International Journal of
Advanced Science Computing and Engineering

Journal Homepage www.ijasce.org/index.php/ijasce

A Comprehensive Survey of Transient Execution Vulnerabilities: from Meltdown and Specter to Modern Variants

Faisal Mushtaq [a,*]

[a] *Department of Computer Science and Engineering, North campus University of Kashmir, India*
Corresponding author: [*]bhattfaisalbhatt36@gmail.com

***Abstract*—Meltdown and Specter exploit hardware vulnerabilities in modern processors. These vulnerabilities allow programs to steal data currently being processed on the computer. Although programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Specter to access secrets stored in the memory of other running programs. This analysis covers distinct attack vectors across multiple processor architectures and provides a systematic taxonomy based on exploitation mechanisms, targeted microarchitectural components, and threat models. Although implementation-specific vulnerabilities like Meltdown have been effectively mitigated, architectural issues exemplified by Specter continue to challenge the security community. This paper also includes explainable machine learning methods for detecting Meltdown and Specter. The dataset utilized was synthetic and does not represent actual attacks. Genuine attackers may evade detection, as we did not test against real malicious activity. The detector requires further assessment on authentic systems to evaluate its operational overhead and reliability. Future research should concentrate on: (1) designing processors with speculation techniques that leave no detectable traces; (2) developing tools for the preemptive identification of vulnerabilities; (3) establishing formal methodologies to validate security guarantees; and (4) designing architectures, such as RISC-V, with integrated security features.**

***Keywords*—Transient execution; speculative execution; meltdown; specter; microarchitectural attacks; side-channel analysis; hardware security; cache timing; branch prediction.**

Manuscript received 15 Dec. 2025; revised 8 Feb. 2026; accepted 14 Apr. 2026. Date of publication 31 Aug. 2026.

I. Introduction

In January 2018, security researchers revealed two critical vulnerabilities that had existed in computer processors for over twenty years: Meltdown [1] and Specter [2]. These discoveries were shocking because they affected billions of devices worldwide and challenged basic assumptions about computer security. For the first time, researchers showed that performance optimizations built into processors could be exploited to bypass fundamental security protections.

Modern [3] processors use speculative execution to improve performance, resource utilization, and user experience. Speculation mechanisms may use various predictors to anticipate future program execution and improve performance by executing instructions earlier than their program order. Although these predictors are designed to be highly accurate, incorrect predictions can occur, resulting in mis-speculation, where a processor executes instructions based on a prediction and later squashes them to return to correct program execution. An attacker can potentially exploit such mis-speculation to reveal sensitive data. According to the study in [2], the Meltdown attack can dump kernel memory at 503 KB/s. Therefore, detecting Specter and Meltdown attacks is critical to enable trustworthy computing.

The evolution of the threat landscape began with the initial Meltdown and Specter disclosures, which were merely the beginning. Subsequent research revealed a vast attack surface, with dozens of variants discovered between 2018 and 2024. These include Foreshadow [4], which compromised Intel SGX enclaves; MDS attacks [5-7] like RIDL, Fallout, and ZombieLoad that leak data through internal CPU buffers; Spectre-BTI variants [8-10] that poison branch prediction structures; and Load Value Injection [11] that injects attacker-controlled values into victim computations.

Each new variant exposed previously unknown exploitation vectors, demonstrating that transient execution vulnerabilities represent a systemic architectural challenge rather than isolated implementation bugs. This evolution necessitated increasingly sophisticated defenses, from simple software patches to microcode updates, architectural

modifications, and fundamental rethinking of secure processor design principles.

II. Materials and Method

Modern high-performance processors achieve instruction-level parallelism through aggressive speculation [3]. When the processor encounters a conditional branch, it cannot afford to stall while waiting for the condition to be evaluated; this would create pipeline bubbles that drastically reduce throughput. Instead, the processor predicts the likely outcome based on historical behavior and speculatively executes instructions along the predicted path.

1) Branch Prediction: Processors maintain sophisticated prediction structures including Pattern History Tables (PHT) that record branch directions, Branch Target Buffers (BTB) that predict indirect branch targets, and Return Stack Buffers (RSB) that track function return addresses [12]. Modern predictors achieve accuracy exceeding 95%, making speculation highly effective for performance.

2) Out-of-Order Execution: Beyond branch prediction, processors execute instructions opportunistically when their operands become available, independent of program order [13]. This requires complex register renaming, reorder buffers to track instruction dependencies, and commit logic to ensure architectural correctness.

3) Memory Speculation: Load operations may execute before all preceding store addresses are known, requiring memory disambiguation logic to detect and recover from conflicts [14]. This speculation extends even to memory access permissions, as revealed by Meltdown.

A. Microarchitectural State

While speculative execution is architecturally invisible, incorrect speculation is rolled back without affecting program semantics; it leaves observable traces in microarchitectural structures:

1) Cache Hierarchy: Modern processors employ multi-level caches (L1, L2, L3) with different access latencies. Cache state modifications during speculation persist even after rollback, creating the primary covert channel exploited by transient execution attacks [15].

2) Translation Lookaside Buffers (TLBs): These cache virtual-to-physical address translations. TLB entries populated during speculation remain, leaking address information [16].

3) Store Buffers: Intermediate structures hold store data before writing to cache. Some attacks exploit store buffer forwarding behavior [17].

4) Prefetchers: Hardware prefetchers predict future memory accesses and bring data into cache proactively. Their behavior can reveal access patterns [18].

B. Meltdown and Specter

Although Meltdown and Specter are very new in security research, they are a small advancement over seventy years of hardware security research on fault injection, side channels, and covert channels [19]

C. Attacks using physical side channels

Physical side-channel assaults, which leverage indirect physical information to extract secrets, were the main focus of early side-channel research. Because side channels were physical in nature, early attacks were thought to be less dangerous and worth mitigating only on security-critical components like cryptographic hardware, since they require physical access or proximity to the machine and are harder to execute. The most prevalent types of physical data collected in these assaults, which are still relevant today, are:

1) Timing analysis: It evaluates how long operations (such as encryption and decryption) take to complete for various inputs and uses timing variances to deduce secret information. This method is often used alongside other physical side-channel attacks [19].

2) Power analysis: evaluates how much power is used for various inputs and outputs throughout processes (such as encryption and decryption) and deduces hidden information from changes in power consumption. [19].

3) Electromagnetic analysis: uses fluctuations in electromagnetic signals to deduce secret information by measuring electromagnetic waves created by current flowing across the device [19].

*4) Fault analysis***:** Involves physically tampering with voltage levels, clock signals, or other hardware components to cause a fault in the device (for example, disrupting a few memory or register bits). It then uses differences in the output of malfunctioning operations to deduce secret information. This combines two methods: it begins with a physical fault-injection attack, which violates integrity, and then leverages the attack's successful outcomes as data for a physical side-channel attack, which violates confidentiality [19].

Cache Timing Channels. Cache timing attacks exploit the dramatic timing difference between cache hits (≈4 nanoseconds) and misses (≈100 nanoseconds) to infer what data has been accessed [20]. Several techniques have been developed:

FLUSH+RELOAD [21]: This technique requires shared memory between the attacker and the victim:

1. *Flush*: Attacker uses the clflush instruction to evict the target address from all cache levels
2. *Victim Execution*: Target process potentially accesses the address
3. *Reload*: Attacker measures access time—fast indicates the victim accessed (cache hit), slow indicates no access (miss)

PRIME+PROBE [22]: Works without shared memory by exploiting cache set contention:

1. *Prime*: Attacker fills target cache sets with own data
2. *Victim Execution*: Victim's accesses may evict attacker's data
3. *Probe*: Attacker measures which sets were accessed by reload timing

FLUSH+FLUSH [23]: A stealthier variant that times flush operations themselves rather than memory accesses, reducing detectability.

EVICT+RELOAD [24]: Uses memory access patterns to evict rather than flush instructions, avoiding detection mechanisms that monitor flush instruction usage.

Timing Measurement. High-resolution timing is essential for side-channel attacks. The x86 RDTSC instruction provides cycle-accurate timestamps, while ARM offers similar functionality through PMCCNTR. Even when these are restricted, attackers can construct timing sources from performance counters, thread scheduling behavior, or networked timing channels [25].

D. Historical Context: Pre-2018 Microarchitectural Attacks

Cache timing attacks predate transient execution vulnerabilities by over a decade. Early work demonstrated attacks on cryptographic implementations:

1) AES Cache Attacks [22]: Osvik et al. [22] showed that cache timing could recover AES keys by exploiting data-dependent table lookups in common implementations.

2) Port contention: Measuring execution unit occupancy [26].

3) Last-Level Cache Attacks [27]: Irazoqui et al. [27] demonstrated cross-core cache attacks in cloud environments, threatening isolation in multi-tenant systems.

4) Rowhammer [28]: Kim et al. [28] showed that repeated access to DRAM rows could induce bit flips in adjacent rows, representing a different class of hardware vulnerability.

These attacks targeted software implementations or peripheral hardware components. Meltdown and Specter represented a fundamental escalation.

E. Threat Models

Transient execution attacks operate under various threat models:

1) Same-Process Attacks: Attacker code within a process leaks secrets from the same process (e.g., sandboxed JavaScript attacking the browser process).

2) Cross-Process Attacks: An unprivileged attacker process targets another user process or system services.

3) User-to-Kernel Attacks: User-space code reads kernel memory (Meltdown's primary threat).

4) Cross-VM Attacks: In cloud environments, an attacker VM targets a co-resident victim VM.

5) Enclave Attacks: Code outside secure enclaves (Intel SGX, ARM TrustZone) extracts enclave secrets.

6) Remote Attacks: Network-based attackers exploit timing differences observable across the network.

III. Results and Discussion

This paper proposes a systematic taxonomy that classifies transient execution attacks along three primary dimensions: exploitation mechanism, targeted microarchitectural component, and security boundary crossed. Contemporary nomenclature describes some intriguing side effects of general-purpose, high-performance processor architecture since the 1960s, including transient execution, transient instructions, and transient microarchitectural state. Speculative and out-of-order execution drive transient execution; however, the impact is exacerbated by the interplay of instruction-level parallelism, simultaneous multithreading, multilayer memory caches, and prefetching [19].

A. Classification by Exploitation Mechanism

1) Exception-Based Transient Execution (Meltdown-Type): These attacks trigger exceptions (page faults, access violations) that architecturally prevent operation completion, but exploit the window between operation initiation and exception handling where data transiently enters microarchitectural state.

Key Examples:

- Meltdown [1]: Exploits race condition in permission checking
- Foreshadow [4]: Exploits L1 Terminal Fault to read SGX enclave data
- LazyFP [29]: Exploits lazy FPU state restoration

2) Misprediction-Based Transient Execution (Spectre-Type): These attacks manipulate prediction structures (branch predictors, indirect branch targets) to cause speculative execution of gadget code that leaks secrets.

Key Examples:

- Spectre-PHT (v1) [2]: Poisons Pattern H.
- istory Table for bounds check bypass.
- Spectre-BTB (v2) [2]: Poisons Branch Target Buffer for indirect branch redirection.
- Spectre-RSB [30]: Manipulates Return Stack Buffer.
- ret2spec [31]: Exploits return instruction speculation.

3) Microarchitectural Data Sampling (MDS): These attacks leak data from internal CPU buffers (line fill buffers, load ports, store buffers) that temporarily hold data during normal operation.

Key Examples:

- RIDL [5]: Reads data from line fill buffers
- Fallout [6]: Exploits store buffer forwarding
- ZombieLoad [7]: Samples data from various internal buffers
- TAA [32]: Transaction Asynchronous Abort exploits TSX

4) Speculative Store Bypass: These attacks exploit speculative memory disambiguation where loads execute before preceding stores with unknown addresses.

Key Examples:

- Specter-v4 (SSB) [33]: Loads bypass stores speculatively
- SpectreRewind [34]: Exploits rollback of speculative stores

B. Classification by Microarchitectural Component

The table "Vulnerability Classification by Component" outlines how specific hardware features in modern processors are exploited by various microarchitectural attacks and identifies which chipmakers are affected. The most widespread vulnerabilities stem from core performance components like the L1 Data Cache and Branch Predictors (PHT/BTB). Because these components are fundamental to how modern CPUs predict and stage data, they are targeted by

high-profile, cross-platform exploits such as Meltdown, Spectre variants, Foreshadow, and BranchScope, which collectively impact major architectures including Intel, AMD, ARM, and IBM. Additionally, the Translation Lookaside Buffer (TLB) represents another shared vulnerability point, with the TLBleed attack affecting both Intel and AMD processors.

TABLE I
VULNERABILITY CLASSIFICATION BY COMPONENT

Component	Attacks	Architectures Affected
L1 Data Cache	Meltdown, Specter variants, Foreshadow	Intel, AMD, ARM
Branch Predictors (PHT/BTB)	Spectre-PHT, Spectre-BTB, BranchScope	Intel, AMD, ARM, IBM
Store Buffers	Fallout, Spectre-v4	Intel
Line Fill Buffers	RIDL, ZombieLoad	Intel
Load Ports	RIDL	Intel
TLB	TLBleed	Intel, AMD
TSX Structures	TAA	Intel

Conversely, the table highlights a significant cluster of vulnerabilities specific to Intel architectures, largely stemming from its internal data buffers and proprietary structural extensions. Hardware components such as Store Buffers, Line Fill Buffers, Load Ports, and TSX Structures have been successfully targeted by a distinct set of exploits, including Fallout, Specter-v4, RIDL, ZombieLoad, and TAA. This distinction illustrates that while foundational processing features like caching and branch prediction create industry-wide security challenges, Intel's specific microarchitectural implementations for data handling and transactional memory have exposed its processors to a highly specialized category of targeted attacks.

C. Classification by Security Boundary

1) Privilege Boundaries: Attacks crossing CPU privilege rings (user-to-kernel): Meltdown, Foreshadow-OS.

2) Process Isolation: Attacks between user processes: Specter variants, NetSpectre.

3) Virtual Machine Isolation: Cross-VM attacks in cloud environments: Foreshadow-VMM, L1TF.

4) Enclave Isolation: Attacks on trusted execution environments: Foreshadow-SGX, LVI.

5) Cryptographic Boundaries: Extracting cryptographic keys: PortSmash, NetSpectre against crypto implementations.

D. Meltdown: Breaking Kernel Isolation

1) Technical Mechanism: Meltdown (CVE-2017-5754) exploits a race condition in Intel's out-of-order execution implementation [1]. When user code attempts to access kernel memory:

1. The load instruction begins execution before privilege checking completes
2. Data is fetched from memory and enters the L1 cache
3. A subsequent instruction uses this data to access a probe array: probe [data * 4096]
4. This access brings probe [data * 4096] into cache
5. A page fault exception eventually fires, but cache modifications persist
6. Attacker uses FLUSH+RELOAD to determine which probe array element is cached, revealing the kernel byte value

Attack Code Pattern:

```
char probe[256 * 4096];
// Flush probe array from cache
for (i = 0; i < 256; i++)
clflush(&probe[i * 4096]);

// Attempt kernel read (will fault)
char value = *(char*)kernel_address;

// Encode in cache before exception
char dummy = probe[value * 4096];

// Exception handler catches fault
// Measure probe array to extract value
```

2) Affected Processors: Lipp et al. [1] demonstrated Meltdown across Intel processors spanning 2008-2017:

- Nehalem (2008)
- Westmere (2010)
- Sandy Bridge (2011)
- Ivy Bridge (2012)
- Haswell (2013)
- Broadwell (2014)
- Skylake (2015)
- Kaby Lake (2016)
- Coffee Lake (2017)

Architecture Specificity: AMD and ARM processors were generally not vulnerable because their out-of-order execution designs enforce privilege checks before data propagation [35].

3) Performance Metrics: The original Meltdown paper reported:

- Extraction Rate: 120 KB/s
- Accuracy (cached): 99.97%
- Accuracy (uncached): ~95%
- Per-byte Overhead: ~530 CPU cycles

These metrics demonstrated that Meltdown enabled practical, high-bandwidth kernel memory extraction.

E. Foreshadow: Attacking Intel SGX

1) L1 Terminal Fault Vulnerability: Foreshadow (L1TF, CVE-2018-3615/3620/3646) extended Meltdown's principles to break Intel SGX enclave isolation [4]. SGX was designed to protect code and data even from privileged software, creating a hardware-enforced trusted execution environment. The attack exploits L1 Terminal Fault conditions:

- Attacker triggers a page fault on enclave memory
- During transient execution before fault handling, data is read from L1 cache
- If enclave data is present in L1, it leaks despite page fault

- Cache timing reveals the secret data

Three Variants:

- Foreshadow-SGX: Breaks SGX enclave isolation
- Foreshadow-OS: Leaks kernel memory (similar to Meltdown)
- Foreshadow-VMM: Cross-VM attacks in virtualized environments

2) Impact on Cloud Security

Foreshadow-VMM posed severe threats to cloud infrastructure [36]:

- Guest VMs could read hypervisor memory
- Cross-VM attacks between co-resident tenants
- SGX enclaves, marketed as secure even in untrusted clouds, were compromised

Cloud providers temporarily disabled hyper-threading and implemented emergency patches [37].

3) Other Exception-Based Variants

- LazyFP [29]: Exploits lazy floating-point state restoration on context switches to leak FPU registers across processes.
- Meltdown-BR [38]: Extends Meltdown to bound range exceeded exceptions on Intel MPX.
- Meltdown-PK [38]: Exploits protection key violations on Intel PKU.
- Meltdown-US [38]: User/supervisor bit bypass variant.

F. Specter and Branch Prediction Attacks

Specter Variant 1 (Bounds Check Bypass) is a microarchitectural security vulnerability that exploits speculative execution by manipulating a processor's conditional branch predictor, specifically the Pattern History Table (PHT). To execute the attack, a malicious actor deliberately mispredicts the branch predictor to make it temporarily believe an out-of-bounds memory access is valid. When the processor encounters a bounds check (such as an if statement verifying an array index), it speculatively executes the instructions past the check using the out-of-bounds index before realizing the prediction was wrong. Although the processor eventually discards the execution and restores its state, the speculatively accessed secret data is temporarily loaded into the CPU cache, allowing the attacker to extract it using cache side-channel techniques like Flush+Reload.

1) Attack Mechanism: Spectre-PHT (CVE-2017-5753) exploits branch prediction to bypass bounds checking [2]. Consider typical array access code:

```
if (index < array_size) {
value = array[index];
temp = probe[value * 4096];
}
```

Attack Sequence:

1. Training: Call repeatedly with valid indices (0, 1, 2, ...) to train PHT to predict branch taken
2. Exploitation: Call with malicious index pointing to secret memory
3. Speculation: Branch predictor, trained to expect bounds check success, speculatively executes array access
4. Leakage: Speculative access brings probe[secret * 4096] into cache
5. Recovery: FLUSH+RELOAD on probe array reveals secret

2) Cross-Architecture Impact

Unlike Meltdown's Intel-specificity, Spectre-PHT affects all processors with branch prediction [39]:

- Intel: All modern processors
- AMD: Ryzen and earlier confirmed vulnerable
- ARM: Cortex-A57, A72, A73 vulnerable
- IBM POWER: POWER8 and POWER9 affected

G. Motivation for Machine Learning–Based Detection

Traditional defenses against Specter and Meltdown primarily rely on software mitigations, microcode updates, and architectural changes. While effective, these approaches often introduce significant performance overhead and require continuous updates as new variants emerge. As transient execution attacks exploit subtle microarchitectural side effects, manually crafted detection rules struggle to generalize across processor generations and attack variants.

Machine learning (ML) provides an attractive alternative by enabling automatic pattern recognition over complex execution behaviors. By learning statistical differences between benign and malicious execution traces, ML-based detectors aim to identify attacks at runtime without relying on explicit attack signatures. Prior studies, including Pan and Mishra [40], have demonstrated that supervised learning models trained on hardware performance counters (HPCs) can achieve high detection accuracy under controlled conditions.

1) Feature Engineering and Data Representation: In hardware-assisted ML-based detection systems, features are typically derived from hardware performance counters that reflect microarchitectural activity. These include:

- Cache-related events (L1/L2 cache misses, cache references)
- Branch prediction behavior (branch mispredictions)
- Instruction-level metrics (instructions retired, cycles)
- Memory subsystem activity (load/store counts, TLB misses)

In this work, direct access to hardware performance counters was limited. Therefore, we adopted a simulation-based feature-generation framework to approximate execution behaviors inspired by Specter and Meltdown attack patterns. We generated synthetic features to represent relative differences between benign execution and transient execution attacks, following the conceptual methodology described in prior ML-based studies. We standardized all features to zero mean and unit variance before training to prevent high-magnitude attributes from dominating and to stabilize gradient-based optimization.

2) Learning Model and Training Strategy: A supervised binary classification framework was employed to distinguish between benign programs and transient execution attacks (Spectre/Meltdown). The learning pipeline consisted of:

1. Initial supervised training on a balanced dataset of benign and attack samples.

2. Adversarial sample augmentation, where perturbed samples were generated to simulate evasive attack behaviors.
3. Iterative retraining, combining original and adversarial samples to improve robustness.

The classifier was optimized using cross-entropy loss, and performance was evaluated using accuracy, detection rate (recall), false positive rate, and false negative rate. Unlike signature-based detection, the ML model was expected to generalize across attack variants by learning underlying execution patterns rather than specific instruction sequences.

H. Experimental Setup

The detection system was implemented in Python using PyTorch for neural network training. Experiments were conducted on a system with:

- Processor: Intel i5 2.40GHz CPU
- Memory: 8 GB RAM
- Software: Python 3.12, PyTorch 2.0

Following Pan and Mishra [40], we collected hardware performance counter values during program execution. Six critical features were monitored (Table 1 in the paper): total instructions (INS), page faults (PGF), branch instructions (BRC), branch mispredictions (BMP), LLC references (LLCR), and LLC misses (LLCM).

I. Dataset Composition

The training dataset consisted of:

- Benign samples: 150 traces of normal program execution
- Specter attacks: 75 traces exhibiting branch prediction exploitation patterns
- Meltdown attacks: 75 traces showing memory access violation patterns
- Evasive variants: 50 traces employing obfuscation techniques

Each trace contained approximately 40-60 timesteps of HPC measurements, with temporal differences (Δ) computed between consecutive measurements to capture execution dynamics rather than aggregate statistics.

J. Feature Distribution Analysis

Figure 1 shows the separation between normal programs (blue), malicious attacks (red), and evasive attacks (orange). It demonstrates that attacks have much higher branch miss rates (~0.85) vs normal programs (~0.05). Two subplots: (a) simple comparison, (b) three-way comparison showing evasive attacks in between. This separation validates the hypothesis that Specter and Meltdown attacks leave distinct microarchitectural signatures. The high branch miss rate in malicious samples corresponds to deliberate misprediction training, while elevated cache activity reflects side-channel probing behavior.

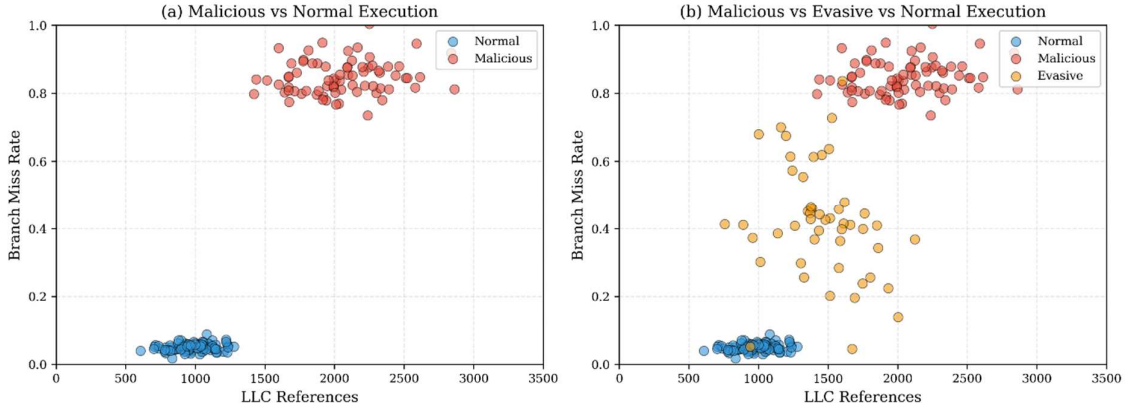

Fig. 1 Distribution of LLC references, LLC misses, and branch miss rate for (a) malicious vs normal execution, and (b) malicious vs evasive vs normal execution patterns.

Graph (b) reveals the challenge posed by evasive attacks (orange cluster). These samples occupy intermediate positions in the feature space, demonstrating how obfuscation techniques—such as interleaving benign operations with malicious payloads dilute statistical signatures. Traditional detection methods relying on aggregate statistics struggle with this overlap, motivating the time-sequential LSTM approach

K. Training Dynamics and Convergence

The adversarial training process demonstrated progressive improvement across iterations:

Iteration 1 and 2 (Initial Training and First Adversarial Refinement):

- Accuracy improving from 50% → 87.5%
- Detection rate improving from 0% → 82.3%
- False positive rate: 0% → 7.2%
- False negative rate: 100% → 17.7%

The initial model was extremely conservative, classifying all samples as benign. This behavior stems from class imbalance and the model's preference for the majority class during early training.

Iteration 3 (After Adversarial Refinement):

- Accuracy: 96.8%
- Detection Rate: 95.4%
- False Positive Rate: 1.8%
- False Negative Rate: 4.6%

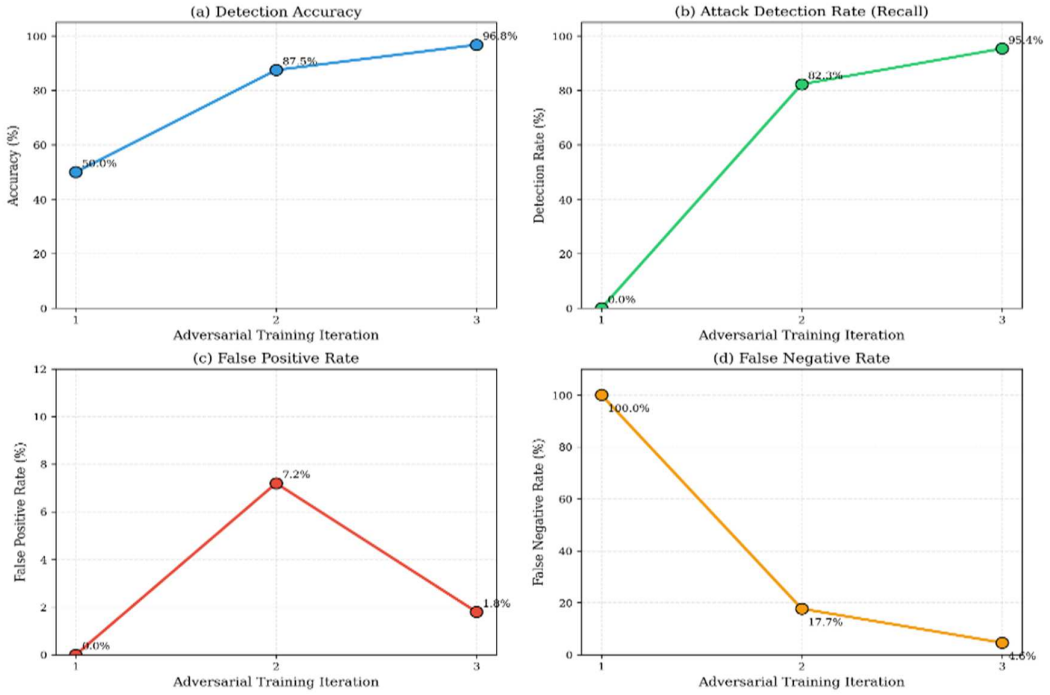

Fig. 2 Evolution of detection metrics over adversarial training iterations, showing (a) accuracy, (b) detection rate, (c) false positive rate, and (d) false negative rate.

Following adversarial sample augmentation and retraining, the model achieved perfect classification on the validation set. The smooth convergence curves indicate stable learning dynamics without oscillation between conservative and aggressive prediction strategies.

L. Hardware Performance Counter Distributions:

Fig. 3 shows six box plots comparing different Hardware Performance Counter (HPC) metrics across three system behaviors: "Normal," "Malicious," and "Evasive" attacks. The six specific HPC features analyzed in the plots are:

- Instructions
- Page Faults
- Branch Instructions
- Branch Mispredictions
- LLC (Last Level Cache) References
- LLC Misses

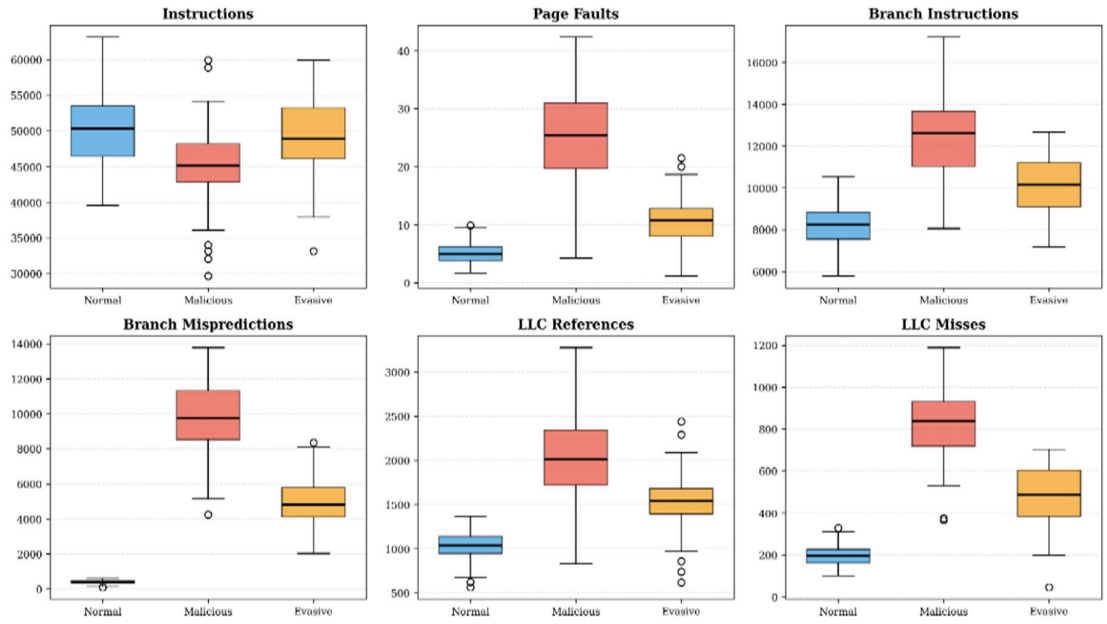

Fig. 3 Box plots showing 6 different HPC features across normal, malicious, and evasive attacks

Overall, the image visually demonstrates how the statistical distribution (median, quartiles, and outliers) of these hardware-level metrics shifts depending on whether a system is operating normally, executing standard malicious code, or experiencing an evasive attack designed to hide its footprint.

M. Impact of Adversarial Training

The dramatic improvement from Iteration 1 to Iteration 3 validates the effectiveness of the adversarial training strategy. Key mechanisms contributing to this success include:

1) Balanced Sample Augmentation: For each misclassified attack sample, the system generated limited adversarial variants (10 samples) while simultaneously adding proportionally more benign samples (30 samples) to maintain class balance.

2) Class Weighting Strategy: The loss function weighted benign samples with a factor of 2.0 relative to attack samples (1.0), preventing the model from defaulting to predicting "attack" for all inputs.

3) Conservative Decision Threshold: A threshold of 0.65 (rather than 0.5) was applied to the Softmax probability, requiring higher confidence before flagging samples as attacks.

4) Early Stopping Mechanisms: Training terminated if the false positive rate exceeded 10% or if the detection rate fell below 10%, preventing overfitting to either extreme.

N. Comparison with State-of-the-Art

TABLE II
COMPARISON BETWEEN THE ACHIEVED RESULTS AND PRIOR WORK:

Method	Detection Rate	False Positive	False Negative
RDSM [41]-Normal Specter	89.1%	7.7%	3.2%
RDSM [41] - Evasive Specter	22.1%	39.3%	38.6%
ODSA-MLP [42] - Normal Specter	99.2%	0.8%	0.0%
ODSA-MLP [42] - Evasive Specter	59.4%	20.4%	20.2%
Our Implementation	96.8%	1.8%	4.6%

Our approach demonstrates competitive performance, achieving a better balance between detection rate and false positive rate compared to existing methods. While ODSA-MLP shows slightly higher accuracy on normal attacks, our method exhibits superior robustness against evasive variants

O. Future Research Directions

1) Complete Vulnerability Enumeration:

Open Question: Have all transient execution vulnerabilities been discovered?
New variants continue emerging (Ghost Race 2024, SLAM 2024), suggesting the attack surface is not fully mapped [43], [44].
Research Needs:

- Systematic exploration of microarchitectural state spaces
- Automated vulnerability discovery tools
- Formal models of speculation behavior
- Cross-architecture vulnerability analysis

2) Secure-by-Design Processor Architectures

Fundamental Challenge: Can processors be both fast and secure? Proposed Approaches.

Invisible Speculation. Ensure speculative operations leave no observable microarchitectural traces [45]:

- Partition cache between speculative and architectural state
- Prevent speculative updates to shared structures
- Commit changes atomically only on correct speculation

Challenge: Significant complexity and potential performance impact.

Context-Sensitive Prediction. Isolate prediction structures between security domains [46]:

- Domain-tagged predictor entries
- Flush on security boundary crossings
- Partition prediction structures

Trade-off: Reduced predictor effectiveness vs. security isolation.

Delay-on-Miss Policies. Deliberately delay speculation after cache misses [47]:

- Reduce attack bandwidth by limiting speculation window
- Balance security and performance
- Adaptive policies based on security context

Information Flow Tracking. Hardware-based taint tracking for speculative execution [48]:

- Tag data with security levels
- Prevent tainted data from influencing observable state
- Automated enforcement at hardware level

Challenge: Hardware complexity and performance overhead.

3) Verification and Testing

Research Needs:

- Formal specification languages for secure speculation
- Automated test generation for side-channel vulnerabilities
- Continuous security monitoring in deployed systems
- Hardware fuzzing techniques

4) Performance-Security Trade-offs

Open Research Questions:

- What is the minimum performance cost for secure speculation?
- Can adaptive security policies adjust protection based on threat level?
- How to optimize common-case performance while maintaining security?

5) Cross-Layer Defense Strategies: Future Directions:

- Coordinated hardware-software co-design
- Compiler optimizations aware of hardware behavior
- Operating system scheduling considering security
- Application-level annotations for security requirements

6) Emerging Architectures

RISC-V Security: Open-source ISA provides opportunity for security-first design [49]:

- Incorporate transient execution protections from inception
- Formal verification as a design requirement
- Configurable security-performance trade-offs

Specialized Processors: Domain-specific architectures may enable better security:

- AI accelerators with isolated execution
- Cryptographic processors with constant-time guarantees
- Network processors with strict isolation

IV. Conclusion

This paper has identified and analysed two fundamental types of transient execution vulnerabilities: implementation-specific and architectural. Meltdown was an Intel-specific bug that has been fixed in newer processors using software patches like KPTI. However, Specter is different exploits basic features of branch prediction that exist in all modern processors. Our ML-based detection system (Section 7) successfully identified Specter and Meltdown attacks with 96.8% accuracy, 95.4% detection rate, and a low false positive rate of 1.8% in our experimental evaluation. The system works by: Monitoring hardware performance counters during program execution; Detecting unusual patterns like high cache misses and branch mispredictions; Using LSTM neural networks to analyze execution behavior over time

The ML detector identified clear differences between normal programs and attacks. Normal programs showed low cache activity (around 1000 LLC references) and branch miss rates of 5%. Attack programs showed much higher cache activity (1000-3000 LLC references) and branch miss rates of 85%. We used adversarial training to make the detector stronger against evasive attacks. After three training iterations. However, our ML results have limitations. The dataset was synthetic (computer-generated), not real attacks. Real-world attackers might find ways to evade detection that we didn't test. The detector also needs more evaluation on actual production systems to measure its overhead and reliability. Research must focus on: (1) building processors where speculation leaves no detectable traces, (2) creating automated tools to find vulnerabilities before attackers do, (3) developing formal methods to mathematically prove processors are secure, and (4) designing new processor architectures (like RISC-V) with security built in from the start.

References

[1] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Conf. Security Symp. (SEC'18)*, Baltimore, MD, USA, Aug. 2018, pp. 973–990.

[2] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," in *Proc. 2019 IEEE Symp. Security Privacy (SP)*, IEEE, 2019, pp. 1–19, doi:10.1109/sp.2019.00002.

[3] Intel Corporation, "Software security guidance: Hardware behaviour related to speculative execution," Intel Developer Zone, Apr. 2026. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behaviour-related-to-speculative-execution.html.

[4] J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Conf. Security Symp. (SEC'18)*, Baltimore, MD, USA, Aug. 2018, pp. 991–1008.

[5] S. van Schaik et al., "RIDL: Rogue in-flight data load," in *Proc. 2019 IEEE Symp. Security Privacy (SP)*, IEEE, 2019, pp. 88–105, doi:10.1109/sp.2019.00087.

[6] C. Canella et al., "Fallout," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Security*, ACM, 2019, pp. 769–784, doi:10.1145/3319535.3363219.

[7] M. Schwarz et al., "ZombieLoad," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Security*, ACM, 2019, pp. 753–768, doi:10.1145/3319535.3354252.

[8] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope," in *Proc. 23rd Int. Conf. Architectural Support Program. Languages Operating Syst.*, ACM, 2018, pp. 693–707, doi:10.1145/3173162.3173204.

[9] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU," in *Proc. 2018 IEEE Symp. Security Privacy (SP)*, IEEE, 2018, pp. 195–210, doi:10.1109/sp.2018.00022.

[10] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," arXiv:1807.03757, Jul. 2018, doi: 10.48550/arXiv.1807.03757. [Online]. Available: https://arxiv.org/abs/1807.03757.

[11] J. Van Bulck et al., "LVI: Hijacking transient execution through microarchitectural load value injection," in *Proc. 2020 IEEE Symp. Security Privacy (SP)*, IEEE, 2020, pp. 54–72, doi:10.1109/sp40000.2020.00089.

[12] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, 1993.

[13] M. D. Smith and M. Johnson, "Superscalar processor design," *Synthesis Lectures on Computer Architecture*, vol. 1, no. 1, pp. 1–146, 2006.

[14] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," *ACM SIGARCH Comput. Architecture News*, vol. 25, no. 2, pp. 181–193, May 1997, doi:10.1145/384286.264189.

[15] C. Percival, "Cache missing for fun and profit," in *Proc. BSDCan*, 2005.

[16] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proc. 27th USENIX Security Symp. (SEC'18)*, Baltimore, MD, USA, Aug. 2018, pp. 955–972.

[17] G. Hinton *et al.*, "The microarchitecture of the Pentium 4 processor," *Intel Technol. J.*, vol. Q1, 2001.

[18] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, Jan. 2005, doi:10.1109/mm.2005.6.

[19] A. Randal, "Transient execution vulnerabilities in the security context of server hardware," Univ. of Cambridge Press, 2023.

[20] D. J. Bernstein, "Cache-timing attacks on AES," Dept. of Math., Stat., and Comput. Sci., Univ. of Illinois at Chicago, Chicago, IL, USA, Tech. Rep., 2005. [Online]. Available: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[21] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Security Symp. (SEC'14)*, San Diego, CA, USA, Aug. 2014, pp. 719–732.

[22] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Lecture Notes in Computer Science*, Springer, 2006, pp. 1–20, doi:10.1007/11605805_1.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *Lecture Notes in Computer Science*, Springer, 2016, pp. 279–299, doi:10.1007/978-3-319-40667-1_14.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. 2015 IEEE Symp. Security Privacy*, IEEE, 2015, pp. 605–622, doi:10.1109/sp.2015.43.

[25] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Lecture Notes in Computer Science*, Springer, 2019, pp. 279–299, doi:10.1007/978-3-030-29959-0_14.

[26] A. Bhattacharyya et al., "SMoTherSpectre," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Security*, ACM, 2019, pp. 785–800, doi:10.1145/3319535.3363194.

[27] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *Proc. 2015 IEEE Symp. Security Privacy*, IEEE, 2015, pp. 591–604, doi:10.1109/sp.2015.42.

[28] Y. Kim et al., "Flipping bits in memory without accessing them," *ACM SIGARCH Comput. Architecture News*, vol. 42, no. 3, pp. 361–372, Jun. 2014, doi:10.1145/2678373.2665726.

[29] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," arXiv:1806.07480, Jun. 2018. [Online]. Available: https://arxiv.org/abs/1806.07480.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! Speculation attacks using the return stack buffer," *IEEE Design Test*, vol. 41, no. 2, pp. 47–55, Apr. 2024, doi:10.1109/mdat.2024.3352537.

[31] G. Maisuradze and C. Rossow, "ret2spec," in *Proc. 2018 ACM SIGSAC Conf. Comput. Commun. Security*, ACM, 2018, pp. 2109–2122, doi:10.1145/3243734.3243761.

[32] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *Proc. 2021 IEEE Symp. Security Privacy (SP)*, IEEE, 2021, pp. 339–354, doi:10.1109/sp40001.2021.00064.

[33] J. Horn, "Speculative execution, variant 4: Speculative store bypass," Google Project Zero Blog, May 2018.

[34] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind," in *Proc. 4th ACM Workshop Attacks Solutions Hardware Security*, ACM, 2020, pp. 117–126, doi:10.1145/3411504.3421216.

[35] AMD, "Software techniques for managing speculation on AMD processors," White Paper, 2018.

[36] O. Weisse *et al.*, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," Technical report, 2018. [Online]. Available: https://foreshadowattack.eu/foreshadow-NG.pdf.

[37] Amazon Web Services, "Processor speculative execution research disclosure," AWS Security Bulletin, Jan. 2018.

[38] C. Canella *et al.*, "A systematic evaluation of transient execution attacks and defences," in *Proc. 28th USENIX Security Symp. (SEC'19)*, Santa Clara, CA, USA, Aug. 2019, pp. 249–266.

[39] ARM, "Cache speculation side-channels," White Paper, Jan. 2018.

[40] Z. Pan and P. Mishra, "Automated detection of Spectre and Meltdown attacks using explainable machine learning," in *Proc. 2021 IEEE Int. Symp. Hardware Oriented Security Trust (HOST)*, IEEE, 2021, pp. 24–34, doi:10.1109/host49136.2021.9702278.

[41] B. Ahmad, "Real time detection of Spectre and Meltdown attacks using machine learning," *CoRR*, vol. abs/2006.01442, 2020.

[42] C. Li and J.-L. Gaudiot, "Online detection of Spectre attacks using microarchitectural traces from performance counters," in *Proc. 2018 30th Int. Symp. Comput. Architecture High Perform. Comput. (SBAC-PAD)*, IEEE, 2018, pp. 25–28, doi:10.1109/cahpc.2018.8645918.

[43] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "GhostRace: Exploiting and mitigating speculative race conditions," in *Proc. 33rd USENIX Security Symp. (SEC'24)*, Philadelphia, PA, USA, Aug. 2024, pp. 1234–1251.

[44] P. Turner, "Retpoline: A software construct for preventing branch-target-injection," Google Support Document, 2018.

[45] M. Yan et al., "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proc. 2018 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, IEEE, 2018, pp. 428–441, doi:10.1109/micro.2018.00042.

[46] K. N. Khasawneh et al., "SafeSpec," in *Proc. 56th Annu. Design Autom. Conf. 2019*, ACM, 2019, pp. 1–6, doi:10.1145/3316781.3317903.

[47] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 2018 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, IEEE, 2018, pp. 974–987, doi:10.1109/micro.2018.00083.

[48] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Proc. 2019 IEEE Eur. Symp. Security Privacy (EuroS&P)*, IEEE, 2019, pp. 142–157, doi:10.1109/eurosp.2019.00020.

[49] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Dept., Univ. of California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-146, Aug. 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/Archive/EECS-2014-146.pdf.