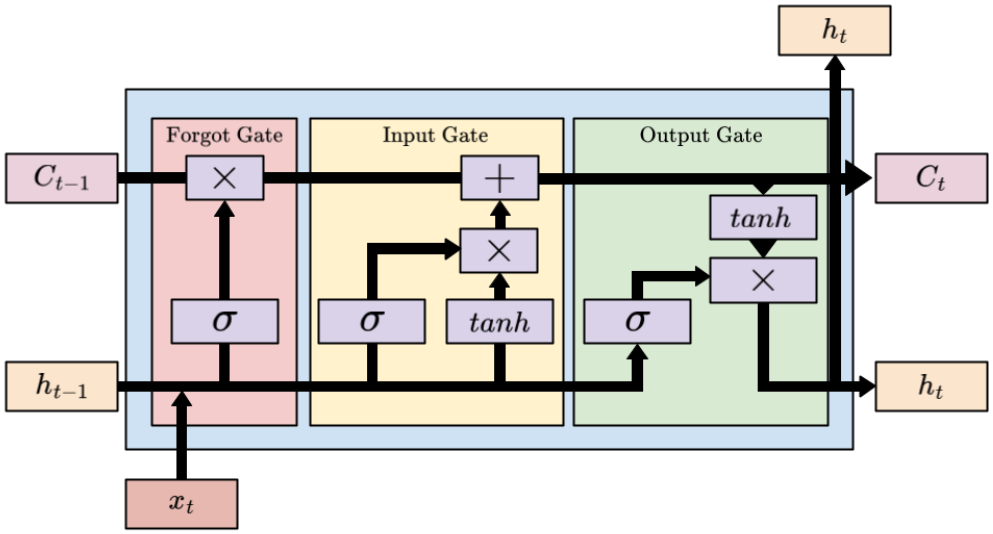


Figure 5.2: Diagram of an LSTM memory cell. The cell receives the previous cell state C_{t-1}, the previous hidden state h_{t-1}, and the current input x_t. It outputs the new cell state C_t and hidden state h_t, which are passed to the subsequent time step.

At each time step, the forget gate f_t, the input gate i_t, and the output gate o_t are computed from the current input x_t and the previous hidden state h_{t-1}. These values are passed through a sigmoid activation function σ so that their outputs strictly lie between 0 and 1, effectively acting as soft switches:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{5.3}$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{5.4}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{5.5}$$

The forget gate determines how much of the prior memory to retain, while the input gate decides how much new information to add, resulting in the following cell state update:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5.6}$$

where $\odot$ denotes element-wise multiplication. Because this update is additive, the network can preserve information over long time spans without experiencing exponential vanishing gradients.

The variable $\tilde{c}_t$ referenced in Equation 5.6 represents the candidate cell state, calculated as follows:

$$\tilde{c}t = \tanh(W_c x_t + U_c ht - 1 + b_c) \tag{5.7}$$

Its values range between -1 and 1, representing the new candidate information to be added to the memory prior to gating.

Finally, the output gate controls how much of the internal memory is exposed to the hidden state, which is subsequently used for predictions at the current step:

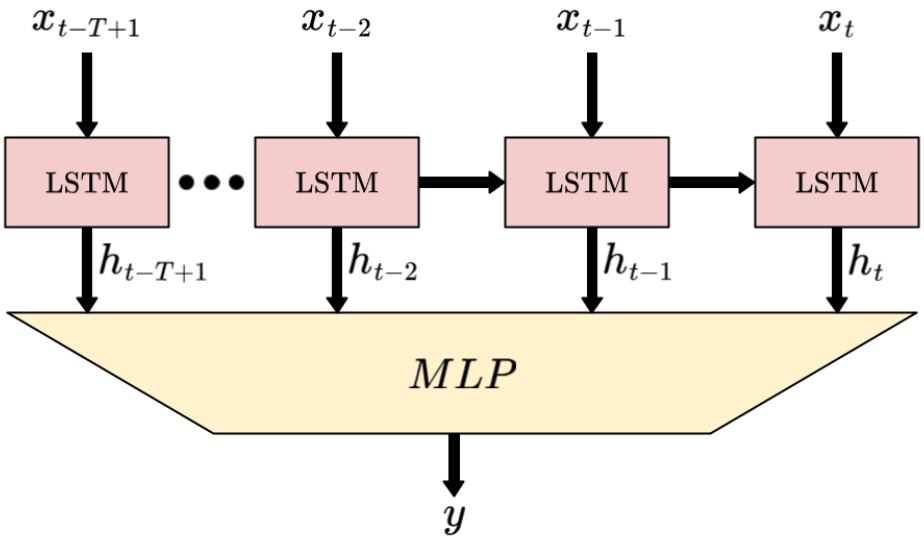


Figure 5.3: Diagram of a unidirectional LSTM model with a window of size T. The hidden states produced by the standard LSTM blocks are utilized as inputs to a traditional Multi-Layer Perceptron (MLP), which executes the final prediction.

$$h_t = o_t \odot \tanh(c_t) \tag{5.8}$$

This LSTM model can be directly applied to time-series data by windowing sequences of data, as surveyed by Gers *et al.* [10]. Multiple LSTM blocks are chained together, passing outputs sequentially into the next block (see Figure 5.3). The hidden states of the model produce a temporally aware feature representation of the data, which can then be passed into a feed-forward network to perform tasks such as classification. The LSTM's ability to follow patterns over time makes it a strong choice for supervised anomaly detection [27].

5.3.2 Autoencoder Methods

Autoencoders & Anomaly Detection Background

A standard autoencoder learns to compress an input vector x into a smaller latent representation z using an encoder, and then reconstructs the original input from that representation

using a decoder. For an autoencoder to learn meaningful features, the dimensionality of the latent space must be strictly smaller than the input space ($|z| < |x|$). This creates a bottleneck that forces the model to emphasize only the most relevant features necessary for reconstruction:

$$z = \text{Encoder}(x), \quad x' = \text{Decoder}(z) \tag{5.9}$$

An autoencoder is trained by minimizing a reconstruction loss, such as the mean squared error between the original input x and its reconstruction x':

$$\mathcal{L}(x, x') = |x - x'|^2 \tag{5.10}$$

Through this optimization process, the autoencoder learns a low-dimensional, compressed representation of the original data's distribution.

The architecture of an autoencoder gives it the intrinsic ability to function as an anomaly detector. If an autoencoder is trained only on "normal" data, it will learn to reconstruct those specific patterns with great accuracy, resulting in a low reconstruction error during inference. Conversely, when presented with outlier data that is poorly represented or entirely absent from the training set, the network struggles to reconstruct it accurately, producing a significantly higher reconstruction error.

By setting a predefined threshold τ on the reconstruction error during inference, we can classify whether an input data point x is normal or anomalous using the following decision

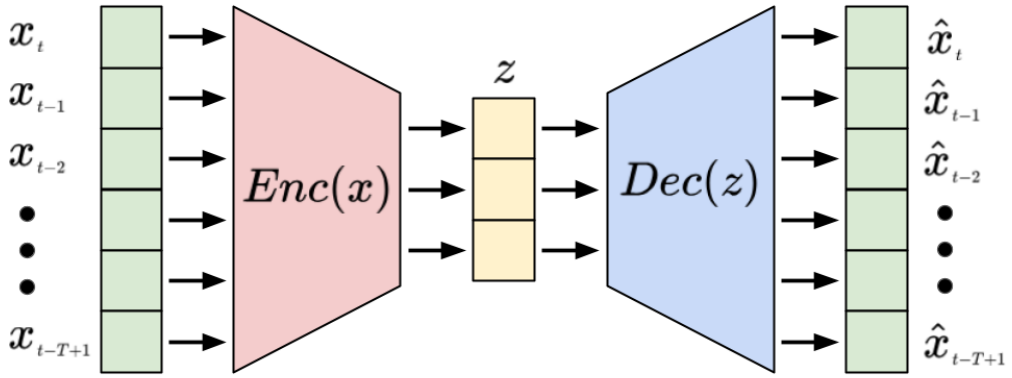


Figure 5.4: Architecture of a standard autoencoder, illustrating the data bottleneck between the encoding and decoding phases.

rule:

$$y(x) = \begin{cases} 0 \quad \text{(Normal)}, & \text{if } \mathcal{L}(x, x') \leq \tau \\ 1 \quad \text{(Anomalous)}, & \text{if } \mathcal{L}(x, x') > \tau \end{cases} \tag{5.11}$$

Variational Auto-encoder (VAE)

The bottleneck layer of a standard autoencoder is still susceptible to overfitting, which prevents it from properly generalizing the structure of the latent space. If the compression constraint is too weak, the model may simply memorize the input and reconstruct any type of data perfectly, severely degrading its performance as an anomaly detector.

The Variational Autoencoder (VAE) solves this issue by regularizing the latent space, forcing the model to learn a continuous probability distribution rather than mapping inputs to discrete deterministic points. Instead of directly outputting a static vector z, the encoder predicts the mean μ and log-variance $\log \sigma^2$ of a Gaussian distribution. The latent vector z

is then sampled using the reparameterization trick:

$$\mu, \log \sigma^2 = \text{Encoder}(x), \quad z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{5.12}$$

To balance reconstruction precision with latent space regularity, the VAE loss function combines a reconstruction loss with a distribution regularization term:

$$\mathcal{L} = \underbrace{|x - \hat{x}|^2}\text{reconstruction} + \beta \cdot \underbrace{DKL\left(\mathcal{N}(\mu, \sigma^2), |, \mathcal{N}(0, I)\right)}_{\text{regularization}} \tag{5.13}$$

The regularization term uses Kullback-Leibler (KL) divergence to penalize differences from a standard normal prior distribution $\mathcal{N}(0, I)$. This prevents the model from overfitting and encourages a smooth latent space, allowing the network to evaluate how probable an input is under normal behavior rather than relying solely on raw reconstruction error.

Temporal Convolutional (TCN) Autoencoder

Unlike standard autoencoders and VAEs, which take static vectors without a temporal dimension as input, the Temporal Convolutional (TCN) autoencoder operates on a time-ordered window. To preserve temporal ordering, its convolutional layers utilize **causal padding**, which forces each layer in the encoder and decoder to only consider current and past time steps.

For a given layer l, the hidden state representation $z_t^{(l)}$ at time step t is updated as:

$$z_t^{(l)} = \text{GELU}\left(\sum_{k=0}^{K-1} W_k z_{t-d\cdot k}^{(l-1)}\right) \tag{5.14}$$

where K is the kernel size, d is the dilation factor, and W_k represents the filter weights.

By stacking convolutional layers with exponentially increasing **dilation**, the model's **receptive field** grows rapidly, enabling even a shallow network to capture **long-range dependencies**. The TCN autoencoder utilizes the same overarching loss function as a standard VAE.

5.3.3 Tree-Based Methods

Decision Trees

Decision trees are binary tree structures that query specific feature characteristics and route each sample down a branch accordingly until reaching a leaf node that holds the final prediction. A decision tree constructs a prediction function by recursively partitioning the feature space into disjoint regions. Tree-based methods are extremely popular due to their efficiency. While decision trees are constructed greedily based on the data, most modern implementations incorporate ensemble concepts such as bagging or boosting.

The random forest algorithm implements the bagging method, which trains an ensemble of M trees independently in parallel on different subsets of the training data, then aggregates their individual predictions through an average or majority vote. Conversely, boosting methods construct trees sequentially rather than independently, allowing each additional tree to fit specifically to the residual errors of the ensemble constructed up to that point. Every new tree attempts to rectify the errors that the previous trees failed to capture, and the final prediction is made using a weighted average based on each tree's individual predictive ability.

XGBoost

XGBoost is a regularized implementation of tree boosting where the model is constructed additively across K boosting rounds. As a boosting method, at each round t, a new tree f_t is introduced to minimize the residual objective of the ensemble through round $t-1$.

XGBoost improves traditional boosting methods by adding a regularization term to the learning objective to prevent overfitting. This term penalizes the tree for both its number of leaves and the magnitude of the leaf weights:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{5.15}$$

Standard boosting methods approximate the loss at round t using a first-order gradient, but XGBoost expands this to a second-order Taylor approximation. This allows the model to account for not only the direction of the error but also the curvature (confidence) of that direction, yielding faster and more stable convergence than first-order methods.

Standard decision tree methods can grow indefinitely or until reaching a predefined depth limit. To further prevent overfitting, the XGBoost algorithm explicitly defines the quality of a potential node split using its gain:

$$\text{Gain} = \frac{1}{2}\left[(\text{score of left child}) + (\text{score of right child}) - (\text{score of parent})\right] - \gamma \tag{5.16}$$

The parameter γ represents the complexity cost of introducing an additional node to the tree. If the gain is negative or zero, it is insufficient to justify a split at that point. In this manner, XGBoost strikes a balance between making fine-grained predictions and avoiding unnecessary model complexity.

5.3.4 Domain-Adaptation Methods

Domain Descriptions

A domain is defined by a feature space $\mathcal{X}$, a label space $\mathcal{Y}$, and a joint probability distribution $P(X, Y)$ such that $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$. For the domain adaptation problem, there exists a **source domain** $\mathcal{DS} = \mathcal{X}, P\mathcal{S}(X, Y)$, for which labeled data $\mathcal{YS}$ is available, and a **target domain** $\mathcal{DT} = \mathcal{X}, P_{\mathcal{T}}(X, Y)$ for which there is little or no labeled data. Both domains span the same feature and label spaces, but their underlying distributions vary:

$$P_{\mathcal{S}}(X, Y) \neq P_{\mathcal{T}}(X, Y) \tag{5.17}$$

In the context of microarchitectural security, two different generations of Intel processors represent separate domains. The same hardware performance counters and metrics can be measured from both, but there are distinct variations in their statistical behavior owing to subtle underlying microarchitectural changes. Domain adaptation methods aim to learn a model using labeled source data and unlabeled target data that generalizes well to $\mathcal{D}_{\mathcal{T}}$, bridging this distributional mismatch.

Domain-Adversarial Neural Network (DANN)

A prevalent method for domain adaptation is the **Domain-Adversarial Neural Network (DANN)**, which focuses on learning a feature representation $\phi(X)$ such that the induced feature distributions are aligned while preserving enough information in $\phi(X)$ to accurately predict $\mathcal{Y}$ on the source domain:

$$P_{\mathcal{S}}(\phi(X)) \approx P_{\mathcal{T}}(\phi(X)) \tag{5.18}$$

The system attempts to find a solution where the model performs its primary classification task accurately while simultaneously being unable to distinguish which domain the underlying data originated from.

DANN achieves this goal through an **adversarial training** scheme consisting of three components. The **feature extractor** $G_f(x;\theta_f)$ maps the raw input to a learned representation $f = G_f(x)$. Two adversarial heads share this feature extractor: the **label predictor** $G_y(f;\theta_y)$, which is trained to predict the label Y from f, and the **domain classifier** $G_d(f;\theta_d)$, which is trained to predict whether f originated from the source or target domain.

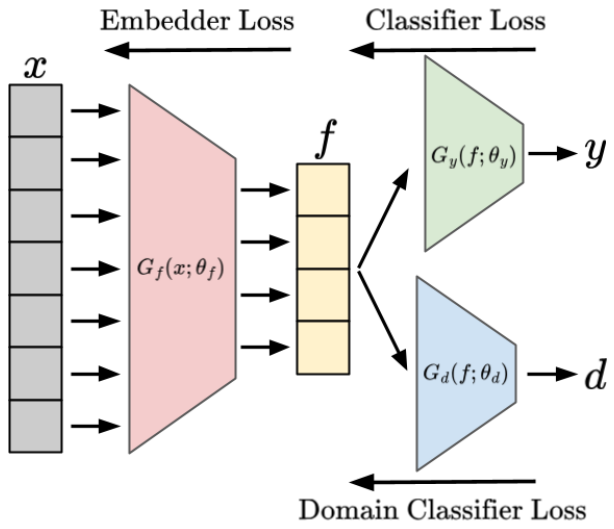


Figure 5.5: Architecture of a Domain-Adversarial Neural Network (DANN), highlighting the adversarial relationship between the feature extractor and the domain classifier.

Each component relies on its own respective loss function. The core premise of DANN is that G_f is trained with opposing objectives with respect to the losses of the label predictor and domain classifier. The feature extractor must simultaneously produce features that aid

the label predictor's success while actively degrading the accuracy of the domain classifier. This optimization can be expressed as a minimax objective:

$$\hat{\theta}_f, \hat{\theta}y = \arg\min \theta_f, \theta_y \left[\mathcal{L}_y(\theta_f, \theta_y) - \lambda \mathcal{L}_d(\theta_f, \hat{\theta}_d) \right], \qquad \hat{\theta}d = \arg\min \theta_d \mathcal{L}_d(\theta_f, \theta_d) \quad (5.19)$$

Source-Free Domain Adaptation (SFDA)

While DANNs perform well under domain adaptation, they require simultaneous access to both source and target data during training, as the loss functions are computed jointly. This presents a major limitation when source data cannot be retained or reused at adaptation time due to storage, privacy, or deployment constraints.

Source-Free Domain Adaptation (SFDA) methods perform adaptation to $\mathcal{D}T$ using only the model pre-trained on $\mathcal{D}S$ and unlabeled target data $x_i i \in T$, entirely independent of the original source data $(x_i, y_i) i \in S$. SFDA methods adapt the model using objectives that are computable solely from the target data and the pre-trained source model. The structural topology remains largely the same as DANN, but omits the explicit domain classifier G_d.

Entropy Minimization: The pre-trained label predictor's output distribution on a target sample x_i is treated as a pseudo-confidence signal. The model is fine-tuned to improve the confidence of its own predictions. SFDA works to minimize the following error function:

$$\mathcal{L}\text{ent}(\theta_f) = \frac{1}{n_T} \sum i \in T H \left(G_y(G_f(x_i)) \right), \qquad H(p) = -\sum_c p_c \log p_c$$

Self-training: The model's high-confidence predictions on target data are utilized as proxy labels. This allows the model to be iteratively refined using the most trusted target predictions as if they were ground truth. This process induces a shift in the decision boundary,

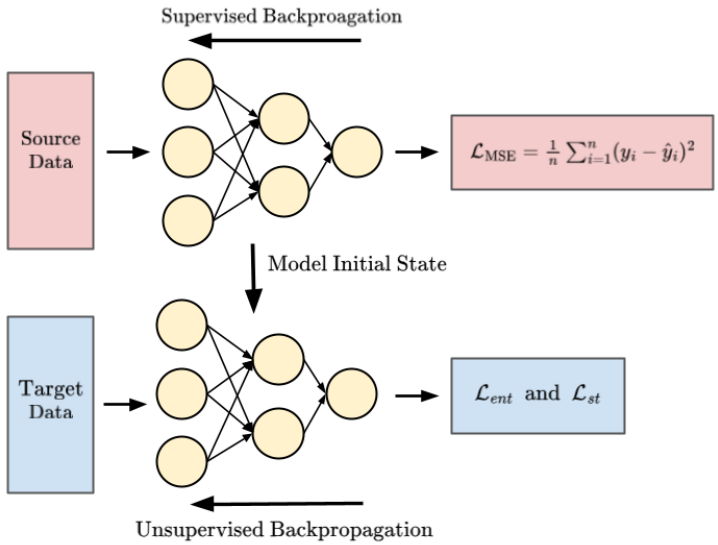


Figure 5.6: Workflow of Source-Free Domain Adaptation (SFDA), demonstrating adaptation using only unlabeled target data and a pre-trained model.

leading to a better fit for the target distribution.

5.3.5 Transfer Learning

Transfer learning is the reuse of knowledge gained from one task or domain to improve performance on a related task, rather than training a model from scratch. A model is first trained on a source task with abundant data, learning a representation f_θ that captures general structure. Some or all of the parameters θ can then be used as initialization for a target task:

$$\theta_{\text{target}} = \theta_{\text{source}} + \Delta\theta \tag{5.20}$$

where $\Delta\theta$ is learned by fine-tuning on the target task. Often, fine-tuning only updates later layers (such as the classification head) while keeping the earlier layers frozen. This approach operates under the assumption that early representations are more general, whereas later

representations are more task-specific. Source-Free Domain Adaptation (SFDA) functions as a specialized subset of transfer learning where no target task labels are available during the adaptation phase.

5.4 Hybrid LSTM-XGBoost Classifier

To effectively detect runtime microarchitectural attacks, such as Spectre-style vulnerabilities, the detection mechanism must resolve two conflicting requirements: identifying complex, long-range temporal patterns within hardware performance counter (HPC) telemetry, and maintaining a lightweight, highly accurate decision boundary suitable for real-time inference. Traditional models often struggle to balance these needs, either sacrificing deep feature extraction for execution speed or suffering from severe performance degradation under hardware-induced domain shifts.

To address these limitations, we propose a Hybrid LSTM-XGBoost Classifier. This framework synergizes the deep, temporal representation learning of Long Short-Term Memory (LSTM) networks with the robust classification capabilities of gradient-boosted decision trees (XGBoost). The following subsections detail the architectural design of this hybrid model, its dual-stream feature embedding backbone, the boosted classification mechanism, and the source-free domain adaptation strategy utilized to maintain resilience across varying domains.

5.4.1 Model Architecture

The hybrid architecture is partitioned into two distinct stages: an initial feature embedding backbone that rapidly captures long-range temporal dependencies and preserves representational information across multiple domains, and a subsequent gradient-boosted tree classifier that processes the backbone's combined output to generate the final system predictions. Specifically, the embedding backbone comprises an LSTM encoder (f_1) and a statistical

function (f_2). Their respective outputs are concatenated to form a unified feature vector $[f_1 \parallel f_2]$, which is then fed into the boosted-tree model to perform the final classification, as illustrated in Figure 5.7.

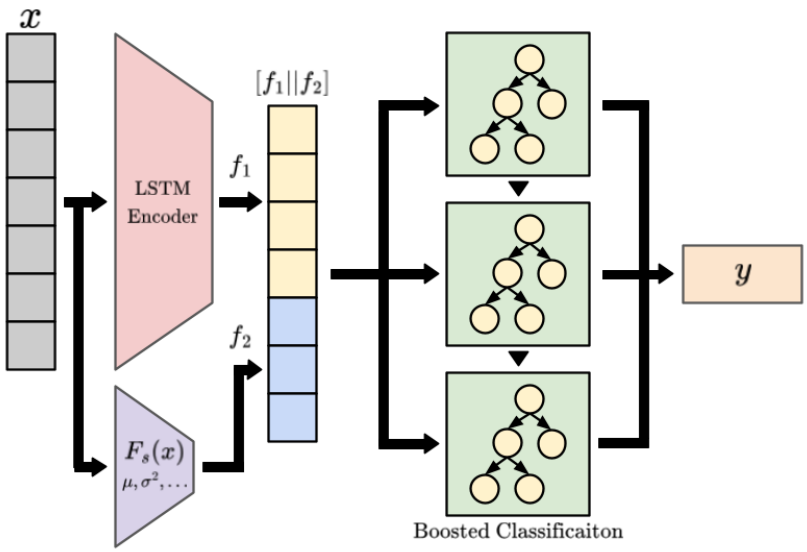


Figure 5.7: Architecture of the Hybrid LSTM-XGBoost classification model.

5.4.2 Feature Embedding Backbone

The feature embedding backbone consists of two parallel processing streams: the LSTM encoder and the deterministic statistical function $F_s(x)$.

As illustrated in Figure 5.8, the LSTM encoder is pre-trained independently prior to its integration into the full pipeline. This training is conducted via a supervised contrastive loss applied to its embeddings, paired with a pseudo-linear classifier that serves as a differentiable proxy for the non-differentiable XGBoost classification head. Specifically, this pre-training is conducted jointly alongside two auxiliary modules: a linear classification head and a non-linear projection head. The optimization is governed by a unified training objective:

$$\mathcal{L}_{\text{source}} = \mathcal{L}_{\text{CE}} + \lambda_{\text{con}} \mathcal{L}_{\text{SupCon}} \tag{5.21}$$

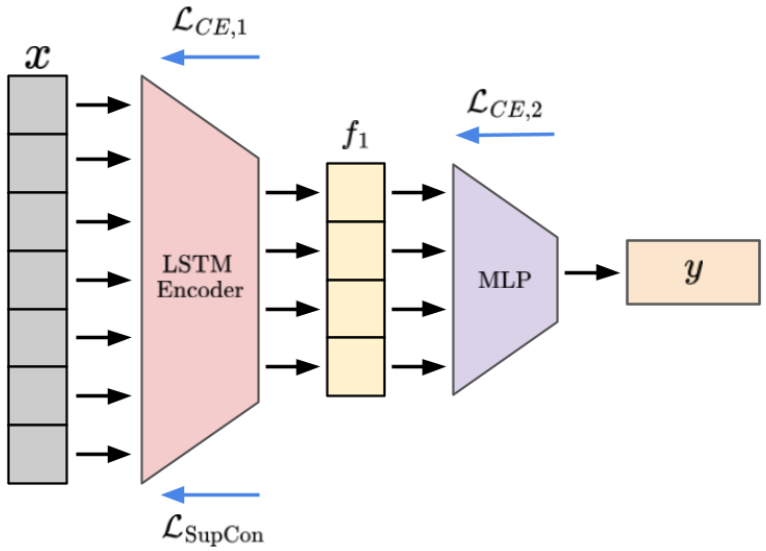


Figure 5.8: Independent pre-training pipeline for the LSTM encoder.

$\mathcal{L}_{\text{CE}}$ represents the class-weighted cross-entropy loss of the classification head, which directly mitigates potential label imbalance within the training dataset. $\mathcal{L}_{\text{SupCon}}$ represents the supervised contrastive loss [15], defined as:

$$\mathcal{L}_{\text{SupCon}} = -\frac{1}{|P(i)|} \sum_{i} \sum_{p \in P(i)} \log \frac{\exp(z_i \cdot z_p/\tau)}{\sum_{a \neq i} \exp(z_i \cdot z_a/\tau)} \tag{5.22}$$

where $P(i)$ denotes the set of same-label samples within the batch and τ is a temperature scaling parameter. While the cross-entropy loss shapes the explicit decision boundary, $\mathcal{L}_{\text{SupCon}}$ actively regularizes the geometry of the embedding space. This objective structurally encourages same-label samples to cluster tightly together while forcing different-label samples apart—a topological property that is highly advantageous for downstream domain adaptation. Once the LSTM is trained on the source distribution, its output representation is denoted as f_1.

In parallel, the statistical function $F_s(x)$ computes deterministic features over the raw input

sequence, capturing metrics such as the mean, median, standard deviation, skewness, slope, and magnitude. A comprehensive list of these statistical formulations is provided in the Appendix. Broadly, $F_s(x)$ quantifies central tendency, distributional shape, and temporal trends. These statistics are structured into a secondary feature vector f_2 maintaining a fixed, consistent ordering. Because $F_s(x)$ calculates mathematically pre-defined metrics, it remains entirely static and is not subjected to gradient updates during model training or adaptation.

The final comprehensive feature embedding is generated via vector concatenation, resulting in $[f_1 \parallel f_2]$. Explicit feature concatenation has been demonstrated to enhance both embedding efficacy and model interpretability [4, 12]. This structural design acts as a critical information retention mechanism and establishes a direct shortcut for gradient descent, facilitating accelerated model convergence.

5.4.3 Boosted Classification Head

The boosted classification head receives the concatenated feature vector from the embedding backbone. This component utilizes a standard XGBoost architecture without structural modifications; however, its input is explicitly defined by the learned and statistical embedding representations rather than the raw telemetry windows. As detailed in Table 6.1, XGBoost empirically outperforms baseline architectures when supplied with complete domain knowledge (including device microarchitecture, system configurations, and specific attack/benign data distributions). This module is responsible for the ultimate classification output of the framework.

5.4.4 Source-Free Domain Adaptation (SFDA)

In standard real-world deployments, maintaining simultaneous access to both the original source training corpus and the streaming target data is highly impractical. The framework accommodates this constraint by adapting to novel target distributions—where labels are

strictly unavailable—utilizing Source-Free Domain Adaptation (SFDA).

The adaptation follows a SHOT-style procedure [26]: the auxiliary classification and projection heads are explicitly frozen, restricting parameter updates solely to the encoder. The adaptation objective is formulated as:

$$\mathcal{L}_{\text{SFDA}} = \lambda_{\text{ent}}\mathcal{L}_{\text{ent}} + \lambda_{\text{cls}}\mathcal{L}_{\text{pl}} \tag{5.23}$$

This objective minimizes information entropy, defined as $\mathcal{L}_{\text{ent}} = -\mathbb{E}_i \left[\sum_c p_{i,c} \log p_{i,c}\right]$, which forces the network to generate high-confidence predictions on the target distribution. Simultaneously, pseudo-label supervision, formulated as $\mathcal{L}_{\text{pl}} = -\log p_{i,\hat{y}_i}$, supplies a proxy classification signal. The target variable $\hat{y}_i$ is derived through a constrained number of weighted k-means refinement iterations executed within the embedding space. This iterative refinement allows the framework to generate pseudo-labels that exhibit high stability even when subjected to substantial microarchitectural distribution shifts.

Chapter 6

Model Validation and Live Client-Server Evaluation

Chapter 5 introduced the design of the Hybrid DA-LSTM and XGBoost framework. This chapter evaluates how well that system actually performs. The evaluation has two main parts. First, we test the hybrid model against standard and adaptive baselines in controlled offline settings. This step confirms the model's core accuracy, its ability to extract features, and its robustness when facing different types of domain shift. Second, we move from offline testing to a real-world setup by deploying the system in a live client-server pipeline. This tests how well the model handles real-time operating system interrupts, network lag, and unpredictable background noise across different hardware devices.

6.1 Comparative Validation of the Hybrid Architecture

Before testing the system in a live environment with varying hardware, we must first establish its baseline performance. This section evaluates the hybrid model's core classification

accuracy. We compare it to both traditional and adaptive models to show the benefits of combining deep learning for feature extraction with a tree-based classifier.

6.1.1 Intra-Domain Baseline Performance

In this subsection, we establish a baseline for the performance of various common models on the Spectre detection time-series dataset. The models are trained and tested on the same device, thus experiencing no domain shift.

The datasets used exhibit a high degree of class imbalance between Spectre attack data and benign data, as is typical in anomaly detection tasks. Consequently, our analysis focuses on precision, recall, and F1-score rather than raw accuracy. In this context, high precision indicates that the model effectively minimizes false positives, whereas high recall signifies its capability to reduce false negatives.

In the field of anomaly detection, high recall is generally prioritized because an undetected security breach carries far greater consequences than the resource overhead incurred by investigating a false alarm. The model hyperparameter and training settings can be found in the appendix.

The baseline results in Table 6.1 demonstrate model performance under controlled intra-domain conditions without domain shift. Across all four hardware platforms, the hybrid LSTM-XGBoost model consistently outperforms the standalone LSTM baseline. On Intel 1, for example, LSTM-XGBoost achieves an F1-score of 0.7812 compared to 0.7354 for the standalone LSTM, with similar improvements observed on AMD (0.5840 vs. 0.4934) and Raspberry Pi (0.8890 vs. 0.8236). Combining sequential deep feature representations with boosted tree decision heads provides a clear advantage over relying solely on continuous temporal features, leading to higher precision and improved overall stability across cross-validation folds.

Table 6.1: Performance metrics (mean ± standard deviation with $K = 5$ folds) evaluated across different methods and device datasets.

Dataset	Metric	Intel 1	Intel 2	RPi	AMD
LSTM	Accuracy	0.9054 ± 0.0279	0.8039 ± 0.0892	0.7312 ± 0.1368	0.8126 ± 0.0994
	Precision	0.6413 ± 0.1351	0.7560 ± 0.1301	0.9439 ± 0.0294	0.4872 ± 0.2840
	Recall	0.8874 ± 0.0569	0.7231 ± 0.1419	0.7446 ± 0.1635	0.6413 ± 0.1796
	F1-Score	0.7354 ± 0.0793	0.7247 ± 0.0900	0.8236 ± 0.1018	0.4934 ± 0.1900
XGBoost	Accuracy	0.9401 ± 0.0347	0.8734 ± 0.0921	0.8734 ± 0.0501	0.8954 ± 0.0534
	Precision	0.7836 ± 0.1465	0.8428 ± 0.0973	0.9606 ± 0.0253	0.6688 ± 0.2756
	Recall	0.8889 ± 0.0387	0.7085 ± 0.1848	0.9190 ± 0.0638	0.7767 ± 0.2043
	F1-Score	0.8251 ± 0.0856	0.7517 ± 0.1147	0.9380 ± 0.0301	0.6559 ± 0.1423
TCN-VAE	Accuracy	0.6812 ± 0.2463	0.5017 ± 0.1056	0.8842 ± 0.0466	0.6266 ± 0.1468
	Precision	0.3023 ± 0.2400	0.4113 ± 0.1661	0.8844 ± 0.0468	0.1861 ± 0.1787
	Recall	0.3044 ± 0.3049	0.7191 ± 0.1966	0.9998 ± 0.0003	0.4190 ± 0.3146
	F1-Score	0.2003 ± 0.0969	0.4937 ± 0.1255	0.9380 ± 0.0271	0.2471 ± 0.2147
SDFA	Accuracy	0.8904 ± 0.0400	0.8394 ± 0.0621	0.7715 ± 0.1086	0.8335 ± 0.0782
	Precision	0.6026 ± 0.1487	0.7963 ± 0.1101	0.9139 ± 0.0382	0.5380 ± 0.3190
	Recall	0.8914 ± 0.0366	0.7612 ± 0.1072	0.8208 ± 0.1224	0.6368 ± 0.1809
	F1-Score	0.7094 ± 0.1037	0.7700 ± 0.0645	0.8610 ± 0.0709	0.5128 ± 0.1733
LSTM-XGBoost	Accuracy	0.9235 ± 0.0310	0.8412 ± 0.0820	0.8120 ± 0.0840	0.8540 ± 0.0710
	Precision	0.7150 ± 0.1380	0.7920 ± 0.1110	0.9510 ± 0.0280	0.5820 ± 0.2410
	Recall	0.8880 ± 0.0480	0.7180 ± 0.1510	0.8350 ± 0.1120	0.7120 ± 0.1820
	F1-Score	0.7812 ± 0.0810	0.7410 ± 0.0980	0.8890 ± 0.0650	0.5840 ± 0.1610

6.1.2 Model Degradation and Adaptation Under Domain Shift

Background Load Variation

To understand how background noise affects detection, we tested each device under four different load conditions: Idle, CPU-Intensive, Memory-Intensive, and Mixed-Intensive. These background tasks generate noise that can hide the hardware signatures of a Spectre attack. We first trained and tested a lightweight LSTM model individually on each of these configurations (see Figure 6.1).

Figure 6.1 shows how the standard LSTM model struggles when background activity changes. In the Idle state, the model easily spots the attack because the background is quiet, resulting in high recall. However, it also produces more false positives because it is highly sensitive to minor fluctuations. On the other hand, heavy background tasks—especially CPU-Intensive

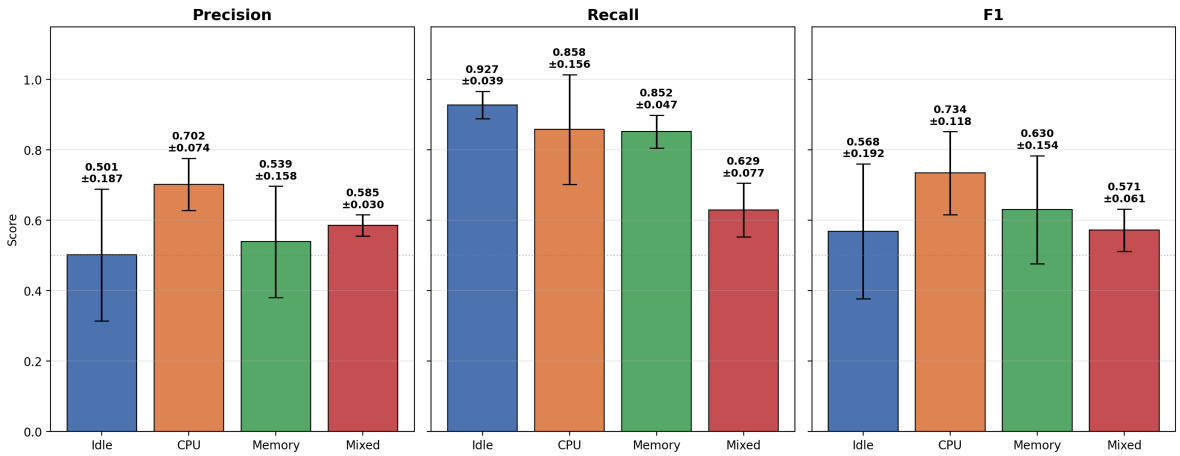


Figure 6.1: Lightweight LSTM Model Performance when individually trained on each background load config for Intel Processor 1.

and Mixed loads—create noise that covers up the attack's hardware traces. Because normal background activity and the Spectre attack look so similar, the model misses many attacks (increasing false negatives). This proves that standard models cannot reliably detect attacks when background noise levels fluctuate.

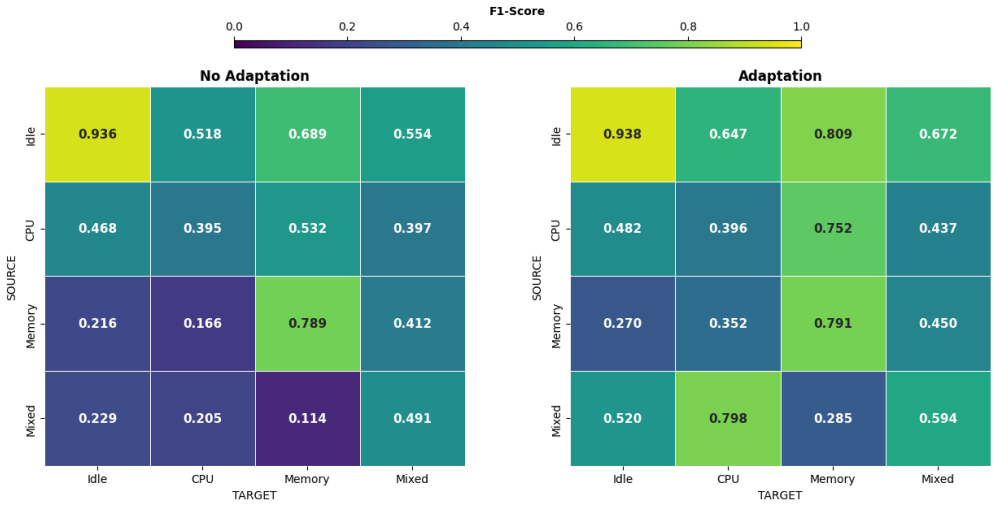


Figure 6.2: Hybrid LSTM-XGBoost Model Cross-Config Performance on Intel Processor 1 with and without domain adaptation.

We apply source-free domain adaptation (SFDA) to solve this problem across the different load configurations. Standard classifiers like XGBoost cannot adjust to new, unlabeled data, and their accuracy drops entirely if forced to adapt this way. In contrast, our hybrid frame-

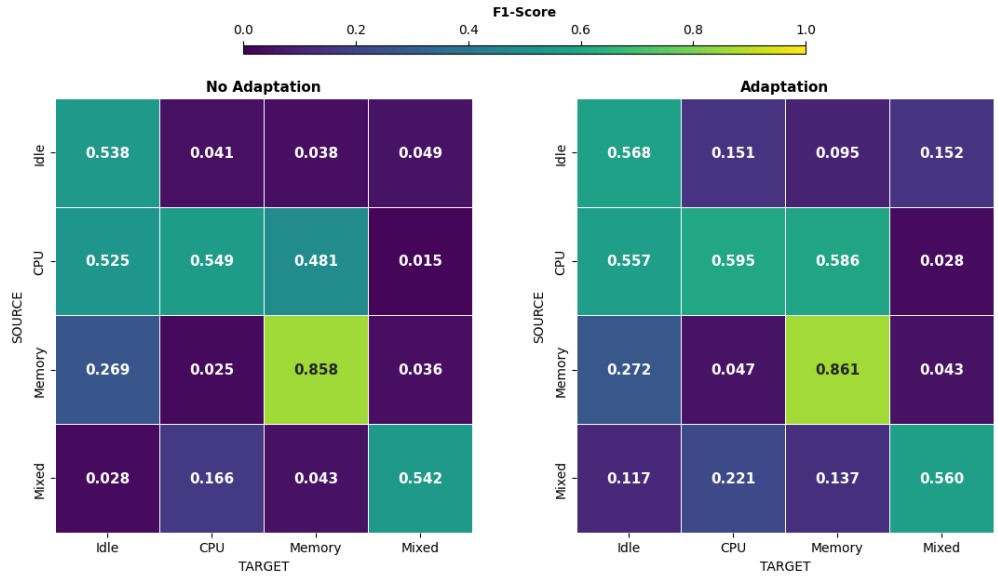


Figure 6.3: Standard LSTM model Cross-Config Performance on Intel Processor 1 with and without domain adaptation.

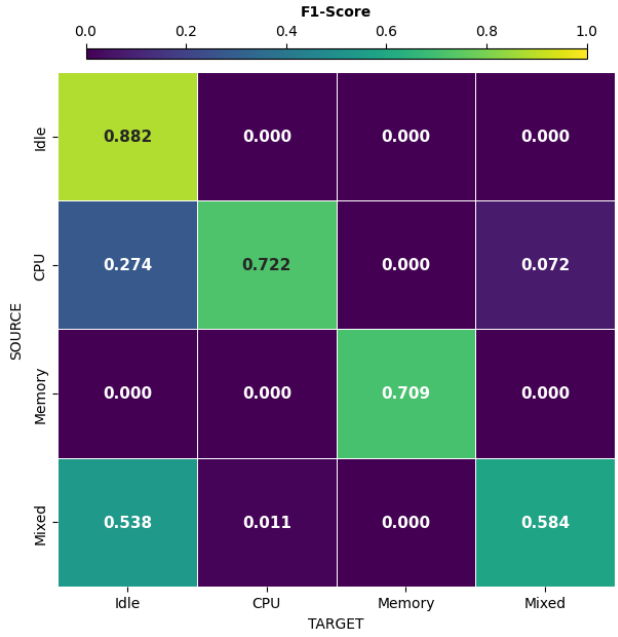


Figure 6.4: XGBoost model Cross-Config performance on Intel Processor 1. XGBoost model is incapable of fine-tuning on unlabelled data.

work successfully realigns the new data without needing labels. As the adaptation heatmaps show, adjusting only the encoder portion of our model recovers a significant amount of accuracy, particularly when moving to CPU and Mixed loads. This demonstrates that updating how the model processes the data features, while keeping the final classifier unchanged, allows it to detect attacks accurately despite severe changes in background noise. Importantly, it achieves this without needing new labeled target data or retraining the classifier.

Attack Pacing Variations

To test how well the model adapts to different attack speeds, we analyzed three attack pacing patterns: batching (ba), bursting (br), and ratio-based sampling (r). We trained the models on two of these patterns combined and then tested them on the third unseen pattern. Training on multiple patterns at once helps the model learn a wider range of timing behaviors, giving it a strong foundation before it faces a completely new attack pace.

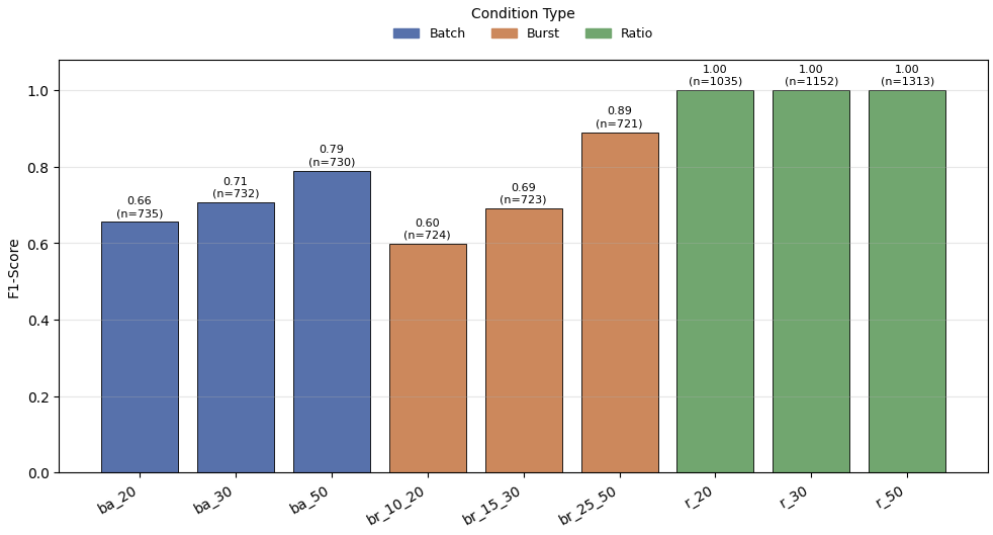


Figure 6.5: XGBoost performance transfer on different attack pacing patterns.

Even though they were trained on multiple patterns, standard models still struggle when tested on a new attack pace. This drop in performance happens because the timing of the data changes. Standard models, such as XGBoost, cannot adapt to these new timing patterns without labeled data, resulting in lower accuracy.

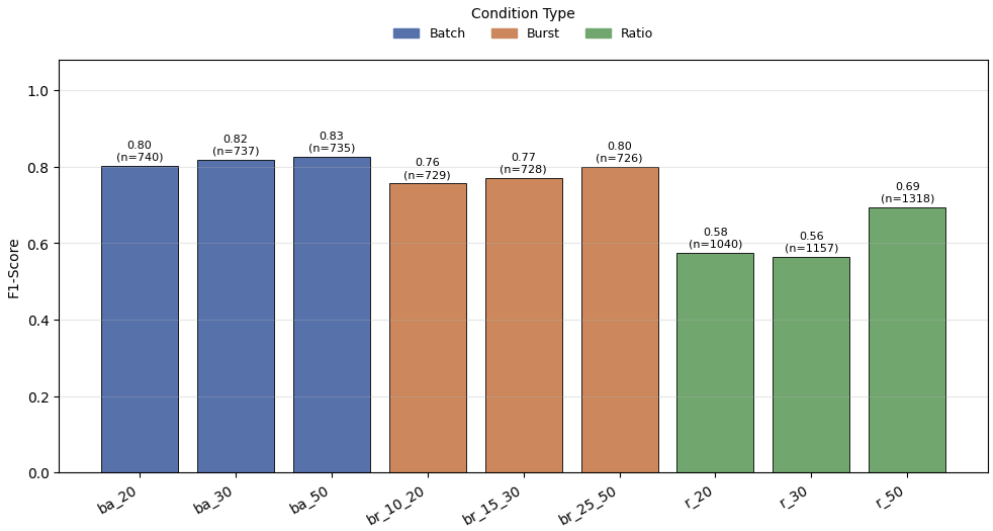


Figure 6.6: Hybrid LSTM-XGBoost performance transfer on different attack pacing patterns.

In contrast, our hybrid framework uses source-free domain adaptation (SFDA) to automatically update itself using the new, unlabeled target data as it arrives. By adjusting its understanding of the data's timing to match the new attack pace, the hybrid model successfully restores its detection accuracy. It achieves this across all tested pacing patterns without needing to retrain the classifier or access the original training data.

Attack Method Variation

Evaluating a model's resilience to different attack strategies is critical for real-world deployment, as exploit mechanisms constantly evolve. To test this, we examined model behavior across three distinct attacks: Spectre v1 (bounds check bypass), Spectre v2 (branch target injection), and Spectre Prime+Probe (cache side-channel measurement). Figure 6.7 shows the baseline difficulty of detecting each variant when the training and testing conditions match, using a lightweight LSTM. Spectre v1 and v2 have strong baseline detection rates because their branch prediction anomalies leave clear microarchitectural traces. In contrast, Spectre Prime+Probe is much harder to detect; its subtle cache access patterns blend tightly into regular memory traffic, making it difficult to isolate even in controlled setups.

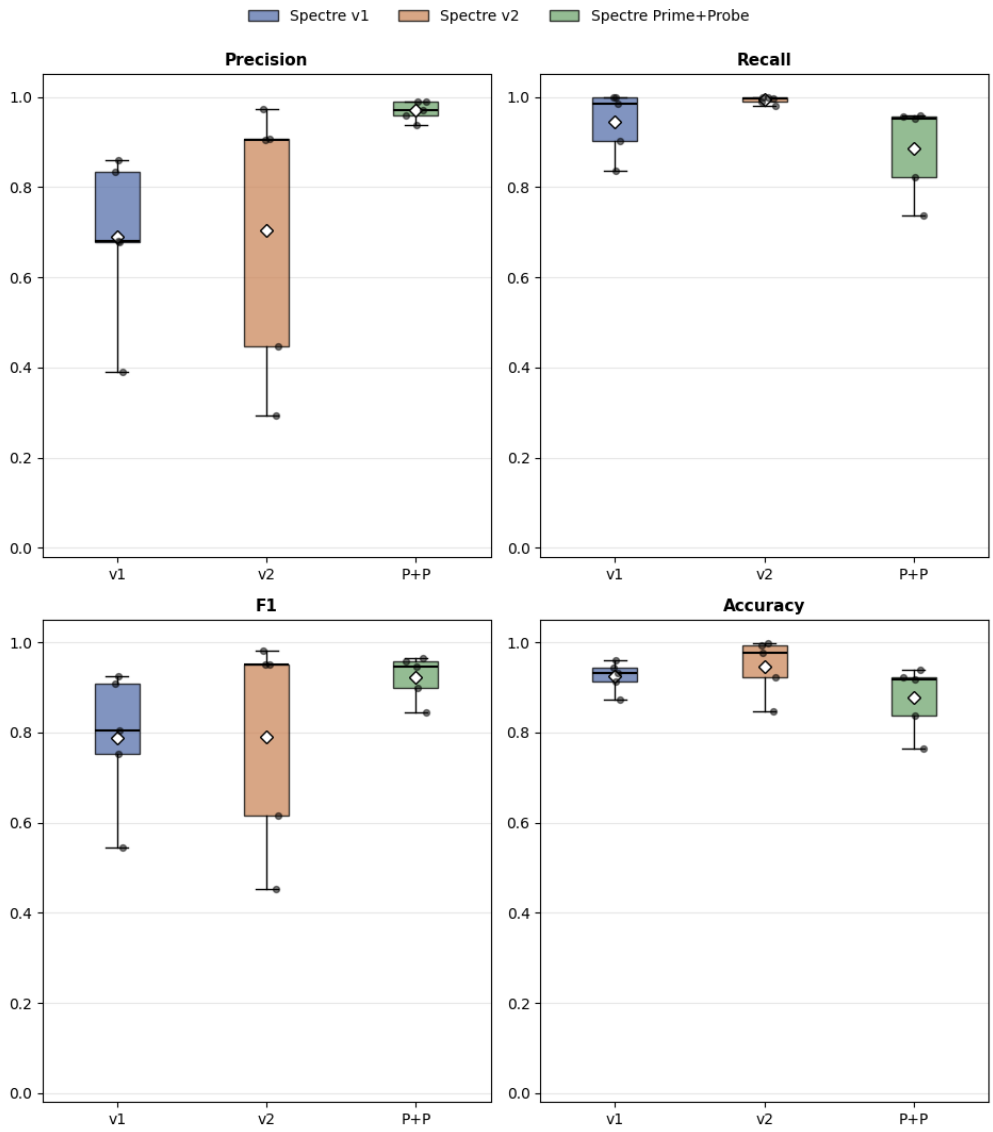


Figure 6.7: Lightweight LSTM model trained and tested on each individual Spectre attack variation.

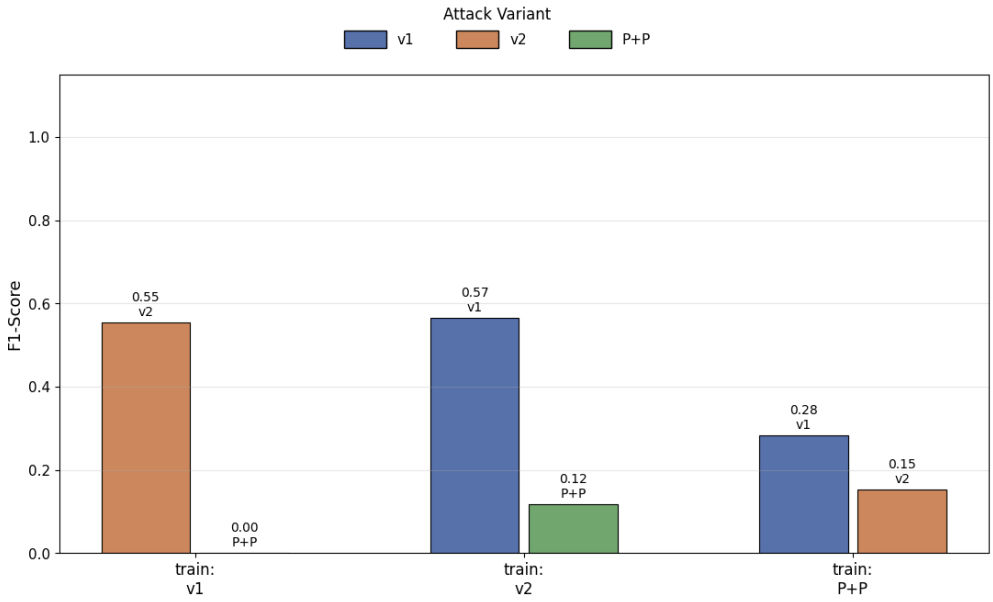


Figure 6.8: XGBoost performance transfer on different attack variants.

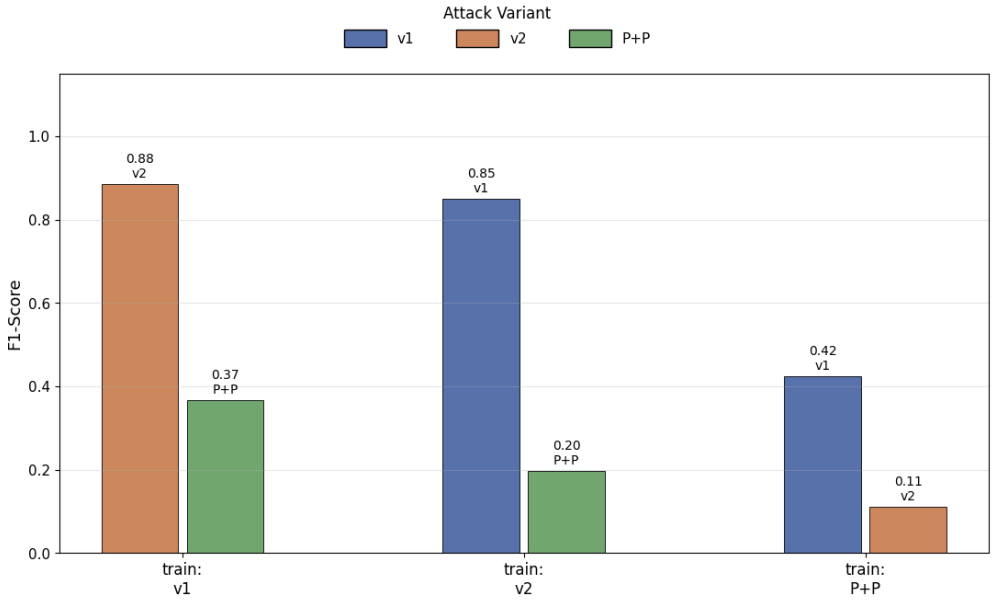


Figure 6.9: Base LSTM performance transfer on different attack variants after adaptation.

When models trained on one attack variant are tested against unfamiliar variants without retraining, their performance drops sharply. A standard XGBoost model (Figure 6.8) degrades significantly when transferring across attack types, especially when moving from speculative execution exploits (v1/v2) to memory-focused tactics like Prime+Probe. Furthermore, because tree-based models lack a mechanism for unsupervised fine-tuning, they cannot adapt once deployed on unlabeled live data. Standard LSTMs (Figure 6.9) offer some flexibility through source-free adaptation, but their basic feature representations struggle to bridge the structural gaps between different attack types.

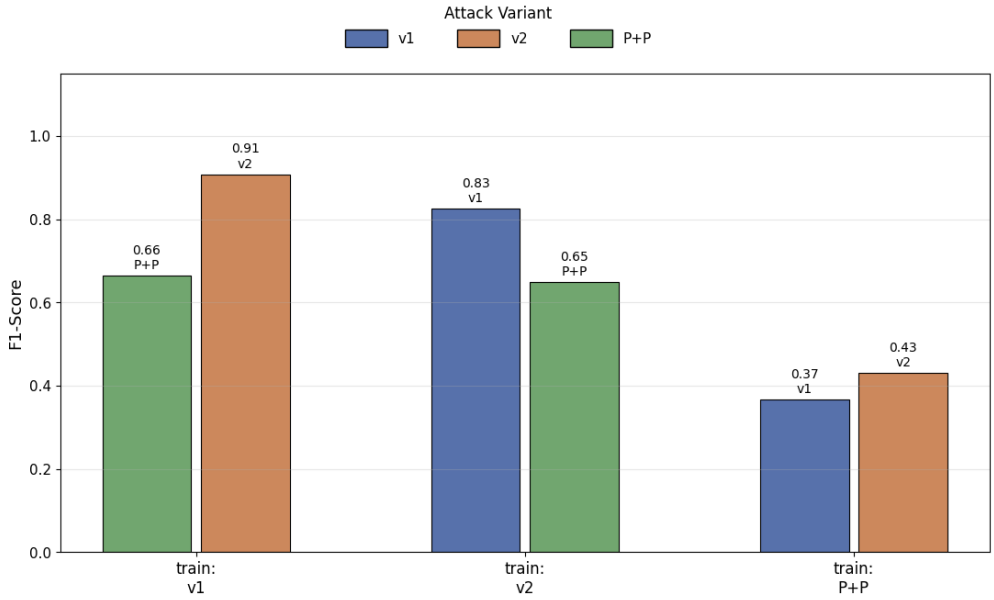


Figure 6.10: Hybrid LSTM-XGBoost performance transfer on different attack variants after domain adaptation.

Our proposed Hybrid LSTM-XGBoost framework (Figure 6.10) overcomes these transfer limitations. By using unsupervised entropy minimization and pseudo-label refinement within its temporal embedding backbone, the hybrid model realigns the target feature distributions before passing them to the decision tree. This allows the network to dynamically adapt to new attack behaviors without needing labeled target data, ensuring robust protection against previously unseen Spectre variants.

Device Variation

Testing across different devices is the hardest challenge in our setup because hardware differences change how the attack signatures look. We trained the models on an older Intel processor (PC1) and tested them on a newer one (PC2). Even though they are both Intel chips, small changes in how they process data cause major differences in the attack signals.

Table 6.2: Cross-device adaptation performance (PC1 → PC2).

Model	Setting	Precision	Recall	F1 Score
Our Model	Source Only	0.758 ± 0.048	0.120 ± 0.009	0.207 ± 0.016
	Domain Adaptation	$\mathbf{0.576 \pm 0.063}$	$\mathbf{0.581 \pm 0.037}$	$\mathbf{0.570 \pm 0.016}$
XGBoost	Source Only	0.296 ± 0.074	0.293 ± 0.032	0.288 ± 0.047
	Domain Adaptation	—	—	—
LSTM	Source Only	0.795 ± 0.048	0.187 ± 0.026	0.301 ± 0.036
	Domain Adaptation	0.491 ± 0.035	0.529 ± 0.031	0.509 ± 0.030

Table 6.2 shows that without adaptation, the models miss most of the attacks. Their recall drops severely (0.120 for our model and 0.187 for the LSTM), even though their precision seems high. In cybersecurity, high recall is much more important than high precision. It is better to have a few false alarms than to miss a real Spectre attack. Because they miss so many attacks, the unadapted models are essentially useless on the new device.

Applying source-free domain adaptation (SFDA) fixes this problem. Our model recovers its ability to detect attacks, achieving a recall of 0.581 and an F1-score of 0.570. This is better than the adapted standalone LSTM model. Standard tree models like XGBoost cannot adapt without labeled data. However, our hybrid model successfully adjusts to the new hardware. It trades a little bit of precision for a large improvement in recall, making it reliable across different devices.

Precision-Recall Performance: Figures 6.11 and 6.12 show how precision and recall change when moving from PC1 to PC2. Before adaptation, the model's recall drops sharply, missing most attacks. After applying SFDA, the curve moves up and to the right, showing

that the model can spot the attacks again. Although we lose a tiny bit of precision, we gain a massive amount of recall. For stopping Spectre attacks, prioritizing recall is essential so that threats do not slip by unnoticed.

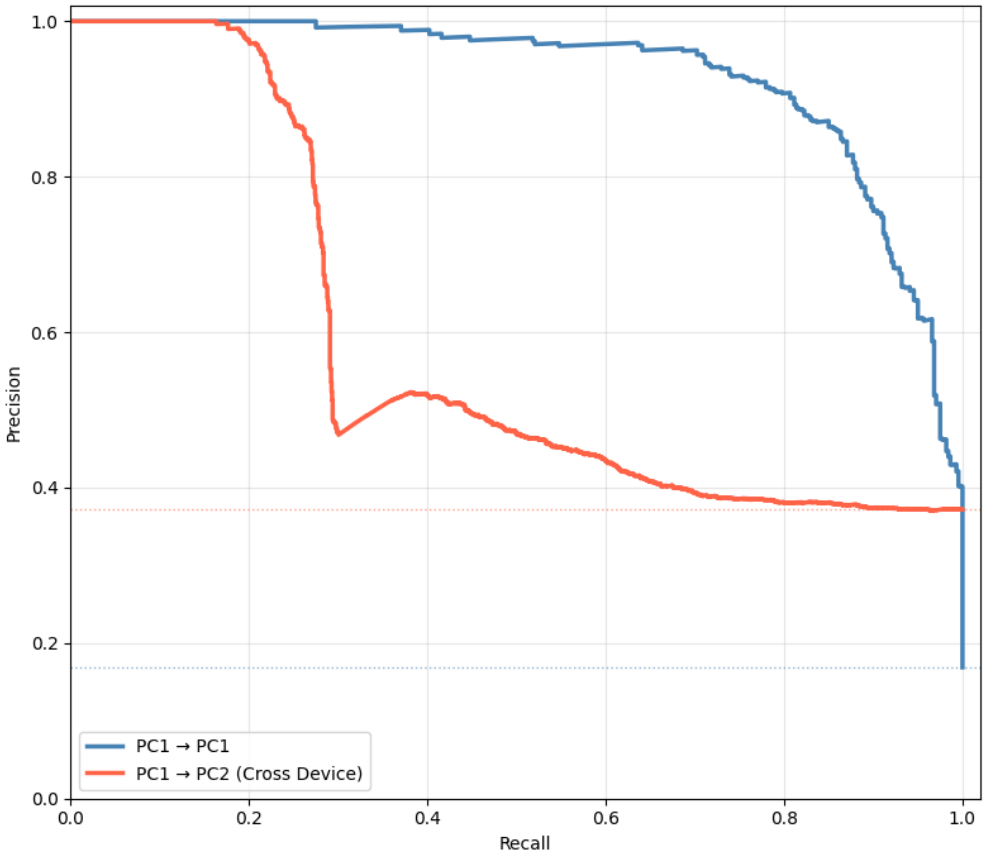


Figure 6.11: Precision-Recall graph for the XGBoost Model on Intel 1 to Intel 2 device transfer.

Adaptation Trajectory: To see how quickly SFDA learns, we measured its performance as we gave it more unlabeled data from the new devices (PC1 → PC2 and PC1 → AMD). In both cases, the model improves very quickly using only a small amount of data. When moving to the AMD processor, recall jumps from 0.18 to over 0.65 using just the first 3,000 data samples, pushing the F1-score from 0.25 to 0.60. The move to PC2 shows similar fast growth in recall, proving the model adjusts quickly without needing huge amounts of data.

The model reaches its peak performance after seeing only about 5,000 to 7,000 samples.

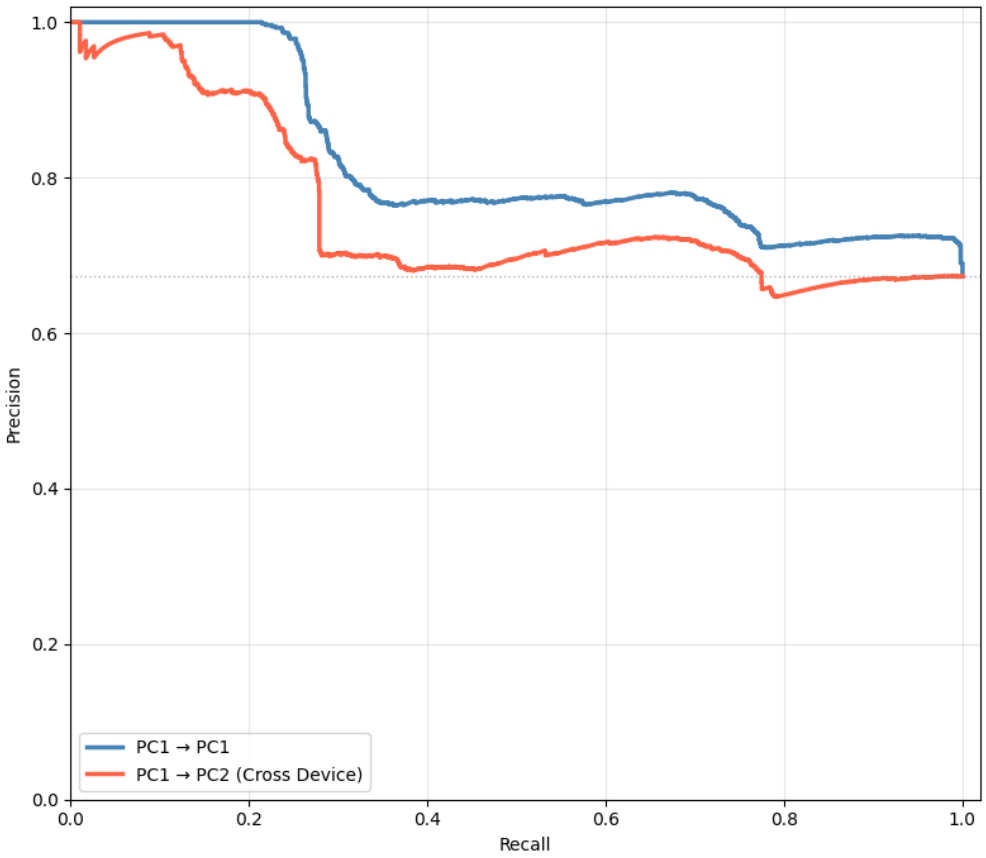


Figure 6.12: Precision-Recall graph for the Hybrid LSTM-XGBoost Model on Intel 1 to Intel 2 device transfer.

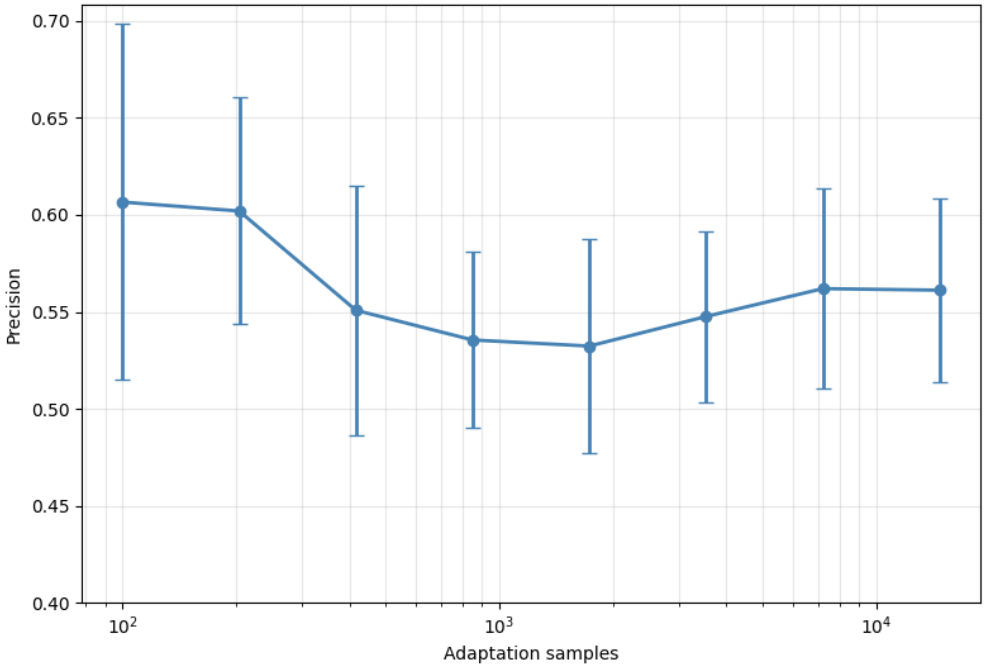


Figure 6.13: Trajectory of model precision with respect to the size of the adaptation dataset.

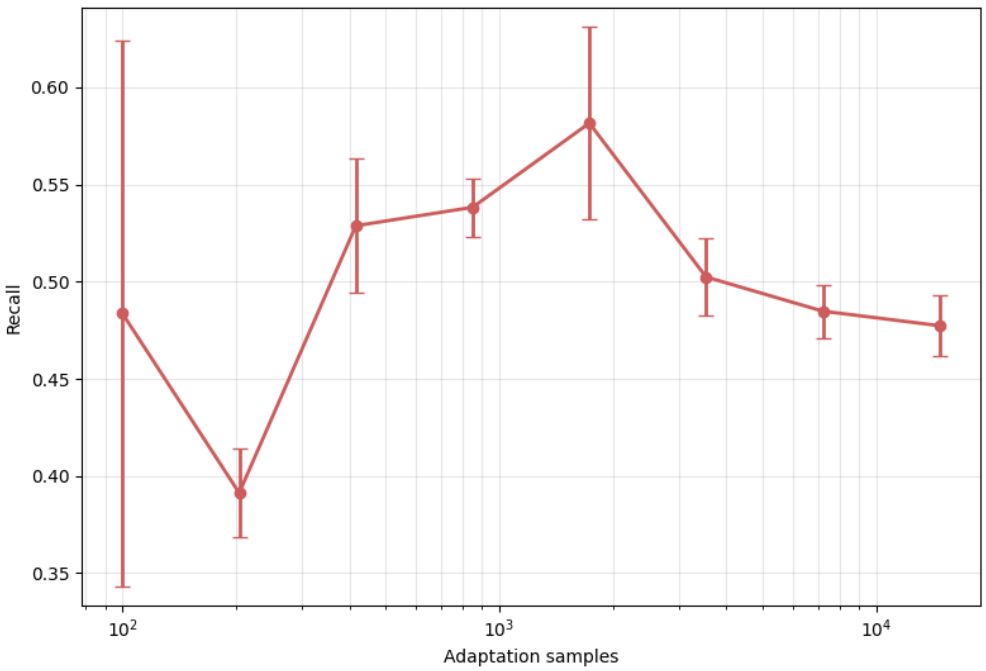


Figure 6.14: Trajectory of model recall with respect to the size of the adaptation dataset.

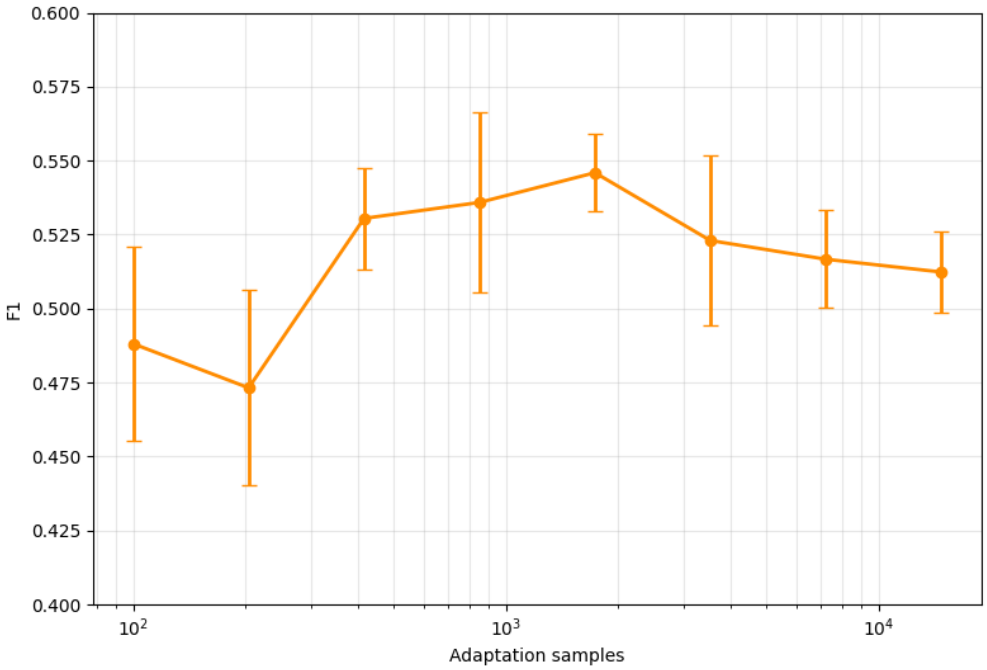


Figure 6.15: Trajectory of model F1-Score with respect to the size of the adaptation dataset.

While precision fluctuates slightly at the beginning as the model adjusts, recall consistently goes up. This steady increase in recall guarantees that the model stays focused on finding threats even when the hardware changes.

6.1.3 Model Inference and Memory Performance

Beyond classification accuracy, a real-time intrusion detection system must process data quickly (low latency) and handle a high volume of data (high throughput). Table 6.3 outlines the computational size, latency, and sample throughput for each model. These tests were run one sample at a time on an Intel(R) Xeon(R) CPU @ 2.00GHz.

The standalone neural networks, including the lightweight LSTM, TCN-VAE, and SDFA, all process samples in under 3 milliseconds (ms). Because it has the fewest parameters, the LSTM is the fastest, taking only 0.8523 ms per sample and processing $1,353.8$ samples per second.

Table 6.3: Comparison of computational complexity and runtime efficiency across models.

Model	**Parameters**	**FLOPs**	**Latency ± SD (ms)**	**Throughput ± SD (samples/s)**
LSTM	22,081	393,280	0.8523 ± 0.4656	$1,353.8 \pm 385.4$
XGBoost	–	–	25.8691 ± 0.4499	38.7 ± 0.6
TCN-VAE	51,852	744,960	2.8269 ± 0.1800	355.1 ± 21.0
SDFA	131,072	1,863,616	2.6990 ± 0.1219	375.3 ± 17.3
Our Model	262,144	2,639,528	29.1972 ± 0.5860	34.3 ± 0.7

In contrast, the tree-based models take longer. The standalone XGBoost and our hybrid model average 25.8691 ms and 29.1972 ms per sample, respectively. However, most of this delay does not come from the models making predictions. Instead, it comes from the data preparation steps required beforehand, such as extracting features and calculating rolling statistics.

Even though our hybrid model is slower because it calculates these statistics alongside generating deep features, it still processes 34.3 samples per second. This speed is more than sufficient to meet the practical requirements for real-time monitoring and processing data in batches.

6.2 Live Client-Server Pipeline Evaluation

To evaluate the real-world viability of our Hybrid DA-LSTM + XGBoost model, we deployed it within a live client-server architecture (see Section 3.6) across two hardware targets: an Intel x86 workstation (Intel PC 2) and an ARM-based Raspberry Pi (ARM RPi). This live deployment tests the model's resilience against realistic operational noise, operating system interrupts, and network latency, moving beyond static offline validations.

6.2.1 Initial Deployment and Domain Shift (Pre-Adaptation)

When the base model—trained entirely on source data—is initially deployed on live targets, its classification efficacy deteriorates immediately due to domain shift. Figure 6.16 illustrates how this microarchitectural mismatch affects distinct hardware architectures in opposing ways.

On the ARM RPi (Figure 6.16a), the model becomes hyper-sensitive to the target's baseline noise, resulting in extreme false positive saturation. The prediction probability remains pinned near 1.0 even during benign execution phases, yielding 57 false positives and rendering the system unusable for practical monitoring.

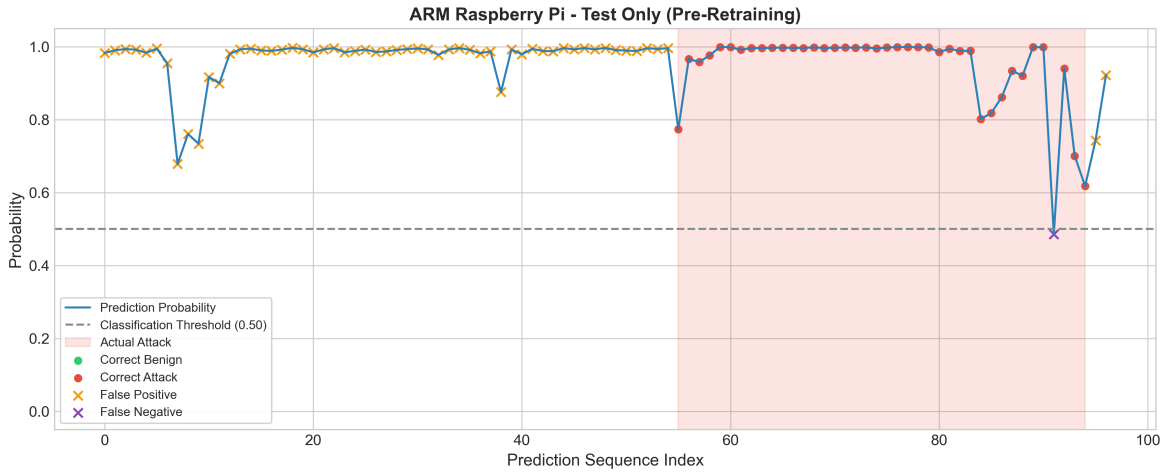
Conversely, the Intel workstation (Figure 6.16b) experiences a complete detection collapse. The live baseline telemetry on the Intel machine is unexpectedly quiet, preventing genuine attack signals from breaching the 0.50 classification threshold. The unadapted model strictly predicts benign states, resulting in a 0% recall rate (34 false negatives across 34 attack windows). This split in how the model fails proves that fixed models cannot protect diverse hardware without being tuned for the specific device.

To understand why the model fails, we need to look at how the data changes between the training and live environments. Figure 6.17 shows this shift by plotting the Branch Miss Ratio against the Cache Miss Ratio for both systems.

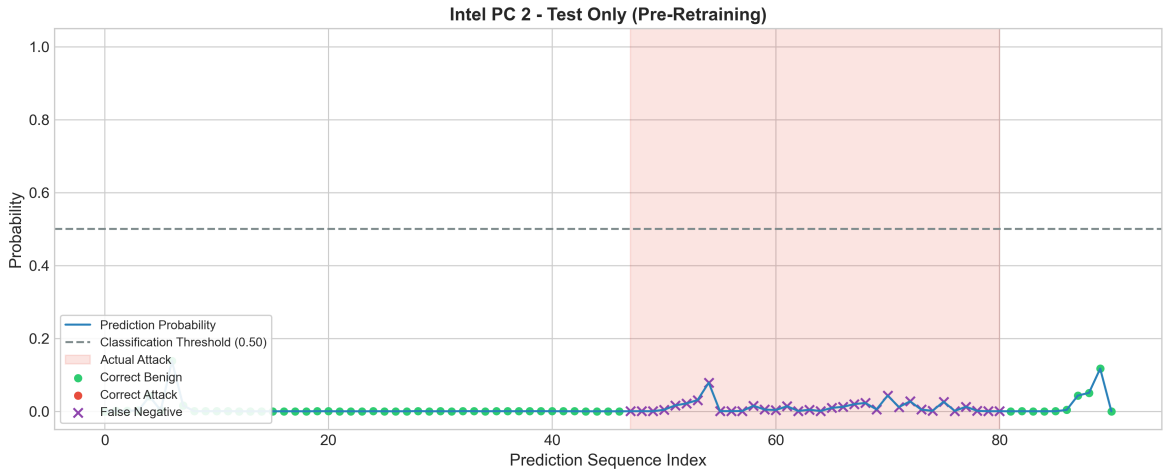
For the left-hand "Training" plots, we specifically selected training data where the attack was slowed down using a delay flag (`-r`). We chose to visualize this specific subset of the training data because it clearly illustrates the baseline separation. As shown, this controlled pacing creates a clear visual gap between the normal (Idle) data and the Attack data, representing the distinct boundaries the model learned.

However, during live deployment, this clean separation disappears. On the ARM RPi (Fig-

ure 6.17a), the live normal data shifts directly into the area that represents an attack. This overlap causes the constant false alarms shown in the timeline. On the Intel workstation (Figure 6.17b), the live attack data spreads out and no longer matches the tight clusters learned during training. Because the live attack data looks completely different, the model misses the attacks entirely.

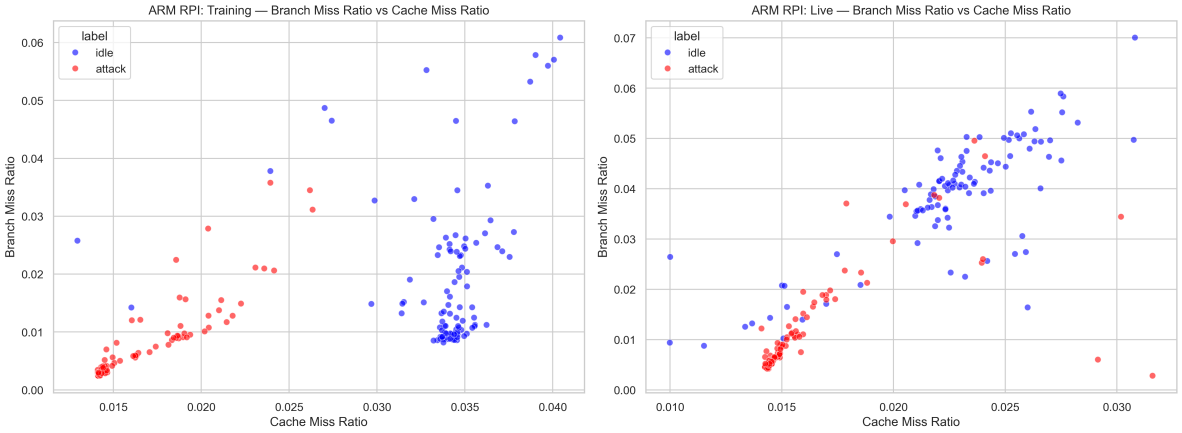


(a) ARM RPi (Test Only): High false positive saturation.

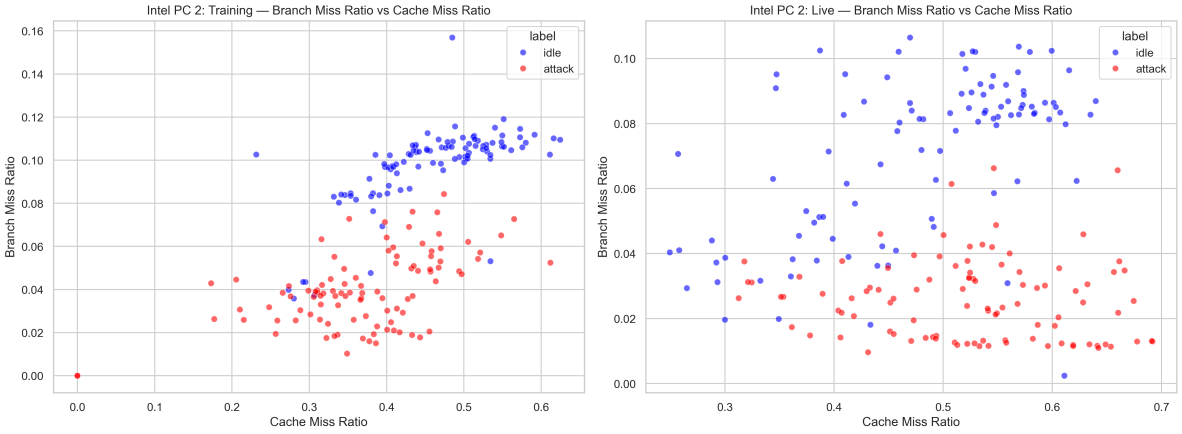


(b) Intel PC 2 (Test Only): Complete detection collapse (0% recall).

Figure 6.16: Live deployment timelines showing the failure of the model before adaptation across different hardware.



(a) ARM RPi: Training vs. Live feature distributions. Benign (idle) data shifts heavily into the learned attack region.



(b) Intel PC 2: Training vs. Live feature distributions. Attack data diffuses away from the learned malicious profile.

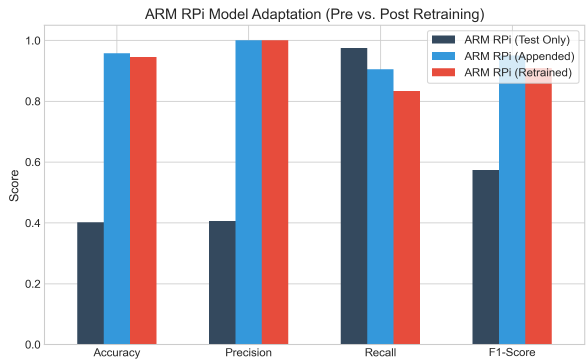
Figure 6.17: Scatter plots demonstrating the root cause of pre-adaptation failure: severe feature space distortion between isolated training environments and live deployment.

6.2.2 Efficient Calibration vs. Full Retraining

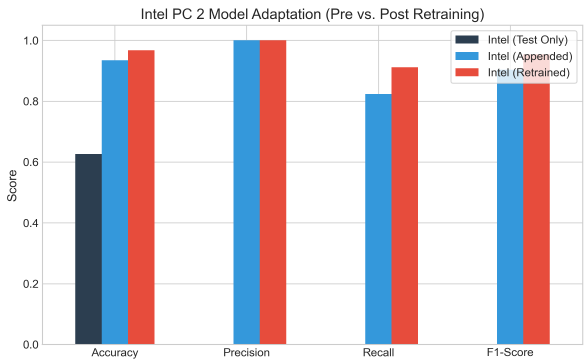
To mitigate this environmental mismatch, we employ an offline calibration phase utilizing Source-Free Domain Adaptation (SFDA) on a strictly limited subset of target data (referred to as the "Retrained" model). To benchmark this lightweight adaptation against a traditional, resource-intensive approach, we also evaluated an exhaustive retraining procedure that appends the entire target dataset to the source data (referred to as the "Appended" model).

As demonstrated in Figure 6.18, our lightweight SFDA adaptation achieves performance metrics that are highly competitive with—and in some cases superior to—the heavy full retraining, but at a fraction of the computational overhead. On the ARM RPi (Figure 6.18a), both the short adaptation and full appended retraining completely eliminate the pre-adaptation false positives. The heavy appended model exhibits a slight edge in recall, reducing false negatives to 4, compared to 5 in the lightweight retrained model.

Crucially, on the Intel workstation (Figure 6.18b), the lightweight adaptation actually outperforms the exhaustive approach. The SFDA adaptation successfully pulls the model out of its blind state, recovering recall to over 91% (31 true positives, 3 false negatives). The heavy appended model only recovers recall to approximately 82% (28 true positives, 6 false negatives). This suggests that the targeted, short-window adaptation is more effective at realigning the specific decision boundaries for transient execution noise without being overwhelmed by the bulk of the target's benign data.



(a) ARM RPi adaptation metrics.



(b) Intel PC 2 adaptation metrics.

Figure 6.18: Quantitative performance comparison highlighting that the lightweight SFDA adaptation (Retrained) achieves parity with, or exceeds, the exhaustive data approach (Appended).

6.2.3 Computational Overhead of Adaptation vs. Retraining

While Section 6.2.2 established the classification efficacy of our lightweight Source-Free Domain Adaptation (SFDA) relative to an exhaustive full retraining, it is equally important to quantify the respective computational costs of these two procedures. To demonstrate this, we benchmarked the execution time required for calibration on the ARM RPi target hardware.

The exhaustive "Full Retrain" baseline processes a merged dataset of 33,958 execution windows, requiring the neural feature extractors to be trained entirely from scratch. Conversely, our proposed "Online Adaptation" (SFDA) operates strictly on a localized, unlabeled update batch of 6,891 windows.

Table 6.4 details the comparative time overhead for the foundational architectures and our proposed hybrid framework.

For our proposed Hybrid DA-LSTM + XGBoost model, the exhaustive full retrain required over 7 minutes (434.51 seconds). In contrast, the online SFDA adaptation completed in roughly 1 minute (63.88 seconds)—yielding an 85.3% reduction in total execution time.

Table 6.4: Computational Time Comparison: Full Retrain vs. Online Adaptation (ARM - RPi)

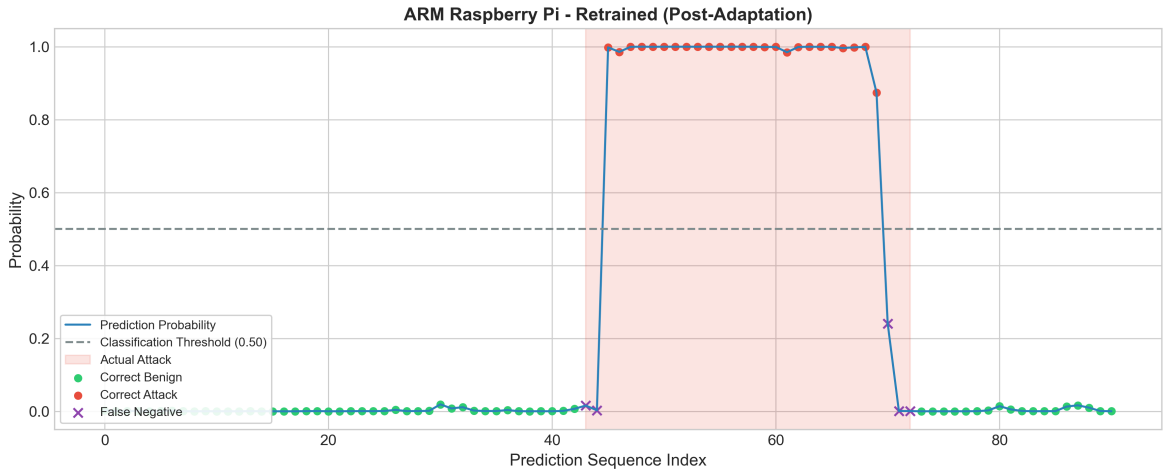
Model	Architecture	Full Retrain	Online Adaptation	Time Savings
LSTM	Standard SGD backprop	∼20.07 s	∼1.89 s	90.6%
XGBoost	Pure GBDT restart	∼1.81 s	∼0.23 s	87.3%
CAE	Autoencoder gate	∼9.02 s	∼1.39 s	84.6%
TCN-VAE	Temporal VAE	∼60.75 s	∼6.27 s	89.7%
DA	SHOT Encoder Adaptation	∼198.07 s	∼29.25 s	85.2%
Hybrid (Ours)	**SHOT + XGBoost Restart**	**∼434.51 s**	**∼63.88 s**	**85.3%**

This significant acceleration is achieved because the hybrid model's deep temporal embedding backbone is adapted using Source Hypothesis Transfer (SHOT) over a limited 15-epoch cycle. During this phase, the classifier and decoder heads are frozen, circumventing the need to retrain the heavy neural feature extractor from scratch on a massive, merged dataset. The remaining 11 to 12 seconds of the adaptation time are allocated to restarting the lightweight XGBoost classifier on the newly aligned feature representations. This empirical timing data firmly validates our SFDA approach as a highly efficient calibration mechanism, uniquely suited for rapid, on-site deployment without imposing severe computational burdens on the host environment.

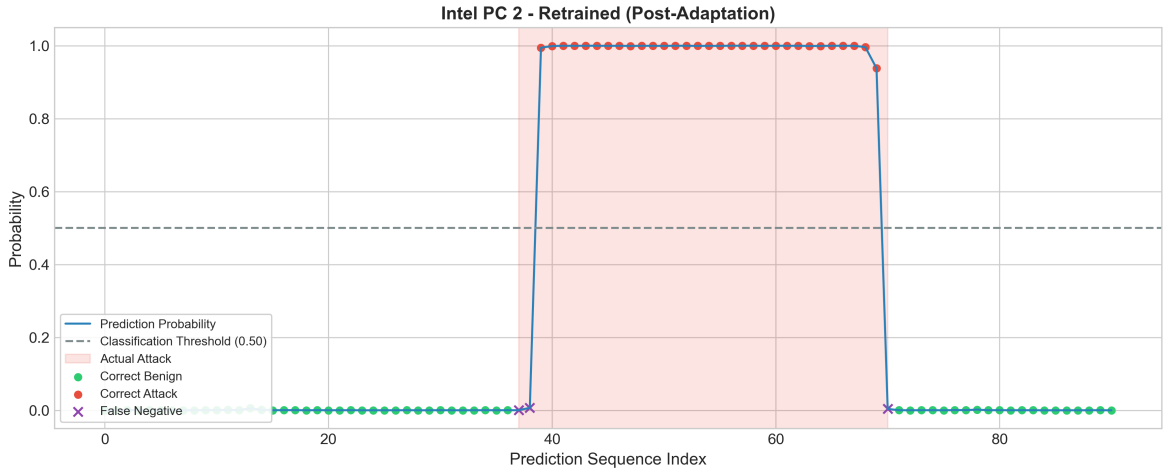
6.2.4 Stabilized Operational State

Deploying either post-calibration approach yields stable, accurate real-time tracking across both architectures, with the lightweight SFDA models establishing robust decision boundaries.

Figure 6.19 illustrates the live prediction sequences post-calibration for the proposed SFDA lightweight adaptation (Retrained). The ARM RPi model (Figure 6.19a) successfully maintains near-zero probabilities during benign execution and distinctly spikes to 1.0 during active attack blocks. Similarly, the previously blind Intel model (Figure 6.19b) decisively crosses the 0.50 threshold during malicious transient execution phases.



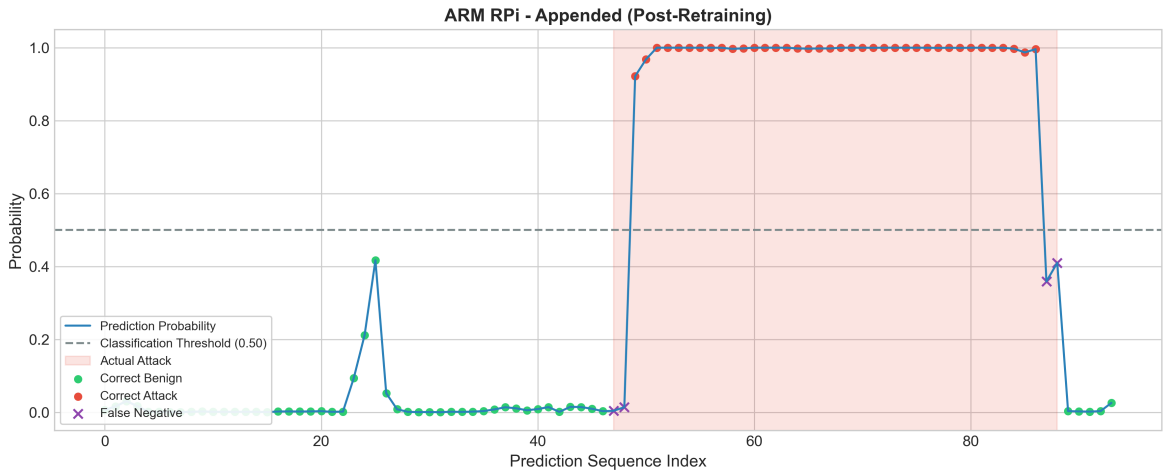
(a) ARM RPi (Retrained): Stabilized benign tracking and accurate attack detection.



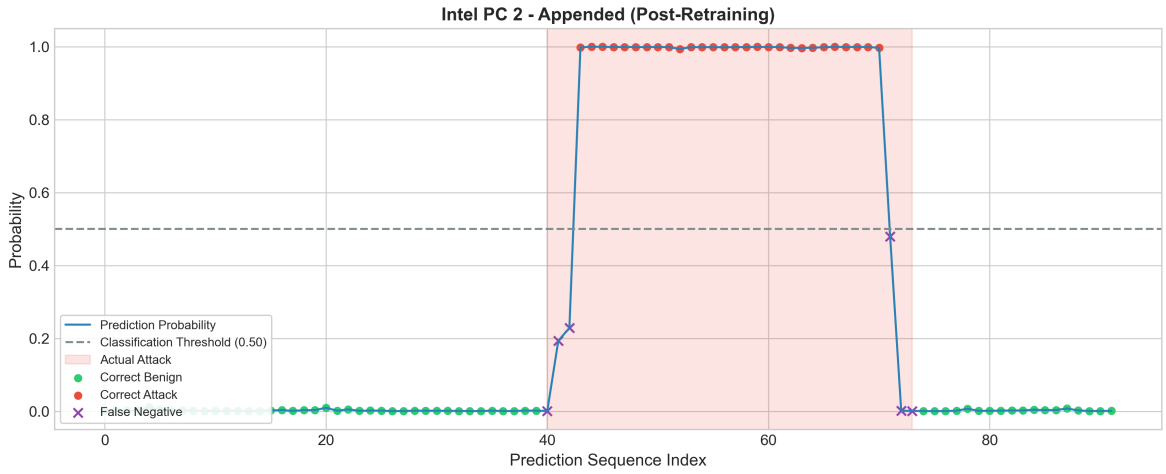
(b) Intel PC 2 (Retrained): Successfully recovered attack detection capability.

Figure 6.19: Live deployment timelines for the proposed lightweight adaptation (Retrained) models, showing successful alignment of prediction probabilities with actual attack windows.

For comparison, Figure 6.20 depicts the corresponding live timelines for the heavy full retraining baseline (Appended). While the appended model on the ARM target (Figure 6.20a) performs comparably to SFDA, its Intel counterpart (Figure 6.20b) experiences slightly more boundary hesitation at the onset and conclusion of the attack window, reflecting its lower overall recall score.



(a) ARM RPi (Appended): Heavy full retraining baseline performance.



(b) Intel PC 2 (Appended): Heavy full retraining baseline performance.

Figure 6.20: Live deployment timelines for the heavy full retraining (Appended) models, provided as a comparison baseline against the lightweight SFDA adaptation.

6.2.5 Real-Time Inference Latency and Client Overhead

A critical requirement for any microarchitectural monitor is that the detection overhead does not eclipse the execution duration of the attack itself. Because our pipeline utilizes a decoupled client-server architecture, the heavy lifting of the Hybrid DA-LSTM + XGBoost inference is offloaded from the host target.

Figure 6.21 details the inference latency distributions for both targets. The ARM RPi classification requests yield a median processing latency of roughly 22 milliseconds, while the Intel PC 2 maintains a median of approximately 24 milliseconds. Both metrics fall well within the 50-millisecond sampling interval established for the live telemetry stream. This confirms that the detection framework can continuously ingest and classify hardware performance counter data without inducing a processing bottleneck or dropping sequential telemetry frames.

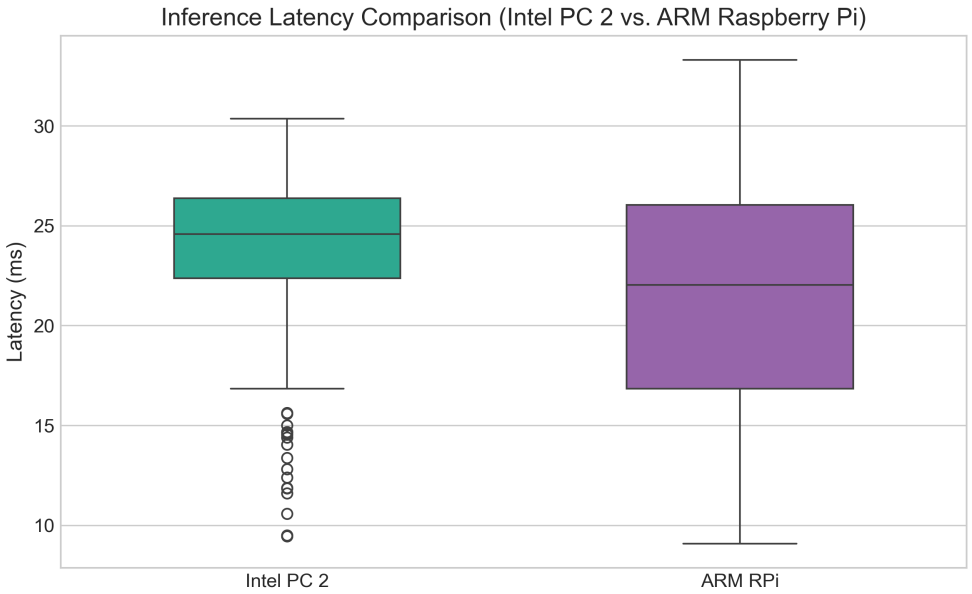


Figure 6.21: Inference latency distribution across the decoupled client-server pipeline, demonstrating real-time processing capabilities within the 50ms sampling window.

We also measured client-side resource usage during live data transmission to ensure the system is practical for real-world deployment. On the Intel_PC_2 workstation, average CPU

usage was 5.43% (peaking at 6.00%), and peak memory usage (VmHWM) was 54.70 MB. Network bandwidth was a steady 46.65 Kbps.

Testing on the Raspberry Pi showed even lower overhead: average CPU usage was 1.33% (peaking at 2.00%), and peak memory was 24.47 MB. Network transmission remained consistent at 43.87 Kbps.

These metrics (Table 6.5) confirm that capturing and transmitting hardware performance counters (HPCs) is lightweight. Our pipeline runs efficiently without causing severe resource contention on either high-performance workstations or constrained edge devices.

Table 6.5: Comparison of Client-Side Operational Overhead

Metric	**Intel x86 Workstation**	**ARM Raspberry Pi**
Average CPU Utilization	5.43%	1.33%
Peak CPU Utilization	6.00%	2.00%
Peak Memory (VmHWM)	54.70 MB	24.47 MB
Sustained Bandwidth	46.65 Kbps	43.87 Kbps

Chapter 7

Conclusions and Future Work

This final chapter concludes the dissertation by reflecting on the core findings of our research, discussing the limitations of the current system, and outlining directions for future work. The preceding chapters detailed the design, implementation, and evaluation of our framework for detecting microarchitectural attacks at runtime. Here, we summarize how the integration of hardware telemetry and adaptive machine learning successfully addresses the challenges of hardware differences and retraining overhead. We also acknowledge the practical hardware constraints that shape our current model and present opportunities to further advance this research.

7.1 Summary of Contributions

This dissertation presented a way to detect microarchitectural vulnerabilities, like Spectre, in real time using hardware performance metrics and adaptive machine learning. We turned theoretical anomaly detection into a practical, fast solution for real-world use. The core contributions include:

- **Analyzed Hardware Variations:** We systematically collected and analyzed Hard-

ware Performance Counter (HPC) data across Intel, ARM, and AMD processors. This created a strong baseline for spotting anomalies across different hardware brands.

- **Measured System Noise and Attacker Evasion:** We evaluated how detection signals degrade under heavy background workloads to identify the most noise-resistant metrics, ensuring the model remains robust against realistic interference and attacker evasion tactics.

- **Designed an Adaptive Machine Learning Model:** We built a new detection model that combines a DA-LSTM network to track time-based patterns with an XGBoost classifier. By integrating Source-Free Domain Adaptation (SFDA), we enabled the model to learn new hardware environments without needing expensive, labeled datasets.

- **Engineered a Live Client-Server Detection Pipeline:** We built a real-time system that separates the workload. A lightweight client collects data on the user's device, while a central server handles the heavy machine learning tasks.

- **Demonstrated Computational Efficiency and Practical Viability:** We proved that our SFDA updating method is just as accurate as traditional full-dataset retraining, but it reduces the computing time by about 85%. This makes continuous, real-time protection practical.

7.2 System Limitations

While our model is efficient, it faces specific hardware and environmental limits. The primary hardware bottleneck is that modern processors restrict us to monitoring a maximum of four performance counters at once. This forces our system to rely on broad, general metrics—such as overall cache misses—instead of tracking detailed, simultaneous microarchitectural events.

Because we are capped at four concurrent metrics, we lose vital context, which inherently limits how precise the model's decision boundaries can be.

Additionally, real-world computing environments introduce severe environmental noise. Normal background applications create constant fluctuations in the hardware data, which can severely lower detection accuracy if the model does not adapt. For example, a model trained on a balanced "Mixed" workload fails when tested against heavy Memory or CPU workloads, dropping to unadapted F1-scores of just 0.114 and 0.205.

Our adaptation method successfully recovers performance in many cases—such as boosting the Mixed-to-CPU transition from 0.205 up to a reliable 0.798. However, some transitions remain challenging. Adapting from a Mixed to a heavy Memory environment only recovers the F1-score to 0.285. This reveals a compounding problem: when extreme background noise distorts the system, the model needs more data points to separate the attack from benign operations. Because we are restricted to only four counters, the model lacks the necessary information to see through heavy memory interference. Ultimately, these metrics show that while adaptation is highly effective, the framework's reliability remains bounded by the combination of environmental noise and strict hardware limitations.

7.3 Future Work

Building on this work, there are several directions for future research to expand the framework's capabilities and deployment flexibility:

- **Testing on Secure Enclaves:** An important next step is to test the framework on advanced hardware modules and trusted execution environments. Specifically, we plan to evaluate how well the model detects attacks on systems using Intel Software Guard Extensions (SGX) and isolated secure enclaves.

- **Offloading to Dedicated External Hardware:** On the deployment front, we plan to explore offloading the server-side inference engine to a Field-Programmable Gate Array (FPGA) hub or a specialized USB coprocessor. This physical isolation would create a secure, out-of-band monitoring channel and completely remove the detection system's computational overhead from the host machine.

- **Overcoming the Counter Limit:** To solve the inherent four-counter hardware limit, we will explore HPC time-multiplexing. This technique rapidly switches which counters are being read, artificially expanding the number of concurrent features the model can monitor to refine its decision boundaries.

- **Unifying the Machine Learning Architecture:** To improve the model's structural cohesion, we aim to replace the XGBoost classifier with a fully differentiable neural network layer. This would integrate seamlessly with the DA-LSTM, allowing for end-to-end backpropagation and eliminating the disjointed training phases in our current hybrid setup.

Bibliography

[1] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[2] S. Chen, S. A. Billings, and P. M. Grant. Radial basis function networks for identification and control. *International Journal of Control*, 55(6):1091–1111, 1991.

[3] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[4] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM, 2016.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, volume 41, pages 559–570, 07 2013.

[7] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[8] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3*, pages 251–261. Springer, 2001.

[9] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. Domain-adversarial training of neural networks. *J. Mach. Learn. Res.*, 17(1):2096–2030, Jan. 2016.

[10] F. A. Gers, D. Eck, and J. Schmidhuber. Applying lstm to time series predictable through time-window approaches. In G. Dorffner, H. Bischof, and K. Hornik, editors, *Artificial Neural Networks — ICANN 2001*, volume 2130 of *Lecture Notes in Computer Science*, pages 669–676, Berlin, Heidelberg, 2001. Springer.

[11] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 279–299. Springer, 2016.

[12] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He. Deepfm: A factorization-machine based neural network for ctr prediction. *arXiv*, 2017.

[13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[15] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan. Supervised contrastive learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 18661–18673, 2020.

[16] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[17] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[18] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999.

[19] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[20] W. Kosasih, Y. Feng, C. Chuengsatiansup, Y. Yarom, and Z. Zhu. Sok: Can we really detect cache side-channel attacks by monitoring performance counters? In *Proceedings of the ACM on Web Conference 2024*, pages 172–185, 07 2024.

[21] J. Kotha and J.-L. Gaudiot. A data-driven approach using hardware performance counters to detect micro-architectural attacks. In *Proceedings of the Nineteenth IEEE International Workshop on Security, Trust, and Privacy for Software Applications (STPSA 2025) at COMPSAC 2025*, pages 2301–2306, Toronto, Canada, July 2025. IEEE.

[22] J. K. C. Kotha and J.-L. Gaudiot. Characterizing the variance envelope: A multi-dimensional analysis of spectre telemetry across architectures and workloads, 2026.

[23] E. C. Larson et al. Real-time edge processing detection of malicious attacks using machine learning and processor core events. In *2021 IEEE International Systems Conference (SysCon)*, pages 1–7. IEEE, 2021.

[24] C. Li and J. Gaudiot. Detecting spectre attacks using hardware performance counters. *IEEE Transactions on Computers*, 71(6):1320–1331, 2022.

[25] C. Li and J.-L. Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 588–597. IEEE, 2019.

[26] J. Liang, D. Hu, and J. Feng. Do we really need to access the source data? Source hypothesis transfer for unsupervised domain adaptation. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, pages 6028–6039. PMLR, 2020.

[27] B. Lindemann, B. Maschler, N. Sahlab, and M. Weyrich. A survey on anomaly detection for technical systems using lstm networks. *Computers in Industry*, 131:103498, 2021.

[28] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[29] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[30] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[31] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.

[32] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[33] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan 2005*, volume 13, 2005.

[34] C. E. Rasmussen. Gaussian processes in machine learning. *Advances in Neural Information Processing Systems*, 16:337–344, 2004.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[36] M. Sugiyama, M. Krauledat, and K.-R. Müller. Covariate shift adaptation by importance weighted cross-validation. *Journal of Machine Learning Research*, 8(35):985–1005, 2007.

[37] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses (RAID)*, pages 109–129. Springer, 2014.

[38] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[39] V. N. Vapnik. *Statistical Learning Theory*. Wiley, New York, NY, USA, September 1998.

[40] C. Windeck. Cpu-sicherheitslücken spectre-ng: Updates rollen an, May 2018.

[41] M. Yan, R. Bhaskar, C. W. Fletcher, and J. Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 347–360. IEEE, 2017.

[42] 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, 2014.

[43] X. Zhang, Y. Xiao, and Y. Zhang. CloudRadar: Real-time detection of cache-based side-channel attacks in clouds. In *Proceedings of the 2018 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 118–140, 2018.

[44] T. Zhou and Y. Makris. Hardware-based detection of Spectre attacks: A machine learning approach. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–11. IEEE, 2021.

[45] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

Appendix A

Training Parameters

This appendix provides the comprehensive hyperparameter configurations utilized for training, evaluation, and adaptation across all evaluated models.

A.1 Baseline and Hybrid Model Configurations

Table A.1: Intra-Domain LSTM Model Hyperparameters

Hyperparameter	**Value / Setting**
LSTM Hidden Units	64
Dense Layer Units	32
Learning Rate	1.0×10^{-3}
Batch Size	512
Training Epochs	30
Early Stopping Patience	7 epochs
Anomaly Threshold	0.5
Optimizer	Adam
Scheduler Factor / Patience	0.5 / 3 epochs

Table A.2: Intra-Domain XGBoost Model Hyperparameters

Hyperparameter	**Value / Setting**
Number of Trees	500
Maximum Tree Depth	6
Learning Rate	0.05
Subsample Ratio	0.8
Colsample by Tree	0.8
Minimum Child Weight	5
Gamma (γ)	0.1
L1 Regularization (α)	0.1
L2 Regularization (λ)	1.0
Early Stopping Rounds	20
Tree Construction Algorithm	Hist (`hist`)
Loss Function	Binary Log Loss
Class Weight Ratio	Scale Pos Weight

Table A.3: Intra-Domain TCN-VAE Model Hyperparameters

Hyperparameter	**Value / Setting**
TCN Channels	32
TCN Kernel Size	3
TCN Dilation Rates	(1, 2, 4)
Latent Dimension	16
KL Divergence Weight (β)	0.5
Learning Rate	2.0×10^{-3}
Batch Size	128
Training Epochs	25
MAD Multiplier (K)	4.0
Anomaly Threshold	0.5
Optimizer	Adam
Activation Function	GELU

Table A.4: SFDA-LSTM Model Hyperparameters

Hyperparameter / Setting	**Value / Setting**
LSTM Hidden Units	128 (Bidirectional)
LSTM Layers	2
Bottleneck Dimension	256
Classifier Weight Regularization	Weight Normalization (`weight_norm`)
Dropout Rate	0.3
Supervised Epochs	25
Supervised Learning Rate	0.001 (10^{-3})
Optimizer	AdamW (weight_decay = 10^{-4})
Learning Rate Scheduler	Cosine Annealing
Loss Function	Cross Entropy (`label_smoothing` = 0.1)
Batch Size	256
Source-Free Domain Adaptation (SHOT)	
SFDA Adaptation Epochs	15
SFDA Learning Rate	0.0003 (3×10^{-4})
Entropy Minimization Weight (θ_{ent})	1.0
Diversity Regularization Weight (θ_{div})	1.0
Pseudo-Label Loss Weight (θ_{cls})	0.3
Pseudo-Label Refresh Interval	Every 1 epoch
Adapted Parameters	Feature Extractor (Classifier Head Frozen)

Table A.5: Hybrid LSTM-XGBoost Model Hyperparameters

Hyperparameter / Setting	**Value / Setting**
Convolutional Channels	64 (Kernels 3, 5, 7)
LSTM Hidden Units	128 (Bidirectional)
LSTM Layers	2
Embedding Dimension	512
Projection / Bottleneck Dimension	128
Dropout Rate	0.3
Supervised Training Epochs	30
Supervised Learning Rate	0.001 (10^{-3})
Batch Size	256
Optimizer	AdamW (weight_decay = 10^{-4})
Learning Rate Scheduler	Cosine Annealing
Label Smoothing	0.1
Supervised Contrastive Weight (λ_{con})	0.5
Reconstruction Loss Weight (λ_{rec})	0.3
Temperature (SupCon)	0.1
Anomaly Threshold	0.5
Source-Free Domain Adaptation (SFDA / SHOT)	
SFDA Adaptation Epochs	15
SFDA Learning Rate	0.0003 (3×10^{-4})
Entropy Minimization Weight (θ_{ent})	1.0
Diversity Regularization Weight (θ_{div})	1.0
Pseudo-Label Loss Weight (θ_{cls})	0.3
Reconstruction Anchor Loss Weight (θ_{rec})	0.3
Pseudo-Label Refresh Interval	Every 1 epoch
Adapted Parameters	Encoder / Feature Extractor (Heads Frozen)
XGBoost Classifier Settings	
Number of Estimators (N_{trees})	500
Max Depth	6
Learning Rate	0.05
Subsample / Colsample	0.8 / 0.8
Min Child Weight	5
Gamma (γ)	0.1
Regularization (α/λ)	0.1 / 1.0
Tree Method	Hist

A.2 XGBoost Statistics Definitions

Before classification, each sliding window of raw HPC readings is collapsed into a fixed-length statistical feature vector. For a window of length T and a given raw counter $x_1, x_2, \ldots, x_T$, we compute fourteen descriptive statistics that summarise the HPC's central tendency, dispersion, distribution shape, and temporal behaviour within the window.

- **Mean:** The average value of the counter over the window, capturing its overall level of activity:

$$\bar{x} = \frac{1}{T} \sum_{i=1}^{T} x_i \tag{A.1}$$

- **Standard Deviation:** Measures the dispersion of the counter around its mean, indicating how volatile the signal is within the window:

$$\sigma = \sqrt{\frac{1}{T} \sum_{i=1}^{T} (x_i - \bar{x})^2} \tag{A.2}$$

- **Minimum:** The smallest observed value in the window:

$$x_{\min} = \min_i \, x_i \tag{A.3}$$

- **Maximum:** The largest observed value in the window:

$$x_{\max} = \max_i \, x_i \tag{A.4}$$

- **Median:** The middle value of the sorted window, robust to outliers that the mean is

sensitive to:

$$\tilde{x} = \text{median}(x_1, \ldots, x_T) \tag{A.5}$$

- **Range:** The spread between the extreme values, a simple measure of overall variability:

$$\text{Range} = x_{\max} - x_{\min} \tag{A.6}$$

- **Slope:** The least-squares linear trend of the counter over time, capturing whether the signal is rising or falling within the window:

$$\text{Slope} = \frac{\sum_{i=1}^{T}(t_i - \bar{t})(x_i - \bar{x})}{\sum_{i=1}^{T}(t_i - \bar{t})^2} \tag{A.7}$$

where t_i indexes the timestep within the window.

- **Skewness:** Quantifies the asymmetry of the counter's distribution within the window; positive skew indicates a longer right tail:

$$\text{Skew} = \frac{\frac{1}{T}\sum_{i=1}^{T}(x_i - \bar{x})^3}{\sigma^3} \tag{A.8}$$

- **Kurtosis:** Measures the "tailedness" of the distribution relative to a normal distribution (excess/Fisher kurtosis, so a normal distribution scores 0):

$$\text{Kurt} = \frac{\frac{1}{T}\sum_{i=1}^{T}(x_i - \bar{x})^4}{\sigma^4} - 3 \tag{A.9}$$

- **25th Percentile (Q_1):** The value below which 25% of the window's observations fall.

- **75th Percentile (Q_3):** The value below which 75% of the window's observations fall.
- **Interquartile Range:** The spread of the middle 50% of the data, a dispersion measure robust to outliers:

$$\text{IQR} = Q_3 - Q_1 \tag{A.10}$$

- **Energy:** The mean squared value of the counter, reflecting the overall magnitude/intensity of activity over the window:

$$\text{Energy} = \frac{1}{T}\sum_{i=1}^{T} x_i^2 \tag{A.11}$$

- **Zero-Crossing Rate:** The rate at which the mean-centred signal changes sign, capturing how erratically the counter oscillates within the window:

$$\text{ZCR} = \frac{1}{2(T-1)}\sum_{i=1}^{T-1} |\text{sgn}(x_{i+1} - \bar{x}) - \text{sgn}(x_i - \bar{x})| \tag{A.12}$$

Applying these fourteen statistics to each of the F raw counters produces $14F$ engineered columns. In addition, three domain-specific *ratio features* are appended, each computed from the window-mean of a numerator and denominator counter:

$$\text{Ratio} = \frac{\overline{x}_{\text{numerator}}}{\overline{x}_{\text{denominator}} + \epsilon} \tag{A.13}$$

where $\epsilon = 10^{-8}$ prevents division by zero. The three ratios used are the LLC miss rate, cache miss rate, and branch miss rate, computed respectively from LLC Load Misses / LLC

Loads, Cache Misses / Cache References, and Branch Misses / Branch Instructions.

Appendix B

Hardware Performance Counter Event Mappings

This appendix details the specific Hardware Performance Counter (HPC) metrics collected across the experimental hardware matrix. Due to semantic discrepancies in event definitions across Instruction Set Architectures (ISAs) and distinct cache topologies, telemetry collection was distributed across sequential execution batches.

Table B.1 details the architectural event mappings for the Intel x86_64 testbeds. Table B.2 outlines the corresponding telemetry metrics for the ARM Cortex-A76 and AMD Jaguar profiles.

Table B.1: HPC Event Mappings for Intel Architectures

Device / CPU	Batch	Logical Metric	Raw `perf` Event Name
Intel 1 (Core i7-3537U)	Default	Cache References	`cache-references`
		Cache Misses	`cache-misses`
		Branch Instructions	`branch-instructions`
		Branch Misses	`branch-misses`
	Batch 1	Offcore Demand Data Read LLC Miss DRAM	`offcore_response. demand_data_rd. llc_miss.dram`
		Offcore Demand Data Read LLC Hit Any	`offcore_response. demand_data_rd. llc_hit.any_response`
		Branch Instruction Executed (All)	`br_inst_exec. all_branches`
		Branch Instruction Retired (All)	`br_inst_retired. all_branches`
	Batch 2	L1 DCache Loads	`L1-dcache-loads`
		L1 DCache Load Misses	`L1-dcache-load-misses`
		Branch Instruction Executed (Conditional)	`br_inst_exec. all_conditional`
		Branch Instruction Retired (Conditional)	`br_inst_retired. conditional`
Intel 2 (Core i5-7200U)	Default	Cache References	`cache-references`
		Cache Misses	`cache-misses`
		Branch Instructions	`branch-instructions`
		Branch Misses	`branch-misses`
	Batch 1	LLC Loads	`LLC-loads`
		LLC Load Misses	`LLC-load-misses`
		Branch Instruction Retired (All)	`br_inst_retired. all_branches`
		Branch Mispredictions Retired (All)	`br_misp_retired. all_branches`
	Batch 2	L1 DCache Loads	`L1-dcache-loads`
		L1 DCache Load Misses	`L1-dcache-load-misses`
		Branch Instruction Retired (Conditional)	`br_inst_retired. conditional`
		Branch Mispredictions Retired (Conditional)	`br_misp_retired. conditional`

Table B.2: HPC Event Mappings for ARM and AMD Architectures

Device / CPU	Batch	Logical Metric	Raw `perf` Event Name
ARM RPi	Default	Cache References	`cache-references`
(Cortex-A76)		Cache Misses	`cache-misses`
		Branch Loads	`branch-loads`
		Branch Misses	`branch-misses`
	Batch 1	LLC Loads	`LLC-loads`
		LLC Load Misses	`LLC-load-misses`
		Branch Predictions	`br_pred`
		Branch Mispredictions	`br_mis_pred`
	Batch 2	L1D Cache Read	`l1d_cache_rd`
		L1D Cache Refill Read	`l1d_cache_refill_rd`
		Branch Immediate Speculation	`br_immed_spec`
		Branch Indirect Speculation	`br_indirect_spec`
AMD Server	Default	L1 DCache Accesses	`r040`
(A4-5000)		L1 DCache Misses	`r041`
		Branch Instructions	`branch-instructions`
		Branch Misses	`branch-misses`
	Batch 1	L2 Cache Accesses	`r269`
		L2 Cache Hits	`r270`
		L2 Fills from L3/Memory	`r140`
		L2 Cache Misses	`r141`
	Batch 2	L2 Cache Operations	`r268`
		L2 Fill from DRAM	`r260`
		Branch Loads	`branch-loads`
		Branch Load Misses	`branch-load-misses`