POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

An automatic framework to detect transient execution vulnerabilities in modern CPUs through an architecture-agnostic analysis

Author:
Luca Bancale

Student ID:
10713897

Advisor:
Prof. Luca Cassano

Co-Advisor:
Dr. Elia Lazzeri

Academic Year:
2025-26

Dedicated to my family.

Abstract

In recent years, modern processors have achieved high performance through aggressive micro-architectural optimisations such as speculative execution, branch prediction, cache hierarchies and out-of-order execution. However, these performance gains have led to the emergence of transient execution vulnerabilities, a novel class of micro-architectural flaws that enable the leakage of sensitive information through traces left behind during transient execution.

Evaluating whether the combinations and interactions of such optimisations will lead to exploitable behaviours remains an open challenge, especially for independent analysis. Existing approaches still rely on privileged knowledge of microarchitectural designs or on ad hoc reverse-engineering techniques that limit their reproducibility and portability across different architectures.

This thesis proposes a systematic architecture-agnostic, software-only approach to infer the presence and properties that enable transient execution effects based purely on observable timing effects. Moreover, we introduce a novel security framework for evaluating these timing measurements and assessing the processor resilience against different transient execution attack families. By relying solely on timing measures, our approach aims to be extremely portable across all different CPU architectures.

Our results show that runtime behaviour analysis can be used to infer security properties of modern processors by using the measurable execution leakage associated with transient execution. This demonstrates that such effects can be systematically measured at runtime across different security boundaries and execution contexts. Lastly, we show that these properties can be used to infer a processor's vulnerability with respect to a class of transient execution attacks. Aggregated vulnerability scores of 45.4% and 33.1% for the Intel Core Ultra 7 and AMD Ryzen 9 7900X, respectively, reflect measurable differences in hardware resilience across platforms.

Keywords: transient execution, microarchitectural attacks, speculative execution, timing side channels, architecture-agnostic analysis, hardware security, CPU, processors

Abstract in lingua italiana

Negli ultimi anni, i processori moderni hanno raggiunto elevate prestazioni grazie ad aggressive ottimizzazioni micro-architetturali, come l'esecuzione speculativa, il branch prediction, le gerarchie di cache e l'esecuzione out-of-order. Tuttavia, queste innovazioni e miglioramenti di prestazione hanno portato all'emergere delle vulnerabilità di transient execution, una nuova classe di difetti micro-architetturali che consentono la fuga di informazioni sensibili attraverso tracce residue lasciate durante l'esecuzione transiente.

La valutazione della sfruttabilità dei comportamenti derivanti dalle interazioni di tali ottimizzazioni rimane una sfida aperta, soprattutto nel contesto di valutazione indipendente dei processori. Gli approcci esistenti si basano ancora su informazioni privilegiate relative alle microarchitetture o su tecniche di reverse engineering ad-hoc, che ne limitano la riproducibilità e la portabilità tra architetture differenti.

Questo lavoro propone un approccio sistematico di tipo architecture-agnostic, interamente basato su software, per inferire la presenza e le proprietà di effetti di transient execution a partire unicamente da misure temporali.

Inoltre, viene introdotto un nuovo framework di sicurezza per valutare tali osservazioni e per analizzare la resilienza dei processori rispetto alle diverse famiglie di attacchi di transient execution. Basandosi interamente su misure temporali, il metodo proposto mira a essere altamente portabile su differenti architetture di CPU.

I nostri risultati mostrano che l'analisi del comportamento a tempo di esecuzione può essere utilizzata per inferire proprietà di sicurezza dei processori moderni sfruttando le fughe di informazione misurabili associate all'esecuzione transiente. Questo dimostra che tali effetti possono essere osservati e misurati in modo sistematico durante l'esecuzione, in diversi contesti e livelli di isolamento. Infine, mostriamo che tali proprietà possono essere utilizzate per valutare la vulnerabilità di un processore rispetto a una classe di attacchi di transient execution. Punteggi di vulnerabilità aggregati del 45.4% e 33.1% rispettivamente per Intel Core Ultra 7 e AMD Ryzen 9 7900X riflettono differenze misurabili nella resilienza hardware tra le piattaforme.

Parole chiave: transient execution, attacchi micro-architetturali, speculative execution, side channel temporali, analisi architecture-agnostic, sicurezza hardware, CPU, processori

Contents

.

1 | Introduction

Since their inception, microprocessors have been designed with performance as their primary and unchanging objective. The paradigms through which this objective is pursued, however, have undergone a fundamental transformation. Prior to the mid 2000s, the primary contribution for performance gains came from the evolution of two metrics: the continuous miniaturization of transistors and the increase of the clock frequency. While the former has continued to follow Moore's Law, the steady escalation of clock speed has stagnated due to power [5] and memory latency limitations [41].

Hardware architects therefore pivoted toward aggressive microarchitectural optimisations that exploit Instruction Level Parallelism (ILP). Techniques such as out-of-order execution, deep cache hierarchies, branch prediction and speculative execution became the dominant mechanisms through which modern processors sustain throughput well beyond what sequential, in-order execution could deliver. Together, these mechanisms created a fundamental divide between the logical instruction flow presented to software and the actual execution behaviour of the processor, with the compounding effect of increasing chip complexity exponentially. Unpredictable data access patterns quickly became a serious performance stopper. Whenever the heuristic that govern cache logic fail, memory latencies do not get any benefit, and the execution unit will have to remain idle, starved waiting for data. Worsening the problem, conditional control flow made matters more complicated, as it is not possible to determine with any degree of certainty which data will be required until the processor actually tries to load it.

This last problem introduced a new execution paradigm. Rather than waiting for the execution unit to resolve the correct control flow, a processor may instead execute an educated "guess", pre-emptively executing the predicted path. Once the execution unit finally resolves the branch condition or memory address, then, in the unfortunate case of a wrong guess, it can correct itself and roll back the execution state to the point that created the speculative flow.

Branch prediction and speculative execution became the operational cornerstones of this shift towards predictive microarchitectures. This design direction altered how software interacts with hardware. Historically, this relationship has been defined by the Instruction Set Architecture (ISA), that being the abstract contract that guarantees instruc-

tions execute sequentially and deterministically from the software's perspective. While earlier ILP techniques like pipelining and out-of-order execution decoupled the physical execution order from the program order, speculative execution decoupled execution from the actual control flow graph itself. By processing instructions along uncommitted execution paths, the microarchitecture routinely computes instructions that do not exist within the program's legitimate logical flow. When a misprediction occurs, the processor maintains the ISA contract by rolling back the architectural state. However, it leaves the underlying microarchitectural state, like cache hierarchies, altered. This exact divergence between the cleared architectural state and a dirty microarchitectural state constitutes a measurable attack surface that forms the technical basis for Transient Execution Vulnerabilities.

Attacks targeting Transient Execution Vulnerabilities are known as Transient Execution Attacks (TEA) and, unlike passive side channel attacks that monitor physical traits like power consumption or electromagnetic radiation during program execution, they are a form of cache-based side channel attack that aim to collect traces of the divergence left in the microarchitectural state by transient execution with the purpose of stealing data from other execution contexts. The first real world exploits of this kind, namely Spectre [15] and Meltdown [21] that were first made public in early 2018 and their subsequent variants, have demonstrated their efficacy in breaking every possible isolation and security boundary present in both the software and the hardware.

In fact, many types of attacks were developed specifically for breaking each of the existing measures, making them able to bypass boundaries such as process level isolation, kernel level isolation, computing enclaves such as Intel's Software Guard Extension (SGX) [26], used heavily in conjunction with cryptographic primitives, and even virtual machine protections. Modern computing environments heavily rely on these isolation layers to maintain data confidentiality across both public multi-tenant infrastructures and private endpoints. In cloud computing platforms, which now serve as the baseline architecture for modern digital services, multiple mutually untrusted virtual machines share the same physical hardware resources. Similarly, modern web browsers routinely execute untrusted, third-party code within isolated sandboxes on end-user devices. Because Transient Execution Attacks bypass both software and hardware-defined security barriers by exploiting the physical resource-sharing that is crucial for modern microarchitectures, they can invalidate the fundamental trust assumptions of these environments. As a result, a malicious tenant in a cloud cluster or an untrusted script in a browser can implicitly observe protected memory, turning the persistence of these vulnerabilities into a severe risk to data confidentiality.

Mitigating this novel type of attack proved rather challenging, mostly due to TEA tar-

geting hardware behaviour, making software mitigation at best inconsistent and at worst ineffective [2]. Hardware mitigations were required also because this type of attack arises from intended hardware optimisations that no hardware manufacturer of high-end enterprise chips could simply ignore due to the performance benefits that they provide. The real-world consequences of this challenge extended well beyond the theoretical. The software mitigations deployed in response, primarily kernel page-table isolation for Meltdown and retpoline-based branch restriction for Spectre, introduced substantial performance overheads on affected systems, with reported degradation ranging from a few percent to over 30% for the most heavily affected workloads [2]. The cost was not limited to throughput alone. A systematic study of the overhead at the operating system level found that the energy penalty of these mitigations is highly application-dependent, ranging from negligible to as much as 72% for subsystems with frequent operating system interaction [11]. At the scale of modern cloud data centres, such figures translate directly into increased power draw and capacity planning pressure, effectively reversing a portion of the efficiency gains that speculative optimisations had originally delivered. Because transient execution vulnerabilities are fundamentally rooted in the physical design of the processor, deploying effective hardware-level mitigations is an exceptionally slow process. Because these speculative mechanisms are deeply ingrained features of the microarchitecture, mitigating these design issues required a complete silicon redesign, a process that takes a significant amount of time measurable in years rather than months, forcing infrastructure providers and end-users to replace the entire processor with a new hardened generation of silicon.

To circumvent this lengthy and costly hardware replacement cycle, vendors have primarily relied on software-level workarounds and microcode updates. These measures typically involve inserting serialisation primitives, such as memory fences, to halt execution until speculation is resolved. However, these solutions introduce a critical trade-off. They impose severe computational overheads, effectively sacrificing the throughput gained from optimisations like out-of-order execution. More importantly, these solutions are often fragile and narrow in scope as they treat the symptoms rather than the root cause. Subsequent security research has consistently shown that existing software mitigations can be bypassed by targeting alternative predictive structures or newly discovered speculation primitives. This pattern of iterative disclosure and reactive countermeasures confirms that transient execution vulnerabilities represent a fundamental feature of speculative microarchitectures rather than a collection of isolated, patchable bugs. Retpoline, the dominant low-overhead software defence against Spectre variant 2, was demonstrated in 2022 to be bypassable by a new class of attacks [38], forcing the reintroduction of a heavier microcode-based mitigation and its associated performance regression. Hardware manufacturers have been compelled to redesign silicon while operating under the expec-

tation that further variants will continue to emerge from the same structural properties of speculative execution.

This reliance on narrow patches creates a dangerous discrepancy between a system's reported security status and its actual physical resilience. When a software mitigation is applied, vendor documentation and architectural status flags typically report the system as "patched" or "safe" [20]. However, because these defences only block specific, known attack vectors, the underlying hardware may still be exploitable by slightly modified variants. On the hardware mitigation front, most of the effort is directed towards the design of mechanisms that do not leave any recoverable microarchitectural trace in case of an architectural rollback. Such results are achieved through the use of architecture-aware design techniques, where the actual physical design is formally tested. Architecture-aware verification, however, has a fundamental limitation that is intrinsic of this type of analysis, that being the requirement of the Register-Transfer Level (RTL) specification. Meaning that only those who have access to the design can verify its security guarantees, and these are very few entities since most chip designs are proprietary.

Regardless of the strong guarantees they provide, architecture-aware methods are entirely inaccessible for evaluating Commercial of the Shelf (COTS) processors, as they require proprietary design specifications unavailable outside the chip manufacturer. Since the primary externally observable side-effect of transient execution is a measurable alteration of the memory's timing behaviour, runtime timing analysis constitutes the most natural and universally accessible signal for post-silicon security assessment, as they require neither access to hardware internals nor elevated architectural privileges. Consequently, post-silicon verification must rely on such architecture-agnostic approaches to assess the mitigations actually present in fabricated hardware. However, current post-silicon tools remain fundamentally limited. Frameworks attempting to infer hardware mitigation status often rely on replaying known Proof of Concept (PoC) exploits, making them entirely dependent on the completeness of existing exploit sets and remain blind to undocumented or yet to be discovered variants. Furthermore, they remain completely hardware dependent due to the very nature of TEA, and, in some cases, are even Operating System (OS) dependent since some PoC are only developed targeting few systems.

In addition to static analysis, attempts to monitor hardware behaviour defensively using Hardware Performance Counters (HPC) have similarly proven unreliable. As demonstrated by evasion techniques like Vizard [32], attackers can effectively bypass these detectors by generating counter-balanced activity or maintaining low-amplitude leakage that hides beneath the hardware noise threshold. This reality highlights a critical evaluation gap for independent security validation. Because vulnerable hardware remains widely deployed due to slow silicon replacement cycles, and because software mitigations provide

an incomplete shield, independent analysts cannot simply rely on OS-level status flags or vendor claims to assume a system is secure.

1.1. Thesis objective and contributions

This limitation directly motivates the aim of this thesis, that is, to provide a systematic, architecture-agnostic methodology that bypasses the opacity of the underlying hardware and the unreliability of software status flags. By doing so, it allows researchers to empirically test and characterise transient execution behaviours purely through observable runtime timing effects, verifying what the processor actually does rather than what it claims to do. To achieve this, this thesis develops a robust, software-only framework for assessing processor resilience against known families of TEA. Central to this framework is a novel security metric designed to quantify the degree of microarchitectural leakage across security boundaries, thus providing an objective, continuous measure of processor resilience rather than a binary vulnerable/patched classification. This metric is the instrument through which the framework moves beyond simply detecting the presence of transient execution effects and begins to characterise their exploitability. A primary objective of this thesis is also to obtain results that are easily reproducible and verifiable by independent third parties. Because architecture-agnostic timing analysis is inherently sensitive to environmental noise and platform-specific variations, a significant portion of this research focuses on establishing rigorous, standardized measurement protocols. By meticulously documenting the techniques, signal-filtering steps, and statistical methods required to isolate transient execution footprints, this thesis aims to provide a solid, open foundation for future independent hardware security research. The central contribution of this research is to act as a comprehensive, vendor-agnostic guide and toolset for evaluating hardware susceptibility to TEAs. Currently, architecture-agnostic analysis of modern processors remains fragmented, often relying on isolated, architecture-specific PoC exploits that do not generalize. This research attempts to address these gaps by providing an end-to-end security assessment methodology that operates consistently across diverse execution contexts. Specifically, the study's focus will be on the following key objectives:

1. The implementation of a architecture-agnostic security verification framework driven entirely by unprivileged software timings and microarchitectural feature detection.

2. The development of a continuous scoring model that translates timing measurements into structured assessments of processor resilience and quantifies the severity of security boundary breaches.

By constructing this unified evaluation methodology, this thesis serves the purpose of

equipping independent researchers and software developers with a reliable, single point of access for hardware vulnerability assessment. Ultimately, this work seeks to enhance overall hardware transparency, shifting the paradigm of microarchitectural security from vendor-dependent claims to verifiable, empirical evidence.

1.2. Thesis Structure

Chapter 1 - Introduction

This chapter establishes the groundwork for the thesis by defining the critical security challenges introduced by modern hardware optimisations. It details the problem statement i.e. the persistence of transient execution flaws in shared computing environments. Furthermore, it outlines the severe limitations of current hardware verification methodologies, contrasting the inaccessibility of pre-silicon design analysis with the fragility of existing post-silicon exploit testing. Finally, the chapter defines the overarching goals and core technical contributions of this research, proposing a shift from vendor-dependent security claims to empirical, independent hardware validation.

Chapter 2 - Background

The second chapter develops the fundamental technical context required to understand microarchitectural vulnerabilities at the hardware level. It begins by exploring the interaction between Instruction Set Architecture and aggressive hardware optimisations, explaining how mechanisms such as superscalar pipelines, dynamic branch prediction, out-of-order execution and hardware buffers operate to maximise instruction throughput. The chapter then transitions to microarchitectural attacks, detailing how these optimisations inadvertently create transient execution windows. It thoroughly examines the underlying mechanics of timing-based side channels, particularly cache-level interactions, explaining how attackers can reliably extract sensitive data from the microarchitectural state left behind after architectural rollbacks. Lastly, it will examine the functioning of known Transient Execution Attacks (TEA) families in order to specify which microarchitectural feature enables them.

Chapter 3 - Related Work

This chapter provides a comprehensive and critical review of the existing literature within the domain of processor security evaluation. It categorises current research efforts into two primary paradigms. The first is pre-silicon architecture-aware analysis, which focuses on formal verification, RTL fuzzing, and hardware-software safety contracts avail-

able strictly to chip designers. The second is post-silicon architecture-agnostic verification, where the chapter distinguishes between offline vulnerability detection and real-time security monitoring via hardware performance counters. Through this systematic review, the chapter explicitly isolates the theoretical and practical evaluation gaps that directly motivate the development of our proposed methodology.

Chapter 4 - Proposed Security Verification Framework

Chapter 4 details the core engineering contribution of this thesis, outlining the design and implementation of the testing framework. It breaks down the system's architectural design, detailing how microbenchmarks are executed across varying privilege levels. The chapter then describes the hierarchical structure of the testing suite, differentiating between isolated *feature tests*, designed to profile the latency and presence of hidden structures like Pattern History Tables or Reorder Buffers, and *system interaction tests*, which calculate transient window sizes and evaluate cross-boundary leakage. Crucially, this chapter formalizes the differential validation methodology, explaining how software-enforced mitigations are used to collapse timing differentials (δ), thereby mathematically validating the accuracy of the framework's measurements. Finally, it defines the three-level vulnerability scoring model (Section 4.1.3) that translates validated timing data into structured security assessments.

Chapter 5 - Experimental Results

This chapter presents the results of this research, reporting the data gathered from executing the framework across different hardware platforms. It begins by explicitly defining the specific attack families under study, detailing the precise exploitation vectors that our developed tests were designed to target. The chapter then systematically analyses the quantitative findings of our cross-boundary security analysis and comparing them with reported results of security advisories, demonstrating how the framework uses runtime timing variations to infer a processor's actual vulnerability status without relying on proprietary vendor documentation.

Chapter 6 - Conclusions and Future Work

The final chapter synthesises the core findings of the thesis, assessing the overall effectiveness and reliability of architecture-agnostic runtime behaviour analysis for independent hardware validation. It then reflects on the success of the methodology in addressing the gaps identified in current literature. Additionally, the chapter critically examines the current boundaries and constraints of timing-only detection, such as the challenges in-

troduced by extreme environmental noise. Finally, it proposes concrete future research directions, discussing the potential flaws and limitations not addressed by this thesis.

2 | Background

This chapter develops the technical context required to understand microarchitectural vulnerabilities. The first half covers modern CPU architecture, building from the ISA abstraction down to the specific hardware optimisations that create transient execution effects. The second half covers the attack side, that is, how timing leakage is observed and exploited.

2.1. Modern Computer Architecture

Transient execution vulnerabilities do not arise from programming errors or flawed protocols, they are the direct consequences of highly optimised designs. To understand how they emerge, it is necessary to first understand the architectural abstractions that processors expose to software, and then the successive layers of microarchitectural optimisation that implement those abstractions in silicon.

At the outermost layer sits the Instruction Set Architecture governing the execution of a program and guaranteeing that the results of execution have to give a sequential, deterministic view of computation. Beneath it, however, the hardware is free to break this contract to pursue whichever execution strategy it sees fit. The performance demands outlined in Chapter 1 forced architects to introduce a myriad of optimisations like pipelining, out-of-order execution, speculative execution, and multi-level caching, each of which decouples the processor's internal behaviour further from the sequential model the ISA projects to the software.

This section traces that widening gap layer by layer. Each subsection introduces one optimisation, establishes the internal hardware state it creates, and identifies the property that will later be exploited. The central observation that connects every mechanism described here is the same: the ISA contract is maintained at instruction retirement, but the microarchitectural state produced along the way has no such guarantees. In fact, it's precisely this residue that is scattered around microarchitectural buffers and pipelines that can never be committed that constitutes the attack surface examined in Section 2.2.

2.1.1. Instruction Set Architecture

The Instruction Set Architecture (ISA) is the formal specification that defines the boundary between software and hardware. It defines the instruction a processor must be able to understand and execute their side effects, it must also describe the type of memory model and privilege levels it must enforce. So, it's the hardware language and API that programs need to adhere to to be able to run, but if a program respects it, then it will be able to be executed by any other ISA compliant processor. From the perspective of the running program, the ISA provides two fundamental guarantees:

- Instructions must appear to be executed sequentially in the specified order, so the effect of the $i + 1$-th instruction must never be visible before the effects of the i-th instruction have been committed to the architectural state

- The state of all specified buffers, such as register file, program counters and memory, are to be updated deterministically and exclusively by retired instructions. Thus, any instruction that is not retired should appear as if it has not been executed yet.

These two properties together constitute what is commonly referred to as the *sequential execution model*, or the *programmer's model* [10]. It's the contract under which all software is written and all compilers generate code. The ISA is an abstraction, not an implementation. It specifies the *what* of processor behaviour, that is, what each instruction must produce, but without constraining the *how*. This separation is deliberate and economically significant, in fact a single ISA can be implemented by many different microarchitectures, each making independent tradeoffs between power, cost, die area, and performance. Intel's Skylake and AMD's Zen 2, for instance, both implement the x86-64 ISA and execute the same binary code, yet their internal pipeline structures and cache topologies are completely different. This distinction implies that security properties are microarchitecture-specific, not ISA-specific. A binary that executes safely on one compliant processor may behave differently on another, and a mitigation that closes a vulnerability on one design may provide no protection on another implementing the same ISA.

The in-order execution model promised by the ISA is, in a sense, just a carefully maintained illusion. Since the ISA is just an architectural contract, hardware optimisations allow modern processors microarchitecture to deviate from this reality. The following subsections examine each of these optimisations in turn, with attention to the specific structures they introduce and the properties that make those structures observable from outside the processor's architectural boundary.

2.1.2. Microarchitectural Optimisations

Instruction-Level Parallelism and Pipelining

Instruction Level Parallelism (ILP) is the main driver through which CPU designers continued to deliver performance gains even after the fall of Dennard Scaling. Since raw frequencies could not be scaled further, execution speed could not achieve further significant improvements, so the focus shifted to improving throughput. So, rather than optimising the execution of single instructions, the CPU is now designed to execute multiple instructions simultaneously, overlapping in their execution. The earliest and most important realisation of this idea is pipelining. A pipeline splits instructions into a fixed sequence of discrete stages. The classical implementation consists of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and, finally, Write Back (WB). Each stage of execution occupies one clock cycle and can operate on each stage simultaneously, so at any given cycle the pipeline holds up to five instructions at different stages of computation. Here is where the definition of performance for a CPU undergoes a paradigm shift: the latency of an operation becomes the number of cycles needed to traverse the entire pipeline and performance begins to be measured by Instructions Per Cycle (IPC). In this first iteration, the ideal performance reaches one instruction per cycle.

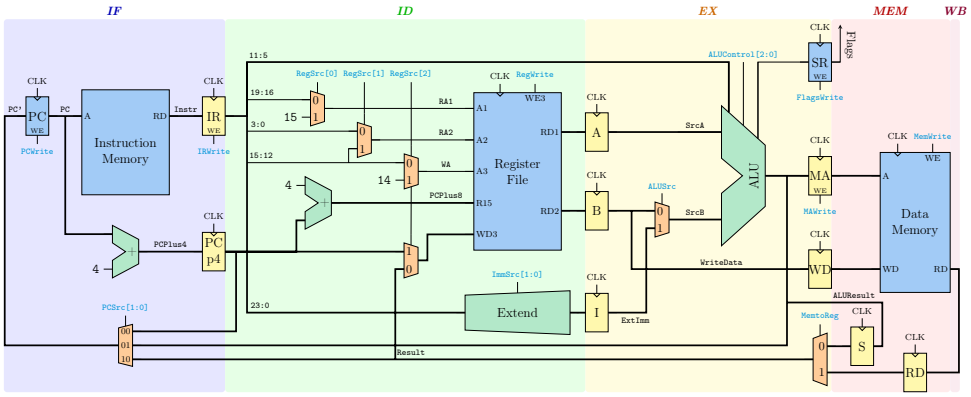

Figure 2.1: Multi-cycle Pipeline Datapath Diagram

One big issue with this approach is that the clock frequency would have to be dictated by the slowest instruction in the EX stage and the MEM stage would still halt execution waiting for memory. To remedy these issues and exceed the theoretical maximum of IPC=1, architects introduced superscalar execution. A superscalar processor implements multiple execution units in its EX stage such that multiple instructions can be issued at the same time, sometimes multiple in the same clock cycle, and executed concurrently. The number of instructions that a superscalar processor can issue in the same clock cycle

is called issue width (IW). Modern processors can offer up to IW=6. The throughput ceiling is no longer the pipeline itself, but the availability of independent work to fill it. This new bottleneck is created because not all instructions are independent from each other. There are three classes of pipeline hazards that prevent the ideal of perpetually full execution units:

- Structural hazards arise when two instructions require the same hardware resource in the same cycle
- Data hazards happen when there is a dependency between the results and operands of instructions. They can be subdivided into true data dependency, or Read-After-Write (RAW), that happens when an instruction requires a value that a preceding instruction has not yet produced, anti-dependency, or Write-After-Read (WAR), and output dependency, or Write-After-Write (WAW), that happen due to the reuse of register names, rather than data flow
- Control hazards are created from conditional branches where the next instruction to fetch after the branch cannot be determined

Whenever hazards occur, they are resolved by stalling the CPU until their resolution. This constraint becomes a hard performance ceiling once instructions with different latencies meet in the pipeline. Consider a load instruction that must wait several hundred cycles for a DRAM response, followed immediately by an arithmetic instruction with no data dependency on that load. In an in-order processor, the arithmetic instruction cannot be dispatched while the load occupies the execute stage, and it cannot retire before the load retires, since retirement must preserve program order to uphold the ISA contract. Every execution unit behind that load sits idle, starved, for the full duration of the miss, regardless of how much independent work is waiting. In such cases the bottleneck is not a lack of execution resources but an artificial sequencing constraint: instructions are being held back not because they depend on each other, but because the hardware has no mechanism to distinguish instructions that genuinely must wait from those that merely appear later in program order.

The Memory Hierarchy

The performance gains delivered by pipelining and superscalar execution depend entirely on the assumption that the data required for executing an instruction is available. In practice, this assumption fails frequently. The fundamental problem, termed the *Memory Wall* by Wulf and McKee in 1995 [41], is a structural timing mismatch: on modern hardware, a processor operates at 3–5GHz, giving it a clock cycle of roughly 0.2–0.3ns, while

DRAM memory access has a latency of at least 2 orders of magnitude higher. Thus, an execution unit will stall for at least hundreds of cycles, leaving the execution unit idle for the full duration of the miss. To alleviate this latency bottleneck, processors incorporate caches, they are faster, but smaller memories that sit close to the core. They hold any data that could be used in any future fetch. For this reason they are designed to exploit both temporal and spatial locality, staging data that has been recently used or resides nearby in memory. Because caches must be small to remain fast, they cannot store the entire address space. Instead it is main memory that will serve as an authoritative backing store, while caches will hold coherent, but incomplete subsets of it. This introduces two critical design constraints: capacity limitations and coherency requirements.

Cache organisation. The standard architectural solution is to employ multiple levels of caches, where a processor has multiple levels of caches, each progressively larger and slower the further they are placed from the execution units. Though modern processors are built using the Von Neumann architecture, that specifies that there is only a single shared address space, the first level of cache usually employs the Harvard architecture where data and instructions live in physically different memories, this is also known as the modified Harvard architecture. Since caches are not tied to the execution pipeline, to optimise throughput, modern multi-core processors organise their memory hierarchy into a mix of private and shared resources to minimise main memory fetches. Table 2.1 summarises the typical parameters of a modern three-level hierarchy.

Level	Latency	Typical size	Sharing
L1 (I$ + D$)	∼4 cycles	32–64 KB per core	Private
L2	∼12 cycles	256 KB – 1 MB	Private
L3	∼40 cycles	8–64 MB	Shared across cores
DRAM	∼200 cycles	GBs	Shared across sockets

Table 2.1: Typical memory hierarchy parameters on a modern x86-64 processor. Latency figures are approximate and vary across microarchitectures.

Caches are organised in *cache lines*, the atomic unit of transfer between memory levels. On most modern processors, these lines are 64 bytes wide, this means that regardless how many words are requested, a load will always fetch 64-byte aligned blocks into the cache. To locate lines efficiently, memory addresses are partitioned into three fields: the low-order bits form a *line offset* that selects a byte within the line; the middle bits form a *set index*; and the remaining upper bits form a *tag*. The number of bits allocated to each field and access patterns are determined by the cache geometry.

A direct-mapped cache maps one address to one specific set, they are very fast but suffer from frequent conflict misses that happen when an address is loaded into an already populated cache line. Fully associative caches are the polar opposite, addresses that get loaded can be put in any available cache line, these greatly reduce conflict misses, but at scale they greatly slows down cache hits since they require a comparison against every resident line on each access. Modern cache implementations are *set-associative*, a mapping design that balances the simplicity of direct-mapped caches against the flexibility of fully associative ones. A set-associative cache is divided into S sets, each containing W *ways*, where every way can hold one cache line. Upon cache access, the set index of the address selects the target set and then the tag field tries to match against all stored W ways of the set. When a match is found the data is returned immediately, this is called a cache hit, the absence of a match propagates the request to the next level of cache and is known as a cache miss. Set-associative caches with $W \in \{4, 8, 16\}$ are the standard solution. L1 data caches commonly employ *Virtually Indexed, Physically Tagged* (VIPT) addressing, using a set index that is derived from virtual address bits, allowing set lookup to begin before the address translation completes, while tag comparisons still use the physical address to prevent aliasing across various processes. The other higher levels all use physical addressing.

An additional structural consideration is whether the hierarchy is inclusive or non-inclusive. Inclusive designs guarantee that if a line is present in a lower level of cache, then it will also be present in the higher levels as well. This is typically Intel's case, conversely AMD's Zen families use a non-inclusive design that does not implement this property.

Replacement policy. Upon a cache miss, if all the W ways in the target cache's set are occupied, the cache must remove, or evict, one of the existing lines to make room for the new one. The choice of victim line is decided by the replacement policy. The most common replacement policy is the Least Recently Used (LRU) replacement, which evicts ways based on usage history. Since this policy requires $\lceil \log_2(W!) \rceil$ state bits per set, it does not scale for highly associative caches. For this reason, variations of LRU replacement policies, like the tree-based pseudo-LRU from Intel's L3 cache, exist, but they still exploit temporal locality.

Address translation and the TLB. As already mentioned, cache indexing operates on physical addresses, but modern OSs make use of virtual address spaces to improve security and memory usage for user space programs. But, since programs rely on virtual addresses, a translation from virtual to physical addresses must happen in order to index

the right words in main memory. To avoid paying this cost upon every instruction, the processor caches recent translations in the Translation Lookaside Buffer (TLB), a small, fully associative structure that maps virtual page numbers to physical address ranges. The TLB, however, introduces complications for context switching, since every process is assigned a virtual memory range, migrating the execution context from one process to another should invalidate the entire TLB, requiring a complete TLB flush on every switch and requiring a full page-walk cost for the first access to each page in the new context. To not have this behaviour, hardware manufacturers add some bits to every line of the TLB to identify the owner of the address range, so that the OS can perform switches without invalidating the entirety of the buffer.

Out-of-Order Execution

All the stalling behaviour due to hazards described at the end of Section 2.1.2 has a common consequence that is catastrophic for performance, when an hazard occurs, the last issued instruction will still occupy the pipelines' issue slot blocking every subsequent instruction from being executed, even if those instructions share no RAW dependency with the stalled one. Out-of-Order (OoO) execution removes the artificial constraint by allowing out-of-order issuing of instructions, while still maintaining in-order retirement and in-order fetch and decode to fulfil ISA specification.

The Reorder Buffer. The Reorder Buffer (ROB) is a circular buffer that acts as a central bookkeeping authority for out-of-order execution. To keep track of instructions, it uses two pointers: the HEAD, which points to the oldest still in-flight operation, and TAIL, where new entries are allocated if the buffer is not empty. Each new entry is allocated in program order at the decode stage and holds one entry for each in-flight instruction. Entries record the decoded instruction, their execution status, the register needed and keep space to hold the result until retirement. This last property is what guarantees program order, regardless on when each instruction actually executed.

Register renaming. Dispatching instructions out-of-order is likely to generate a high number of data hazards, either due to true data dependencies (RAW), which the processor is forced to wait for, or due to register reuse. In the latter case, execution faces either WAR hazards or WAW hazards, both of which do not constitute real dependencies. To eliminate false dependencies, thus preventing these hazards, processors create mappings from architectural registers, like RAX or RSP in x86-64, to a larger set of microarchitectural registers. Then, upon instruction decode, the destination field inside the ROB is set to a newly assigned physical register, thus breaking any false dependency.

Reservation stations and dispatch. Once renamed, instructions are placed in the reservation stations, where they wait until all their source operands are available. And they become available either when a preceding instruction broadcasts its result on the common data bus at the end of execution, or when they are read from the physical register file if the result has already been committed. As soon as all source operands are ready, the scheduler selects the instruction for dispatch to the appropriate execution unit. This selection is opportunistic: a young instruction with all operands ready will be dispatched ahead of an older instruction still waiting on a cache miss, which is precisely the mechanism that allows independent work to proceed during long-latency stalls.

Additional microarchitectural buffers. Memory operations require additional buffering beyond the ROB because loads and stores interact with a cache hierarchy that itself has variable and unpredictable latency. Three structures are particularly relevant to this thesis. The *Load Buffer* keeps track of every load operation from the point of dispatch until the data is fetched and the instruction retires from the ROB. This allows multiple outstanding loads to be in flight. This buffer also creates the opportunity to implement Store to Load (STL) forwarding, a microarchitectural optimisation that allows younger loads to target addresses that older, uncommitted stores have already written to the *Store Buffer*, bypassing the memory hierarchy entirely. The store buffer, much like ROB entries, holds yet-to-be-retired store instruction results. The Line Fill Buffer (LFB) holds outstanding cache miss requests while a line is being fetched from a lower memory level. Rather than stalling the entire load pipeline on a miss, the LFB allows a small amount of concurrent misses to be in flight simultaneously. Once the requested data arrives, the LFB writes it into the cache and wakes the waiting load. Between the point of miss detection and the point of cache fill, the LFB entry holds the in-transit data in a transient, partially valid state.

Commit phase. Retirement, or commit, occurs when the instruction at the ROB head has completed execution, raised no unhandled exception, and all preceding instructions have already retired. At this point its result is written from the physical register file to the architectural register file, any pending store is drained from the store buffer to the cache, and the ROB entry is freed. Only at commit does the result become part of the architectural state visible to software and defined by the ISA. However, if a hardware exception occurs, such as a page fault or a divide-by-zero error, the processor has to perform a rollback, meaning the ROB and the pipeline have to be flushed to preserve the architectural state. Notice that the caches and other hardware buffers do not need this treatment since they do not effect the architectural state, so the microarchitecture does

not fully get restored to its previous state.

Branch Prediction

Branch operations introduce a structural problem for pipelines. By the time the processor has fetched and decoded an instruction, the next instruction already has to be loaded into the fetch stage. This however is not possible to do with absolute certainty, so the processor has to stall until the address of the next instruction is known, wasting clock cycles in the process. Note that the pipeline shown in Figure 2.1 is not indicative of modern pipelines, as they can implement many more stages of execution, so the actual throughput loss is substantial. And, given that conditional branches occur approximately once every five to seven instructions [29], a processor will spend most of its cycles waiting on branch resolution. To try recoup this loss of performance, processors employ predictive mechanisms to continue execution even during the resolution window. Branch prediction eliminates most of these stalls by committing to a fetch address before the branch is resolved, continuing execution along the predicted path, and correcting course only on a misprediction. A correct prediction costs nothing, a misprediction, however, costs a full pipeline flush and a rollback to the correct execution context, the same price as a stall, with the added cost of discarding the speculative work.

Static prediction. The simplest type of predictor requires no runtime state. A static predictor applies fixed heuristics without utilising any prior execution history. These heuristics are: always predict taken, always predict not-taken, most useful for simple if-then constructions or apply predict backward taken, predict forward not taken most used in looping blocks. These strategies are inexpensive and require very little hardware support, but they cannot adapt to the actual runtime behaviour of a program. Accuracy on general workloads is rather poor, but they are not the only types of predictors in modern chips.

The Pattern History Table. In contrast, dynamic predictors use the knowledge of past execution to maintain runtime state indexed by branch characteristics. The most widely deployed family is the *two-level adaptive predictor*. A Pattern History Table (PHT) stores one 2-bit saturating counter per entry, encoding four states: strongly not-taken, weakly not-taken, weakly taken, and strongly taken. The counter operates like a state machine that selects the prediction outcome depending on its values, incremented on a taken outcome and decremented on a not-taken outcome, making a single anomalous result insufficient to reverse a known stable prediction. In the simplest case the PHT is indexed directly by low-order bits of the branch's Program Counter (PC), giving each

branch address its own counter. More capable designs try to include global information to index the table, with the goal of improving prediction outcomes by correlating the outcomes of different branches. They introduce a Global History Register (GHR), a shift register that records the outcomes of the N most recently executed branches. Indexing the PHT by the `xor` of the GHR and the branch PC, the gshare scheme [25] allows the predictor to correlate a branch's behaviour with the execution context that preceded it. A loop-closing branch taken after a specific sequence of outer decisions can be predicted independently from the same branch reached via a different path.

The Branch Target Buffer. While the PHT predicts whether a conditional branch will be taken, it cannot determine its destination. To resolve this, the Branch Target Buffer (BTB) acts as a cache that maps branch instruction addresses to their predicted target addresses. During the instruction fetch stage, the processor queries the BTB using the current PC, on a cache hit, the buffer provides the predicted target address immediately, well before the branch is decoded or executed. BTB entries are indexed by a partial subset of the branch PC's bits, meaning that two branches at different virtual addresses can map to the same BTB entry and overwrite each other's predictions, this is a phenomenon known as aliasing. Critically, the BTB is shared across privilege levels and processes, and its entries carry no tag identifying which context wrote them.

The Return Stack Buffer. Function returns present a special case for indirect branch prediction. The target of a `ret` instruction is the return address pushed by the corresponding `call`, which is not generally predictable from the BTB alone. The Return Stack Buffer (RSB) is a dedicated hardware stack that mirrors the call stack in microarchitectural state. On each `call` instruction, the processor pushes the return address onto the RSB, while the `ret` pops the top entry and uses it as the predicted target. Under normal conditions this produces a near-perfect prediction for function returns at no misprediction cost.

Indirect branch predictors. Lastly there are branches whose target is computed at runtime, these are called indirect branches. Though, at first glance, they may seem a rare occurrence, these types of branches are heavily used by language constructs that make heavy use of virtual dispatch through vtables or compiler generated jump tables such as C++ or Java methods. These types of mechanisms are not well served by simple BTB, whose single entry is not designed to represent multiple changing branch destinations across its lifetime. Nevertheless, modern microarchitectures implement predictors that make use of long history vectors to predict which of the many possible targets a given

branch will take.

The shared predictor problem. Every structure described in this section is a shared microarchitectural resource with no architectural visibility. All are indexed by partial address bits or history patterns, none carry privilege-level or process-identity tags, and all are writable by any code executing on the core regardless of its privilege level. This means that any process can influence the predictive behaviour of any other process.

Simultaneous Multi-Threading

All of the previous microarchitectural optimisations dealt with improving the performance of a single independent instruction stream. This, however, is not indicative of modern workload requirements where hundreds of processes and software threads execute concurrently. With only the previously described strategy, when a thread encounters a long-latency stall and the ROB is exhausted of independent instructions, execution units sit idle regardless of scheduler depth. Simultaneous Multi-Threading (SMT) addresses this residual inefficiency by maintaining multiple independent hardware thread contexts on one physical core, allowing the scheduler to draw instructions from either thread to fill slots that would otherwise go to waste. This technology is also known as Hyper-Threading.

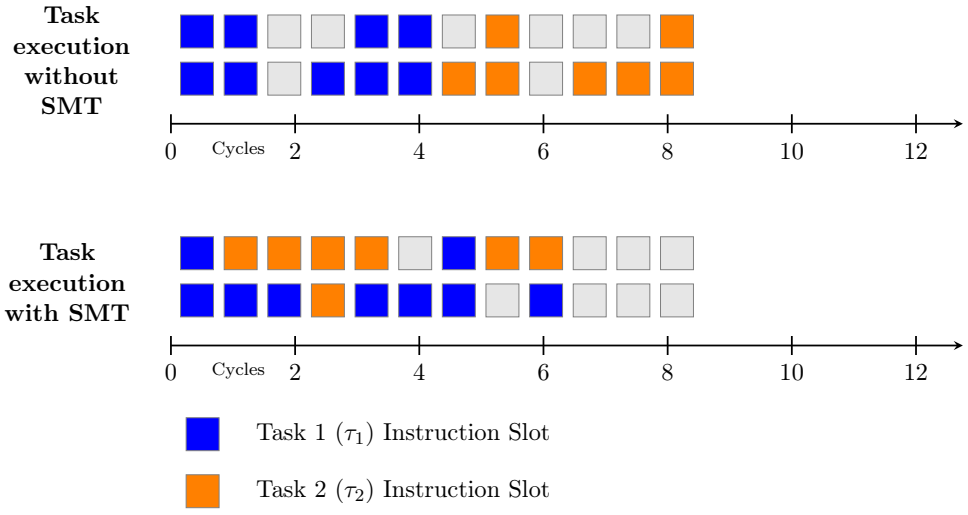

Figure 2.2: Issue slot usage with/without SMT

SMT should not be confused with multi-core parallelism. On a multi-core processor each core is a fully independent physical unit with its own execution hardware and private caches. In SMT, the additional threads are logical processors that share the physical execution resources of a single core. The duplication is deliberately minimal so, only the cheap per-thread state is replicated, while the expensive execution infrastructure is

shared.

Resource partitioning. Table 2.2 summarises what is private to each thread and what is shared between siblings on the same physical core.

Resource Type	Components
Private per thread	Program counter, Architectural register file, Reorder Buffer partition, Register rename state (RAT), Return Stack Buffer
Shared between threads	Execution units (ALU, FPU, load/store), L1 and L2 caches, Translation Lookaside Buffer, Load Buffer, Store Buffer, Line Fill Buffer, Branch Target Buffer, Pattern History Table

Table 2.2: Resource partitioning between SMT sibling threads on one physical core.

From the operating system's perspective, each SMT thread presents as an independent logical processor with its own architectural state. The scheduler interleaves instruction dispatch from both threads on a cycle-by-cycle basis, filling execution unit slots from whichever thread has ready work. The physical sharing is entirely invisible at the ISA level.

Speculative Execution

The underlying technique that makes prediction mechanisms and OoO execution so powerful is the ability, for an execution core, to process instructions even before hazards are resolved, this is called speculative execution. Instructions issued along this unconfirmed path are called transient instructions: they consume execution resources, access memory, and update microarchitectural state, but carry no guarantee of ever retiring. The interval between the dispatch of the first transient instruction and the detection of a hazard is called the *transient execution window*. Its duration is bounded by two factors: the ROB depth, or size, which puts a hard limit on how many instructions can be simultaneously in-flight, and the latency of the operation that resolves the speculation. For a branch whose condition is computed from an L1-resident value, the window may last only 15–20 cycles. However, for a branch whose condition depends on a value loaded from main memory, this window extends to the full duration of the memory access, encompassing potentially hundreds of instructions. This means that a transient window is not an inherent property of the hardware, but the result of the combination of the instruction flow and its executing environment.

Branch misprediction is the most common trigger, but it is not the only one. A transient window also opens when the processor encounters an instruction that will ultimately raise an exception, such as a load from an address the current privilege level is not permitted to access, and continues dispatching subsequent instructions before the fault is delivered. The window may also be opened by a load speculatively issued before an older store whose address has not yet resolved, thus operating on results from incorrect memory disambiguation. Each of these mechanisms produces the same result, the stream of instructions that executed during this window has to be discarded.

Rollback. When a misprediction or fault is confirmed, the processor executes the following sequence. First, the ROB is flushed from the offending entry onward, thus squashing all younger in-flight instructions. Then the register rename tables are restored from the pre-speculation snapshot maintained at the point of dispatch. And lastly, the program counter is redirected and fetch resumes from the correct address. The entire sequence completes in a bounded number of cycles and the architectural state is restored to what it would have been had the processor stalled and never speculated. This is crucial for maintaining the ISA contract intact. The transient instructions, from the programmer's model, never happened. However, the microarchitectural state is not restored. Cache lines brought into the hierarchy by transient loads remain resident after the flush. The Line Fill Buffer may retain data fetched on behalf of transient operations that were never committed. The store buffer may hold addresses and values written by transient stores. The PHT, BTB, and RSB retain updates made during the transient window. None of these are rolled back since the ISA does not require it, and restoring them would impose a measurable performance penalty for no benefit to architectural correctness.

2.2. Microarchitectural Side-Channel Attacks

The consequences of the discrepancies between the architecture and microarchitecture that arise from the optimisations described in 2.1.2 are what makes microarchitectural side-channel attacks like Transient Execution Attacks possible. The sections below will show how these types of attacks allow malicious attackers to steal information using microarchitectural buffers as covert channels and transmit data using timing discrepancies in execution behaviour. The attacks presented don't represent the full landscape of TEAs that have been created and serve as introduction for the techniques used by the attack families covered by this thesis.

2.2.1. Side Channel Attacks

A side channel is an unintended information channel that extracts information of a computation directly from hardware behaviour, bypassing software. Whereas a conventional attack targets the data exchanged through defined logical interfaces, a side-channel attack targets a physically observable quantity that correlates with secret internal state without being part of the intended output. Classic examples of these techniques are power analysis, first formalised by Kocher et al. [14], to recover cryptographic keys by analysing the power draw of the device as it processes different values, electromagnetic emission analysis or acoustic analysis. All of which require physical proximity to the target and use specialised measurement equipment, thus restricting their practical applicability.

Timing-Based Side Channels

Timing channels are a type of side-channel that can remove this last constraint since their observable quantity is elapsed time. This is a completely different type of threat model since time can be measured entirely in software without any physical probe or privileged execution environment. Timing observations are readily available in userspace, thus making them usable even from sandboxed and untrusted environments like virtual machines or JavaScript inside of a browser. The fundamental idea behind timing side channels is to rely on *data-dependent execution times* to deduce information about a running computation. This vulnerability arises when the execution path of a program diverges based on the value of a secret variable, causing a measurable difference in the time it takes to complete.

A classic example of this mechanism is the early-exit behaviour found in naive implementations of string comparison functions, such as standard password verification routines. Consider a routine that validates a user-inputted PIN against a secret target PIN by comparing characters sequentially from left to right. To optimise performance, the algorithm terminates and returns a failure status immediately upon encountering the first non-matching character. If the secret PIN is `"5291"` and an attacker submits `"0000"`, the function fails on the very first character, executing a minimal number of instructions. However, if the attacker submits `"5000"`, the function validates the first character successfully before failing on the second, resulting in a slightly longer execution time. By systematically iterating through candidate characters for each position and measuring these minute variations in latency, an attacker can determine exactly how many characters of their guess were correct. This transforms a brute-force search space from an exponential complexity to a linear one, allowing the secret to be recovered byte by byte. These types of attacks are extremely relevant for cryptographic algorithms with one of

the most significant attacks demonstrated by Kocher's seminal work on RSA [17] and creating the constant time execution paradigm.

Covert channels

The practical feasibility of a timing-based side channel depends entirely on the resolution of the available clock. The target differential, be it a cache hit that approximately takes 4 cycles versus a DRAM access at approximately 200 cycles spans roughly 50 nanoseconds on a 3 GHz processor. Any timer with resolution coarser than this simply cannot pick up the signal. On x86-64, the primary primitive is `RDTSC` (Read Time Stamp Counter), which reads the processor's monotonic timestamp counter directly into a register pair in a handful of cycles. Its serialising variant, `RDTSCP`, additionally waits for all preceding instructions to retire before sampling, preventing reordering due to out-of-order execution. The RISC-V equivalent is the `RDTIME` pseudo-instruction, which reads the platform-level machine timer. Both are unprivileged operations specified by their respective ISA and have extremely high resolution.

However, these are just measurement instruments. In order to recover a secret, an attacker needs more than just a clock, they need a way to encode secret-dependent information encoded onto a shared resource that is observable by the attacker i.e. a covert channel. This marks another big shift, from the other passive side channels, here the encoding of information is deliberate and structured. A *sender* encodes a secret value k by conditionally accessing a memory address that is a function of k, loading a specific cache line into the shared hierarchy. A *receiver* uses a timing primitive to determine which line was loaded and thereby recover k. In this case the communication medium is the cache state and the signal is the timing differential.

Cache-Based Attack Primitives

Cache-based attack primitives are a small category of timing-based attack primitives that can extract information from the cache state left behind by another, victim, process. Since all processes on the same physical core share their microarchitectural state a malicious attacker can manipulate the cache into a known initial state, wait for the victim to alter its state and then recover the data by observing the new state. The difference between the various primitives is in how they establish the initial state, what capabilities the attacker requires and how reliably the signal can be recovered. Four primitives appear recurrently in the transient execution literature, however the framework developed in this thesis relies on the first.

Flush+Reload. Flush+Reload [43] is the highest-accuracy cache primitive, but it requires a shared physical memory mapping between attacker and victim, a condition that can be satisfied in practice by shared libraries, shared file mappings, or pages deliberately shared between processes. The technique proceeds in three phases. In the *flush* phase, the attacker issues a flushing primitive targeting the cache line of interest, evicting it from every level of the cache hierarchy down to main memory. In the *wait* phase, the attacker yields execution so that the victim can run and, if the victim accesses the target address during this window, the corresponding line is reloaded into the cache. In the *reload* phase, the attacker times a load from the target address using a high-resolution timing primitive. If the gathered latency is significantly faster than a cache miss, then the attacker knows that the victim accessed the line. The timing differential between these two outcomes is the 50× gap established in Section 2.1.2, producing a binary signal with an empirical error rate below 5% under controlled conditions [43].

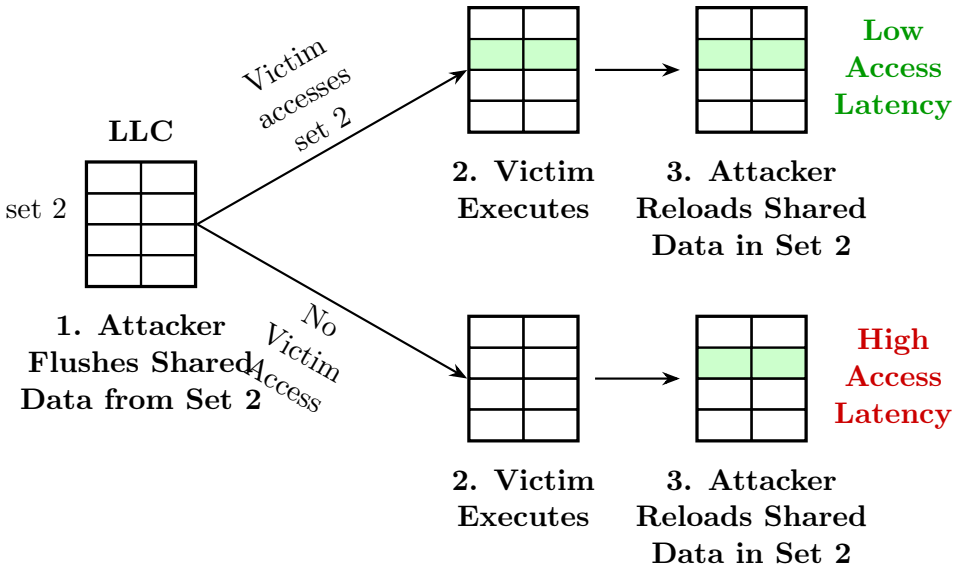

Figure 2.3: Flush+Reload cache attack

Other primitives. The other three primitives are described briefly for completeness. In Prime+Probe [22] the attacker fills a complete cache set with its own lines, waits for the victim to execute, then it finds which lines have been evicted. A slow reload indicates the victim accessed an address mapping to that set. This requires no shared state and relies on creating reliable eviction sets, which is a non-trivial problem. Evict+Time [27] measures total victim execution time before and after evicting a target set, thus a slower victim execution implies that the eviction was relevant to its working set. It is the noisiest of the four primitives and requires the attacker to be able to time the victim's execution as a whole. Finally Flush+Flush [7] exploits the observation that some flushing

operations, like the x86-64 CLFLUSH, executes measurably faster on non cached lines, so the measurement happens in the probe phase when the attacker issues a second flush and times it, producing a signal with extremely low noise and no cache misses in the observation step, making it resistant to detection via hardware performance counters.

2.2.2. Transient Execution Attacks

The covert channel model established in Section 2.2.1 requires a sender that encodes a secret into cache state and a receiver that recovers it through timing. In classical cache-based attacks, both roles are programs running legitimate instruction flows. The sender, by using memory access patterns, leaks sensitive information, and the receiver is the attacker that observes that pattern. Transient execution attacks remove this last constraint.The instruction stream used by the sender isn't a valid one anymore, but is a sequence of instructions that executes speculatively inside of a transient window opened by any speculative execution behaviour. This sequence of instruction is known as a transient execution gadget. To transmit data now the sender has to execute a gadget that will encode sensitive information into a microarchitectural covert channel for the attacker to access. Another big difference from standard cache-based side channel attacks is their scope, while normal attacks can only touch memory accessible by the normal instruction flow, TEA can access any memory that the victim's execution context can reach, this is because, by design, speculative execution will process instructions before the relevant access checks are performed.

The canonical encoding. The standard channel for encoding a secret byte into cache state is a transmission array: a 256-element array in which each element occupies its own cache line by being placed at a cache-line-aligned offset of $i \times cache\_line\_size$ bytes. Each field of this array represents each possible byte value. The transient gadget executes a single speculative load:

$$\texttt{temp} = \texttt{array}[\texttt{secret\_byte} \times \texttt{cache\_line\_size}] \tag{2.1}$$

This load brings exactly one cache line corresponding to the value of `secret_byte` into the cache hierarchy. After the speculative window closes and the architectural rollback occurs, however the cache line remains encoded in the cache. The receiver then probes all 256 entries of the array using Flush+Reload and the single entry returning a cache-hit latency reveals the value of `secret_byte`. Each transmission encodes exactly one byte, and the process is repeated for each byte of the target secret. Under favourable conditions

a Flush+Reload channel of this form achieves throughputs on the order of hundreds of kilobytes per second [43].

Security boundaries. Modern computing systems enforce data confidentiality through a hierarchy of security boundaries, that is, hardware and software mechanisms that restrict which memory a program can access at any given time. They are determined by the execution environment where the program is currently running in. At the lowest level, processor privilege rings separate unprivileged user space processes (ring 3) from the operating system kernel (ring 0). A user space load targeting a kernel virtual address is intercepted by the hardware and results in a fault before any data is returned. Above this, virtual memory gives each process a private address space enforced by the Memory Management Unit (MMU): one process cannot read another's memory regardless of privilege level. In virtualised environments, a second layer of address translation isolates guest VM memory from the hypervisor and from co-resident guests. Hardware enclaves such as Intel SGX add a further boundary that protects enclave memory even from a compromised operating system. Each of these mechanisms operates at the architectural level: they govern what *committed* instructions are permitted to access and produce, enforcing their restrictions at the point where results become visible to software. Transient instructions execute before that point.

Meltdown-Type Attacks

The meltdown attack family exploits *delayed exception delivery*. When a processor issues an invalid load, for instance because the target address is inaccessible by the current environment privilege level, it does not raise an exception immediately. Due to OoO, subsequent instructions continue to execute transiently using the result of the faulting load before the exception is delivered and the ROB is ultimately flushed. This delay for the exception delivery is what defines the transient execution window of meltdown-type attacks.

Meltdown. The original Meltdown attack [21] (CVE-2017-5754) targets the user/kernel privilege boundary. Here a user space process will issue a load from a kernel address, thus ultimately resulting in fault due to insufficient privileges, but not before the Out-of-Order engine has transiently loaded and encoded the result into the cache. The attacker then suppresses the fault using either `SIGSEGV` signal handler, which let's a program handle the error without exiting, or using Intel TSX transactional memory, which aborts the transaction on a fault and resumes at a specified recovery point. In both cases, the transmission array will retain the cache trace of the transient encoding, so it's just a

matter of recovering with one of the primitives described in Section 2.2.1

Foreshadow and L1TF. Foreshadow [35] (CVE-2018-3615/3620/3646) exploits the same delayed-fault window through a distinct mechanism. When a page table entry has its Present bit cleared, the page walker still reads the physical address field left in the entry and the load pipeline transiently accesses the L1 data cache at that address before the fault is delivered. If the target physical address currently holds a cache line from a different security domain — such as an SGX enclave, the hypervisor, or another process — the transient instruction receives that data and can encode it into the cache hierarchy through the standard transmission array. This breaks SGX confidentiality because enclave pages reside in the L1 cache in plaintext during enclave execution, and an attacker with a crafted non-present PTE pointing to the same physical frame can recover the data without ever holding a valid architectural mapping. Level 1 Terminal Fault (L1TF) type attacks are particularly severe because they bypass hypervisor isolation in SMT-enabled environments, allowing a guest VM to leak data from a co-located sibling VM or the hypervisor itself.

Microarchitectural Data Sampling. The MDS family, comprising RIDL [36], Fallout [3], and ZombieLoad [31] is mechanically distinct from the fault-path attacks above. Rather than exploiting the result of a faulting load, MDS attacks sample stale or in-transit data from the internal CPU buffers described in Section 2.1.2: the Line Fill Buffer (RIDL), the Store Buffer (Fallout), and the Load Buffers (ZombieLoad). During a microarchitectural assist or fault, the CPU may service a dependent load from these buffers rather than from the cache or memory, transiently returning data that belongs to a different execution context. The critical consequence of this mechanism is that it exploits the SMT resource sharing established in Section 2.1.2. The LFB , Store Buffer, and Load Ports are shared between sibling threads on the same physical core. A thread executing in a lower-privilege context can therefore sample data placed in these buffers by a sibling thread executing at any privilege level without requiring any shared memory mapping or cross-privilege control flow. Any security boundary that shares a physical core is in scope, making MDS attacks effective against co-located VMs in a cloud environment

Spectre-Type Attacks

Spectre-type attacks exploit *misprediction-induced transient execution*. Here an attacker takes advantage of the transient window created by target branch resolution described in Section 2.1.2. To exploit these predictor, an attacker, has to force the CPU to make a misprediction in the victim's instruction flow, this is achieved by manipulating, or

training, the shared microarchitectural structures accordingly. Afterwards, a transmission gadget on this path will then encode a secret value into a shared cache state where, after the misprediction is corrected and the ROB is flushed, the attacker will recover it.

Spectre v1 — Bounds Check Bypass. Spectre variant 1 [15] (CVE-2017-5753) trains the conditional branch predictor to mispredict an array bounds check. The canonical gadget takes the following form:

Listing 2.1: Canonical Spectre v1 gadget.

```
if (x < array_size) {
    y = array2[array1[x] * CACHE_LINE_SIZE];
}
```

The attacker first mistrain the PHT by invoking the gadget repeatedly with in-bounds values of `x`, establishing a strong prediction that the branch is taken. The attacker then supplies an out-of-bounds value, making the CPU mispredict the branch to be taken and speculatively executing the body with the wrong `x` value. Afterwards, the secret value is encoded into a shared microarchitectural buffer as explained in Paragraph 2.2.2 and finally recovered by the attacker. Spectre v1 gadgets are dangerous because they are a very common programming pattern and can exist anywhere bounds-checked array access patterns appear, like in kernel system-call handlers, in SGX enclaves and in JIT-compiled JavaScript engines. The victim does not need to be a separate process, in fact the attacker can induce the gadget through any interface that passes attacker-controlled data into code containing the pattern. The mitigation is the insertion of a serialising `LFENCE` instruction between the bounds check and the dependent load, which prevents the speculative access by stalling execution until the branch is architecturally resolved, this effort can be automated with compiler support, but it may fail to recognise certain cases.

Spectre v2 — Branch Target Injection. Spectre variant 2 [15] (CVE-2017-5715) targets the Branch Target Buffer rather than the conditional predictor. The attacker poisons a BTB entry so that when the victim executes an indirect branch via a virtual function call, a function pointer dispatch, or a computed jump, then the CPU predicts an attacker-chosen target address rather than the legitimate one. The processor speculatively executes code at this address inside the victim's context and privilege level, encoding victim-accessible data into the shared cache. This is an inherently cross-domain attack. An unprivileged attacker can poison BTB entries that are subsequently consulted by kernel indirect branches, causing the kernel to speculatively execute an attacker-chosen gadget

with full kernel memory access. The same mechanism applies across VM boundaries when the hypervisor and guest share a physical core.

Spectre-RSB. Spectre-RSB [18] exploits the Return Stack Buffer established in Section 2.1.2. On a context switch or privilege-level transition, the RSB is not flushed and retains return addresses from the previous execution context. When the new context executes a `ret` instruction, the CPU pops a stale, attacker-influenced address from the RSB and speculatively executes from it before the correct return address is resolved from the stack. This is mitigated by filling the hardware return stack on context, so popping the stack can only result in the correct instruction flow, this is also known as *RSB stuffing*.

Mitigations and the Evaluation Gap

The mitigation landscape. The attacks described in the preceding subsections have each been addressed by a corresponding mitigation. Table 2.3 summarises the primary deployed countermeasures, their delivery mechanism, and the attack vector each one targets. These mitigations are all software level hardening patches, but they work in different

Mitigation	Delivery	Target
KPTI	OS patch	Meltdown (CVE-2017-5754)
L1D_FLUSH	Microcode + OS	Foreshadow (CVE-2018-3615)
MD_CLEAR	Microcode	MDS (RIDL, Fallout, ZombieLoad)
LFENCE / SLH	Compiler	Spectre v1 (CVE-2017-5753)
Retpoline	Compiler	Spectre v2 (CVE-2017-5715)
eIBRS / IBPB	Microcode	Spectre v2 (CVE-2017-5715)
RSB stuffing	OS patch	Spectre-RSB

Table 2.3: Primary deployed mitigations for the transient execution attack families examined in this chapter.

parts of the software stack. Microcode updates are distributed by processor vendors and installed by the operating system at boot time. They modify the behaviour of specific hardware structures without replacing the silicon. OS and compiler patches insert serialisation primitives, modify control flow patterns, or partition privilege-level page tables in software. The problem is that both types of mitigations are reactive and ad-hoc designed in response to a specific, publicly disclosed attack variant. Compiler patches indicate an even deeper issue, on vulnerable hardware, there are no guardrails protecting a potential victim if the program doesn't itself include the mitigations.

Declared versus actual security. When a mitigation is deployed, its presence is reported through OS-level status interfaces. On Linux, the directory `/sys/devices/system/cpu/vulnerabilities/` exposes a file for each known vulnerability. the contents report `Mitigated` when the relevant patch is detected [20]. Vendor security advisories and tools such as `spectre-meltdown-checker` similarly classify a system as patched once the corresponding software update is installed. These reports describe the *patch installation status*, not the *actual hardware behaviour*. A software mitigation blocks the specific exploitation path it was designed to close, but leaves the underlying speculative mechanism fully operational. A gadget that differs slightly from the patched variant, be it in its branch structure, its memory access pattern, or the predictor it relies on may remain exploitable on hardware that reports itself as fully mitigated. The iterative history of transient execution research confirms this: every major mitigation deployment has been followed by the discovery of bypass techniques targeting alternative predictor structures or novel speculation primitives [2]. This is the evaluation gap that the remainder of this thesis addresses. Chapter 3 surveys the existing approaches to post-silicon security assessment and quantifies their limitations. Chapter **??** then presents a architecture-agnostic, timing-only methodology that operates without proprietary specifications, providing an empirical and independently reproducible basis for assessing what a processor actually does rather than what it claims to do.

3 | Related Work

In this chapter, we will explore the literature examining Transient Execution Attacks detection. It will cover both state of the art architecture-aware and architecture-agnostic methodologies. This will highlight the gap in the current research that this thesis tries to address.Assessing whether a processor is vulnerable to Transient Execution Attacks requires checking whether a specific silicon, under specific conditions, leaves measurable microarchitectural traces that an unprivileged attacker can recover. The literature approaches this question from two very different positions determined by the access to the processor's internal design specification: *architecture-aware* and *architecture-agnostic* approaches

3.1. Architecture-Aware Analysis

Architecture-aware analysis is conducted at the Register-Transfer Level level of the processor design. The RTL is the design of a processor described in terms of registers and the data operations between them, sitting between the high-level micro-architectural specification and the physical gate-level layout. This representation exposes the complete internal state of the processor at every cycle, including the speculative state that is invisible from outside: the contents of the ROB, the taint propagation through the load buffer, the sequence of predictor updates made by transient branches. Because of this deep knowledge of implementation, architecture-aware methods can reason over the complete state space, including microarchitectural buffers. The consequence of this completeness is a hard restriction on applicability. In fact, Intel, AMD, and ARM do not publish RTL for any of their commercial processor families. Access is restricted to internal design teams and, occasionally, to academic collaborators operating under non-disclosure agreements. Notable exception to this trend are RISC-V implementations like BOOM [44], where the RTL is freely available, often with very permissive licences.

3.1.1. Formal Methods for Vulnerability Detection

Architecture-aware vulnerability detection encompasses two methodologically distinct paradigms, the first, RTL fuzzing, it empirically searches for leakage instances by generating and executing large numbers of test programs. The second, formal verification, it proves or disproves security properties over the complete space of possible executions. Both have produced significant results in detecting transient execution vulnerabilities at the design stage, and both face the same fundamental ceiling when applied to independent security assessment.

RTL Fuzzing. RTL fuzzing uses automated test generation to a microarchitectural simulator in order to uncover unexpected information flows. The simulator exposes the all of the internal state of the design at every cycle. The fuzzer framework instruments this state with *taint tracking*, labelling any data derived from a designated secret input and following it through its propagation into the pipeline. A sequence that can cause tainted data to influence any observable output will be classified as a potential leak.

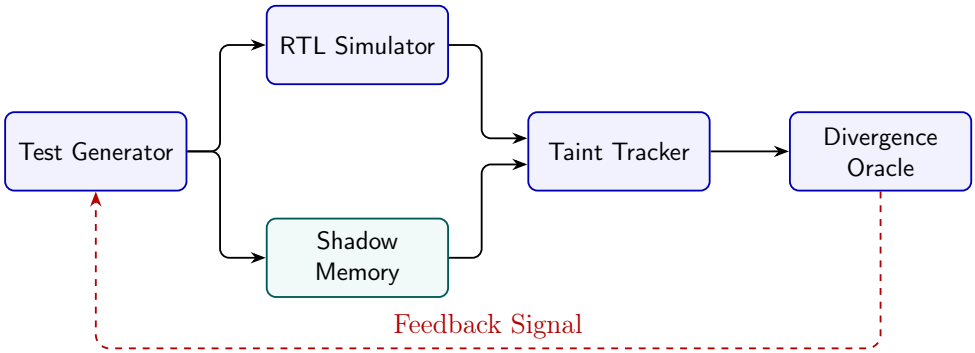

Figure 3.1: Dejavuzz RTL fuzzing detection cycle

Several frameworks have used this approach with increasing sophistication. *AMuLeT* [6] couples an RTL simulator with dynamic taint tracking that monitors propagation across pipeline stages, using taint diffusion across cache and memory components as a signal to prioritise inputs that exercise suspicious leakage paths. *DejaVuzz* [42] introduces a differential approach, rather than solely monitoring data flow, the framework isolates the secret data as an independent variable by executing identical instruction streams twice, using complementary secret values for each run. By comparing their resulting microarchitectural traces the system identifies discrepancies in the executions. Any observable divergence is then classified as a potential leak. More recently, reinforcement learning has been applied to guide the test generation policy, for instance the approach of Lai et al. [19] trains an agent whose reward signal is proportional to taint diffusion or cache-line

divergence, replacing hand-crafted mutation heuristics with a learned search strategy that focuses exploration on the regions of the state space most likely to contain speculative leakage.

Two persistent challenges constrain RTL fuzzing in practice. The first is state-space explosion. A modern out-of-order pipeline with a large ROB, a multi-level cache hierarchy, and hundreds of physical registers exposes a state space too large for exhaustive exploration. Practical implementations maintain full simulation fidelity only within the speculative execution window, while coarsening the model outside it by using more performant, software-based emulators such as gem5 [23] or Unicorn [4] to identify promising instruction sequences before committing to the more expensive RTL simulation for confirmation. The second challenge is the oracle and exploitability gap. Taint tracking reports any information flow from secret to observable state, but not every such flow is practically exploitable. A tainted cache access that occurs too late in the transient window, or whose timing differential falls below the measurable noise floor, will be flagged alongside high-confidence exploitable leaks. Determining which detected flows correspond to real attacks limits the degree to which RTL fuzzing can be fully automated.

Formal Verification. Formal verification approaches the same problem through mathematical proof rather than empirical search. Rather than generating test cases and observing outcomes, formal methods model the hardware and its interaction with software as a set of verifiable logical properties and either prove that those properties hold for all possible inputs or produce a counterexample demonstrating a violation. In this context a violation is represented by a distinguishable microarchitectural trace.

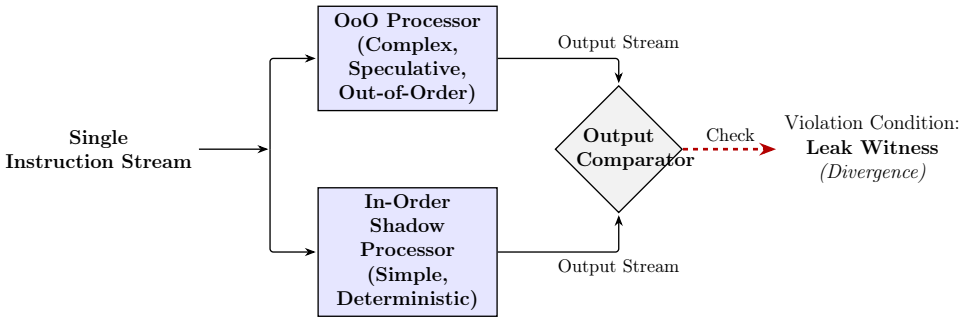

Figure 3.2: Formal RTL Contract Shadow Logic

The dominant formalism for hardware-software security contracts is exemplified by the *Contract Shadow Logic* framework of Tan et al. [33]. The approach augments an out-of-order RTL design with a parallel in-order shadow processor that executes the same instruction stream without any speculative behaviour. Here a contract is defined as a set of execution constraints that software must satisfy. The formal claim is that any software

conforming to the contract will produce identical observable output on both the OoO and the in-order shadow implementations. Any divergence that can be proven unreachable under the contract is a verified security guarantee, inversely any divergence that can be exhibited is a verified leak. This bidirectional result is the distinguishing strength of formal approaches over fuzzing. Information flow analysis extends this paradigm using type systems and model checking. *Spectector* [8] applies symbolic execution over a speculation primitive, so each branch point spawns a speculative execution path alongside the concrete one, and the analysis checks whether the union of microarchitectural observations across all paths reveals any secret-dependent divergence. Garfinkel et al.'s *Checkmate* [34] takes a complementary direction, it models the processor as a memory consistency specification and checks whether attacker-defined microarchitectural access patterns can be realised by any input program. Patterns that are realisable yield concrete PoC programs as witnesses, otherwise they are excluded from the threat model.

Formal methods offer the strongest security guarantees, they provide proof of the absence of leakage, not merely the absence of found instances. Fuzzing, by contrast, can only confirm that a leak exists when one is found. For security validation and hardware compliance, formal guarantees are therefore qualitatively superior. Their practical limitation is their scalability: formal verification of a full OoO pipeline with speculative execution, store forwarding, and multi-level cache interactions is very computationally intensive, and most deployed tools require manual abstraction of the design to reduce the complexity. The verification results are then valid only for the abstracted model, not necessarily for the concrete implementation. Both RTL fuzzing and formal verification share a significant limitation, even where RTL is available, the correspondence between the verified design and the manufactured chip is not guaranteed. Post-synthesis optimisations, timing closure adjustments, undocumented microcode behaviour, imperfections and errors may introduce discrepancies between the model and the physical device. The security guarantees produced by architecture-aware analysis are therefore conditional on the fidelity of the RTL to the silicon, a correspondence that cannot be verified without access to both.

3.2. Architecture-agnostic Analysis

Architecture-agnostic analysis focuses on evaluating the silicon using only externally observable behaviour like timing measurements, Hardware Performance Counterss and OS-reported status flags. Thus removing completely the need for design specification to execute a proper evaluation. Two distinct sub-paradigms have emerged within this space, each targeting a different phase of the attack lifecycle. *Vulnerability detection*,

covered in Section 3.2.1, attempts to determine prior to any attack whether a platform is susceptible to known transient execution exploits. It is a pre-deployment assessment tool, asking whether a system should be considered at risk. *Runtime monitoring via hardware performance counters*, covered in Section 3.2.2, attempts to detect an ongoing attack during execution by identifying anomalous microarchitectural event patterns. It is a deployment-time intrusion detection mechanism, asking whether an attack is currently occurring.

3.2.1. Vulnerability Detection

Post-silicon vulnerability detection operates on fabricated hardware using only externally observable behaviour, making it the only paradigm applicable to the independent assessment of COTS processors. The dominant approach is exploit-driven. It executes set of known PoC programs on the target platform, and the outcome of each determines whether the platform is vulnerable to the corresponding attack. A complementary, lower-effort approach infers vulnerability status from the software and firmware metadata the processor exposes through OS interfaces, without executing any test program at all. Both share structural limitations that make them insufficient for the evaluation goal of this thesis.

Proof-of-concept replay. The core assumption of PoC-based assessment is that a processor is vulnerable to an attack if and only if a concrete exploit for that attack succeeds on the platform. The assessment procedure then is straightforward, collect a corpus of known PoCs for the attack families under study, execute each on the target hardware, and record the outcome. A successful data exfiltration confirms vulnerability. *GhostBuster* [24] is representative of the tools built on this paradigm. It aggregates PoC executables alongside OS and CPU status flags, executing both and cross-referencing their outputs to produce a per-vulnerability report for the target configuration. The tool queries the Linux vulnerability sysfs interface which holds the collection of per-CVE status files under `/sys/devices/system/cpu/vulnerabilities/` and combines the kernel's self-reported mitigation status with the empirical result of executing the corresponding PoC, flagging discrepancies between the two. Spectre-meltdown-checker [20] takes a lighter approach, relying primarily on software metadata read from CPU feature flags in `/proc/cpuinfo`, inspects the installed microcode version, queries the kernel configuration, and produces a per-CVE verdict based on whether the conditions required for each mitigation are present. This approach has quite obvious limitations. The first is *PoC completeness*. The quality of an assessment is determined by the exploit corpus it is built on. Newly discovered

variants, attack primitives not yet reduced to a public PoC, or gadgets specific to an uncommon codepath are entirely invisible to replay-based evaluation. The historical record of transient execution research illustrates the severity of this constraint: the systematic evaluation of Canella et al. [2] identified dozens of attack variants that existing tools did not cover at the time of publication, and subsequent disclosures have extended this pattern consistently. Each major mitigation deployment has been followed by the discovery of bypass techniques that existing PoC corpora did not anticipate. A platform that successfully resists every known PoC cannot therefore be inferred to be secure against the next one.

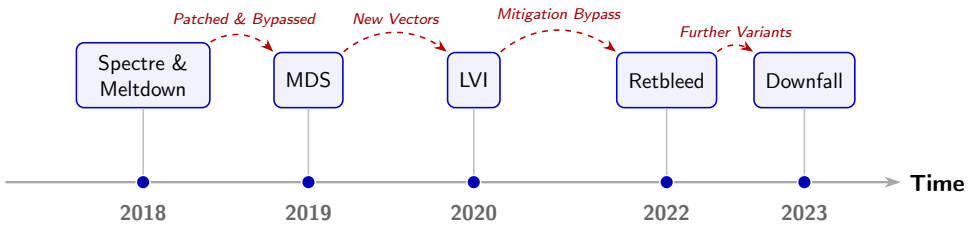

Figure 3.3: The PoC Completeness Argument: Disclosure → Mitigations → Bypass Variants

The second limitation is *binary classification*. PoC replay produces a binary verdict per CVE, it's either vulnerable or not-vulnerable. It provides no continuous measure of how large the transient window is on a given processor, how strong the resulting cache timing signal is, or how much the attack's feasibility depends on environmental conditions. The last limitation is *platform dependency*. PoCs are typically written for a specific kernel version, compiler toolchain, and ISA family. A Spectre v1 exploit developed for x86-64 Linux cannot be directly deployed on an ARMv8 system or a RISC-V platform without complete reimplementation.

3.2.2. HPC Monitoring

Hardware Performance Counters (HPC) are fixed-function registers present in all modern processor families that count microarchitectural events over a measurement interval, measurements include cache hits and misses at each cache level, branch predictions and mispredictions, TLB fills and flushes, speculative instructions issued and subsequently squashed, retired instruction counts, and many others. HPC-based security monitoring repurposes these performance diagnostics as an anomaly detection mechanism: if the microarchitectural event profile of a running program deviates from a baseline model of its benign behaviour, the system flags the deviation as a potential attack in progress.

Mechanism and deployment. The standard HPC monitoring pipeline is a per-workload baseline created by profiling execution under known-benign conditions and recording the distribution of selected HPC event counts over time. At runtime, the monitoring system samples the same counters at a fixed interval and compares the observed counts against the baseline. Anomalies are detected when these counters stray from the expected golden output. Paccagnella et al. [9] demonstrate this approach for detecting Spectre and Flush+Reload attacks using a combination of cache miss rates, branch misprediction events, and speculative rollback counts as features, achieving detection rates sufficient for practical deployment in a hypervisor monitoring context. More sophisticated implementations apply machine learning classifiers trained on HPC feature vectors to separate attack signatures from benign noise, improving discrimination on workloads with high natural variability.

Fundamental limitations. The primary limitation of HPC monitoring is that it is evadable by design. *Vizard* [32] demonstrates a systematic evasion strategy by interleaving attack activity with calibrated benign resource usage at rates sufficient to mask any malicious activity. The second limitation is environmental interference. HPC readings are sensitive to all concurrent activity on the system. Building a reliable detector therefore requires either a controlled single-tenant environment, which contradicts the multi-tenant deployment context where these attacks are most dangerous, or a statistical model sophisticated enough to separate attack signal from system noise at very low signal-to-noise ratios, a requirement that becomes harder precisely as the attacker reduces attack amplitude to evade detection. HPC monitoring is therefore a runtime intrusion detection tool, not a vulnerability assessment tool. It cannot characterise whether the underlying hardware is vulnerable, how large the speculative window is, or whether a given mitigation is effective.

3.3. Gaps in Existing Research and Novel Contributions

The state of the art surveyed in this chapter addresses transient execution vulnerability assessment from complementary angles. Their limitations, however, are structural consequences of each approach's foundational assumptions rather than incidental weaknesses that better engineering would resolve.

Research gaps. Architecture-aware analysis provides the strongest available security guarantees, but suffers with portability and availability for independent evaluation. Also

their reliance on RTL simulators may hide crucial timing-dependent behaviour crucial for TEAs. Post-silicon vulnerability detection operates on real hardware without design access, but is constrained to the space of known attacks. The iterative disclosure record demonstrates that this is a permanent feature of PoC-dependent evaluation. HPC monitoring detects ongoing attack activity rather than characterising hardware susceptibility, and is evadable by design, thus an attacker can normalise observable event counts to a benign baseline while continuing to exfiltrate data at a reduced rate. Taken together, these gaps define a single missing capability. No existing approach is simultaneously accessible to independent analysts without proprietary design access, grounded in empirical measurements on fabricated hardware and able to produce a continuous measure of microarchitectural exploitability rather than a binary verdict.

Novel contributions. This thesis addresses that gap through two contributions. The first is a *architecture-agnostic testing framework* that infers the presence and properties of transient execution effects purely from timing observations. The framework operates on fabricated silicon as delivered, making its results applicable to any commercially available processor. The second is a *novel security metric* for quantifying leakage of microarchitectural features. Providing a continuous metric for defining the exploitability of components that enable transient behaviours. Thus providing levels of vulnerability for the TEA families on which they rely upon. Chapter 4 details the design and implementation of each of these contributions. Chapter 5 presents the empirical results across the target platforms and evaluates the framework against the attack families identified in Chapter 2.

4 | Proposed Security Verification Framework

This chapter presents TEA-checker [1], a architecture-agnostic, software-only testing framework designed to address the evaluation gap identified in Section 3.3, namely the absence of an accessible, reproducible methodology to assess a processors' vulnerability to Transient Execution Attacks without relying on vendor claims. TEA-checker operates exclusively at the ISA level, treating the processor as an opaque device and making no assumptions about its internal microarchitectural design. Security properties are inferred entirely from timing measurements gathered through the standard architectural interface without elevated privilege, hardware debug access, or proprietary specifications. TEA-checker uses two categorises of tests to make its security claims:

- Feature tests are designed to probe a single feature in isolation to assess its presence and their associated parameters. In this context a feature represents a physical implementation of a microarchitectural component, ranging from cache hierarchies, branch predictors, execution buffer sizes and latencies, data forwarding paths and execution strategies, and establishes the building blocks upon which TEA are built.

- System interaction tests, in contrast, are designed to measure the behaviour that emerges when multiple features are simultaneously active during speculative execution. Rather than isolating a single structure, they orchestrate speculative windows and covert channels to observe whether a secret-dependent memory access completes transiently and leaves a recoverable microarchitectural trace. These tests directly quantify properties that determine attack feasibility, such as speculative transient window size and potential security boundary breaches.

Feature tests are prerequisites for system interaction tests. The framework cannot meaningfully report on Spectre v1 [15] speculative window size if it has not first confirmed that the cache covert channel and the predictor are both present and measurable on the target platform. The module system described in Section 4.3 formalizes this dependency. Once measurements are collected, an offline analysis step translates raw timing data into a structured security assessment. Every test in TEA-checker is built around the same

fundamental observation: a microarchitectural feature that influences execution timing will produce a measurable difference between two code paths that are structurally identical except for the presence or absence of that feature. This timing difference δ is the raw observable from which all feature detection and severity scoring derive, its validity as a measurement depends entirely on the structural identity of the two paths. For each feature test, the timing differential δ is normalised into a continuous severity score $S_f \in [0, 1]$ that reflects how strongly the feature manifests on the target platform. A score of zero indicates that the feature is either not present or is effectively indistinguishable from normal execution, while a score near one indicates that the feature is clearly detectable and contributes to the platform's transient execution attack surface. Similarly, system interaction tests use the results of the manifested features to give a continuous severity score $S_a \in [0, 1]$ that represents how difficult it is to generate and encode data into a transient execution window. In addition to this score, system interaction tests also report the security boundary that an interaction is able to cross. Once the previous sets of tests are concluded, the results of both per-feature score and feature interaction tests are aggregated per class of TEA family, and these scores represent the likelihood of a type of attack manifesting on the processor under study, provided that the target processor exhibits the required microarchitectural components for its functioning.

This last measure is what differentiates TEA-checker from binary PoC replay, which can only report whether a known exploit succeeds on a specific platform. TEA-checker instead characterises the underlying microarchitectural behaviour of the processor, making it possible, for a system, to receive a non-zero attack-family score even when no public proof-of-concept for that family exists or when all the known attack vectors are mitigated. Section 4.1 describes the overall architecture of TEA-checker, covering the execution runners, the kernel-level probe interface, the module dependency system, and the measurement infrastructure. Section 4.2 defines the validation methodology and the scoring model through which timing measurements are translated into assessments of processor resilience. And, lastly, Section 4.3 details the design and general structure implemented by each individual microbenchmark presented thereafter.

4.1. Framework Design

The architecture-agnostic and portability requirements stated in the chapter introduction translate into three concrete engineering constraints on the design. Firstly, tests must be executable across different privilege levels, those being, user space, kernel space and bare metal simulation. This ensures a comprehensive picture of security in various types of execution environments. To aid portability, the tests are mostly built with a set

of architecture-agnostic primitives — wrappers for timing reads, cache-line flushes, memory barriers, and serialisation that abstract over the ISA-specific implementations listed in Table 4.3. Porting to a new architecture requires only reimplementing these wrappers; the test logic itself remains unchanged. Currently TEA-checker supports x86-64 and RISC-V. Lastly, because timing measurements are inherently sensitive to interference from concurrent processes, the framework enforces execution isolation through CPU pinning and interrupt suppression in order to maximise the fidelity of each measurement. Figure 4.1 maps the relationships between the components described in this section. The *Orchestrator* acts as the framework's entry point, loading each test module and the kernel-level probe at startup. Test modules are packaged as self-contained units, each comprising a measurement function and a manager that handles its execution lifecycle and first-pass diagnostics. The *Probe Module* runs as a minimal kernel driver that bridges the privilege gap by exposing ring 0 primitives for privileged memory and TLB manipulation through a controlled `ioctl` interface available to the user-space runner. The Kernel and Simulation Runners do not require this bridge, as they already operate at sufficient privilege levels to perform these operations directly.

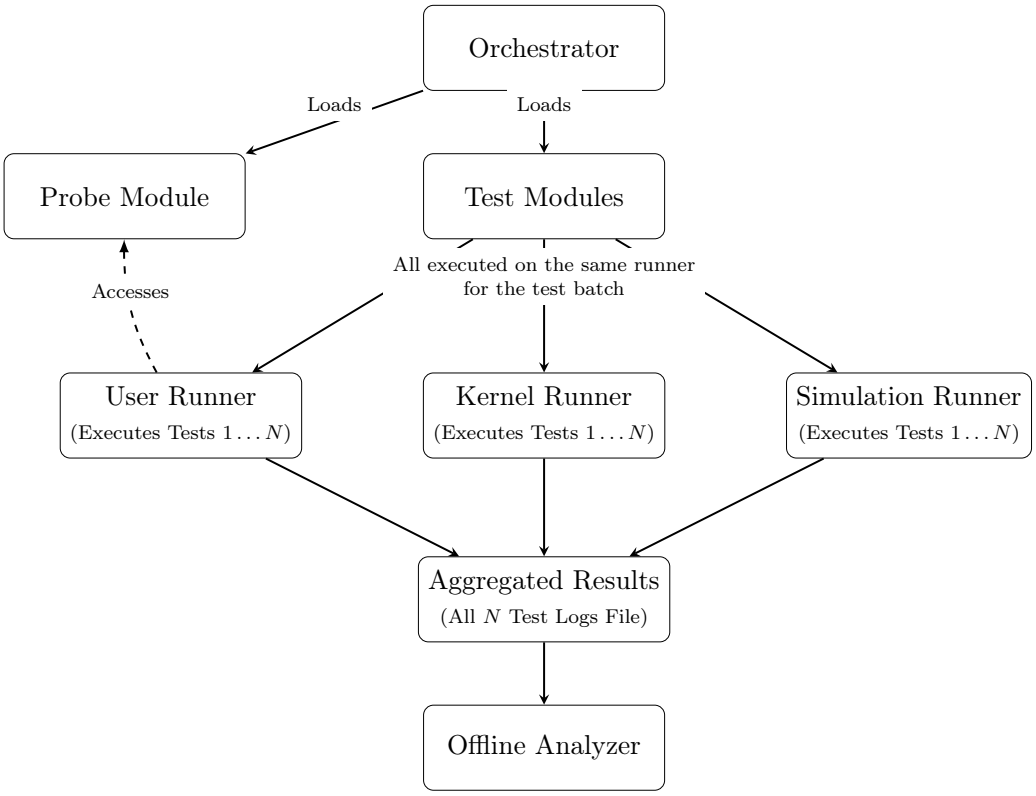

Figure 4.1: TEA-checker component architecture.

Three *runners* provide the execution backends, each targeting a different privilege environment: user space at ring 3, kernel space at ring 0, and bare-metal simulation in

machine mode. The orchestrator selects one runner for the evaluation and loads the same test module into it.

All runner outputs, such as timing data, status flags, and per-runner metadata, are aggregated into a single binary file. The *Offline Analyzer* is a separate component that ingests this file and produces the full structured security assessment defined in Section 4.2, including the per-feature scores, attack-family aggregates, privilege-level classifications, and diagnostic plots. Decoupling collection from analysis lets the scoring model be updated and reapplied without re-running any experiments, and allows measurements gathered on the target machine to be evaluated on a different system entirely.

4.1.1. Execution Runners

The same microarchitectural feature does not necessarily produce an identical timing signal across all execution contexts. OS scheduling noise and software mitigations active at the kernel level all influence both the strength and the reliability of the measurements. Rather than testing exclusively in one environment, TEA-checker compiles each measurement function against three execution runners, allowing the same test to be evaluated in user space, kernel space, and bare-metal simulation. Rather than maintaining three separate codebases, TEA-checker compiles the single testing module alongside the runner implementation selected in the compilation step.

Table 4.1: Execution runners and their properties.

	User	**Kernel**	**Simulation**
Privilege level	ring 3	ring 0	machine mode
Execution noise	high	low	minimal
Target context	unprivileged attacker	privileged code	embedded / IoT
CPU pinning	`sched_setaffinity`	`smp_call_function_single`	hart scheduler

User-space runner. This runner is the most representative of a complete system and of the most common threat model. As shown in Table 4.1, measurements at this level incur the highest noise due to OS scheduling and active software mitigations. Modules can be compiled either as shared libraries loaded dynamically or as standalone executables spawned through `execve`. The distinction matters because certain Spectre v2[15] mitigations, such as Indirect Branch Predictor Barrier (IBPB) [16], flush the indirect branch predictor only on context switches between certain kinds of memory mappings. Results collected at this privilege level reflect what a realistic unprivileged attacker can observe without any exploitation of the operating system itself.

Kernel runner. As detailed in Table 4.1, the kernel runner executes the measurement function at ring 0, where interrupt suppression and dedicated CPU pinning minimise OS-induced noise. This runner serves a dual purpose: it establishes a cleaner baseline against which user space measurements can be compared, revealing how much of the observed signal is genuine microarchitectural leakage versus OS-induced noise, and it characterises the vulnerability from the perspective of privileged code, relevant to scenarios involving compromised kernel modules or hypervisors. The measurement function is compiled as a kernel module and invoked via `smp_call_function_single`, with interrupts disabled through `preempt_disable` and `local_irq_disable`.

Simulation. The simulation runner produces binaries targeting Chipyard RISC-V, an open source platform offering RTL of the BOOM RISC-V processors. As Table 4.1 indicates, this runner operates in machine mode with minimal noise, making it suitable for establishing clean reference measurements. Tests are cross-compiled as bare-metal programs linked against a minimal runtime to support multi-threaded test execution.

```
typedef struct thread_impl_t {
    void *(*entry)(void *);
    void *arg; void *retval;
    volatile thread_state_t state;
    usize requested_hart;
    struct thread_impl_t *next;
} thread_t;

void yeild(void) {
    thread_t *t = dequeue_thread(mhartid());
    if (!t) t = steal_any_thread(); else {
        t->retval = t->entry(t->arg);
        __atomic_store_n(&t->state, THREAD_FINISHED,
           __ATOMIC_SEQ_CST);
    }
}

void hart_scheduler_loop(void) {
    while (scheduler_running) { yeild(); }
}
```

Listing 4.1: Cooperative hart scheduler dispatch

This runner serves a fundamentally different purpose from the other two: it is designed to aid development of new testing modules by validating their implementation against an open source CPU whose RTL is publicly available, allowing results to be cross-referenced against the actual microarchitectural design.
Each hardware thread, or hart, polls a per-hart work queue and, when a queue is empty, the scheduler steals a thread from another hart's queue. This work-stealing discipline is necessary for SMT tests, in which a measurement thread must execute on a specific sibling hart while a sibling thread runs concurrently:

The Probe Module

User space tests may require access to operations that are only available at ring 0. To bridge this gap TEA-checker provides a *probe* kernel module that exposes a very limited set of instructions through `ioctl`. Commands fall into two groups. The first comprises cache and timing commands that give user space access to kernel-mapped memory ranges, enabling the framework to test for kernel-level security boundary bypasses. The second group deals with TLB manipulation at both the microarchitectural and the OS level. The Linux kernel employs various caching mechanisms to improve performance with page-table entries, so simply flushing the physical TLB often is not enough to completely remove stale cached entries without an accompanying OS-level cache flush.

Table 4.2: Probe module `ioctl` commands.

Command	Effect
`PROBE_GET`	Timed read from a kernel-mapped address
`PROBE_CACHE`	Bring a kernel pointer into L1
`PROBE_UNCACHE`	Evict a cache line via `CLFLUSH`
`PROBE_TLB_FLUSH`	Full TLB flush
`PROBE_TLB_FLUSH_PAGE`	Flush a single page translation
`PROBE_TLB_REMOVE_PTE`	Clear the Present bit on a page-table entry
`PROBE_TLB_RESTORE_PTE`	Restore a previously cleared PTE

Cache and timing commands. The first three commands in Table 4.2 provide user space tests with controlled access to the kernel's address space. `PROBE_GET` reads a word from a kernel-mapped physical address while recording the access latency via an appropriate timing primitive, enabling Flush+Reload measurements across the user to kernel space boundary. `PROBE_CACHE` and `PROBE_UNCACHE` allow the test to place a known kernel-mapped cache line into L1 or evict it, establishing a known initial cache state before a probe. These three operations suffice to test for cross-privilege leakage.

TLB manipulation commands. Meltdown-style attacks, as described in Section 2.2.2, may require the TLB to exploit the transient execution window that opens when a load targets a page whose Page-Table Entry (PTE) has the Present bit cleared. To test for this behaviour, the framework must be able to (i) clear the Present bit on a target page, (ii) restore it after the measurement, and (iii) flush stale cached translations from the TLB. The probe module provides exactly these three primitives, all of which require ring 0. The `PROBE_TLB_FLUSH_PAGE` command performs a full hierarchical flush of a single virtual address. Since the Linux kernel caches page-table entries, stale cached copies of a page-table may survive a microarchitectural flush, such as with x86-64's `INVLPG`, requiring also an OS-level flush of these caches.

```
void pte_clear_noflush(volatile char *page) {
    unsigned long addr = (unsigned long)page & PAGE_MASK;
    pte_t *ptep = walk_to_pte(addr);

    saved_pte = *ptep;
    saved_ptep = ptep;
    native_set_pte(ptep,
        __pte(pte_val(saved_pte) & ~(pteval_t)_PAGE_PRESENT));

    asm volatile("clflush (%0)" ::"r"(ptep) : "memory");
    asm volatile("mfence" ::: "memory");
}
```

Listing 4.2: Clearing the Present bit on a page-table entry

After the measurement completes, `pte_restore_noflush()` writes back the saved PTE value, flushes the cache line, and issues an `MFENCE` to ensure the modification is visible to subsequent page walks.

```
void pte_restore_noflush(volatile char *page) {
    if (!saved_ptep) return;

    native_set_pte(saved_ptep, saved_pte);
    asm volatile("clflush (%0)" ::"r"(saved_ptep) : "memory");
    asm volatile("mfence" ::: "memory");

    saved_ptep = NULL;
}
```

Listing 4.3: Restoring a previously cleared PTE

4.1.2. Timing and Serialisation Primitives

Accurate timing measurements are the foundation upon which TEA-checker is built. The signal's differential range can vary considerably, from hundreds of cycles for a cache hit versus a miss to the few cycles of overhead generated by instruction reordering. To achieve the most accurate measurement possible, the framework makes use of the primitives presented in Section 2.2.1. On x86-64 this is `RDTSC`[12], which reads a 64-bit monotonic counter directly into a register pair in a small number of cycles, and on RISC-V the equivalent is `rdcycle`, defined by the Zicntr extension [30]. Both are unprivileged operations available from any execution context. However, cycle-accurate counter readings are not enough to guarantee correct measurements. Out-of-Order execution, described in Section 2.1.2, can reorder instructions across counter reads, effectively querying the hardware clock even before the critical section has finished executing. To mitigate this problem, a memory barrier must precede each counter sample.

```
  serialise();
  memory_barrier();
  volatile u64 start = get_cycle();

  /* operation under test */

  serialise();
  read_memory_barrier();
  volatile u64 elapsed_time = get_cycle() - start;
```

Listing 4.4: Most common timing gadget

Table 4.3 lists the full set of primitives and their ISA-specific implementations. The cache-line flush operation warrants a note: on x86-64 `CLFLUSH`[12] removes a line from the entire cache hierarchy with a single instruction, while on RISC-V the `cbo.flush`[30] extension provides a direct equivalent only on cores that implement it. For the remaining cores the framework falls back to an eviction-set approach that displaces the target line by accessing a set of addresses that map to the same cache set, at the cost of slightly higher noise.

Table 4.3: Timing and memory primitives across the two target ISAs.

Primitive	x86-64	RISC-V
`serialise()`	`serialize/cpuid`	`fence.i`
`get_cycle()`	`rdtsc`	`rdcycle`
`memory_barrier()`	`mfence`	`fence iorw, iorw`
`read_memory_barrier()`	`lfence`	`fence ir, ir`
`cache_line_flush()`	`clflush`	`cbo.flush` / eviction set

Statistical Measurement Processing

Even with noise-reduction techniques, large latency spikes are still present in the data, whether from an OS scheduling decision or a system interrupt. For this reason, feature tests designed to measure sudden jumps in latency require post-processing steps to guarantee accurate readings. Before any feature detection can happen, the raw data is passed through a median filter which rejects spike outliers caused by the aforementioned interrupts without distorting the gradual trends that the detection algorithms need to observe. The detection algorithms used are all designed for observing and characterising a step change in the measured data.

Welch t-test [37]. For tests where the timing shift is more gradual, the framework applies the full Welch t-test. It compares the means of two adjacent windows and tests whether they differ significantly, without assuming equal variance between the two groups. Given two windows A (pre-change) and B (post-change) with sizes n_A, n_B, means μ_A, μ_B, and unbiased variances s_A^2, s_B^2, the test statistic is

$$t = \frac{\mu_B - \mu_A}{\sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}}}, \qquad \nu = \frac{(s_A^2/n_A + s_B^2/n_B)^2}{(s_A^2/n_A)^2/(n_A - 1) + (s_B^2/n_B)^2/(n_B - 1)}.$$

The pair of windows slides across the timing series until the position with the maximum $|t|$ exceeding a critical value derived from the t-distribution marks the detected change

point.

CUSUM [28]. The Cumulative Sum (CUSUM) algorithm is a change-point detector designed for persistent small shifts that accumulate gradually rather than appearing as a sharp step. Let μ_0 be the pre-change baseline mean estimated from the first m observations and let δ be the minimum shift magnitude of interest. The one-sided upper CUSUM is

$$S_0 = 0, \qquad S_k = \max\big(0,\ S_{k-1} + (x_k - \mu_0) - \delta\big),$$

with a change signalled at the first index k where $S_k > h$ for a decision threshold h. The change point is taken as the last index where the cumulative sum was at zero before the crossing. CUSUM identifies the exact parameter value at which the cumulative deviation from baseline first exceeds the threshold.

Relative jump detection. For features that produce a single clean binary transition, the latency is either clearly elevated or it is not; thus the framework applies a simpler test that checks whether the timing differential exceeds a fixed noise threshold.

4.1.3. Vulnerability Scoring Model

The raw timing measurements collected by the tests and validated by the mitigation methodology are not directly interpretable as security assessments. A latency differential δ of, say, 40 cycles for a branch predictor test says nothing on its own about whether a processor creates an exploitable transient execution window. The vulnerability scoring model is the component that translates validated measurements into structured security assessments, operating at three levels of increasing abstraction.

Feature severity. For each feature test, the validated timing differential is normalised into a continuous severity score

$$S_f = \frac{|t_{\text{without}} - t_{\text{with}}|}{|t_{\text{without}}| + |t_{\text{with}}|} \in [0, 1] \tag{4.1}$$

A score of zero indicates that the feature produces no measurable timing effect. This means that it is either absent or effectively indistinguishable from noise. A score near one indicates that the feature is clearly detectable and contributes to the platform's transient execution attack surface. For tests that detect a capacity parameter rather than a direct timing differential, the score is instead derived from the detected capacity relative to the expected hardware range. This is because the larger a hardware buffer is, the easier it is

to open and exploit the transient execution environment it creates. Some tests produce a binary result rather than a continuous S_f score. This is because their only meaningful interpretation needed by TEA-checker is to assess the presence of the aforementioned microarchitectural feature.

Attack-family score. The per-feature scores are aggregated into a score for each of the thirteen TEA families supported by the framework. The score of an attack family A is defined as:

$$S_a = \text{avg}_{f \in \text{req}(A)}(S_f) \cdot S_{cache} \cdot \prod_{m \in \text{mult}(A)} m, \tag{4.2}$$

where req(A) is the set of feature tests whose components the attack family requires to function, and mult(A) is the set of context multipliers that apply. The multipliers encode the observation that certain hardware features broaden the attack surface rather than constituting a required component. For example, the presence of SMT doubles the score of MDS-family attacks because the shared physical buffers give the attacker a cross-hyperthread observation channel. Table 5.1 lists the attack families with their required features and applicable multipliers.

Privilege-level aggregation. For each feature that can open a transient execution window, a corresponding system interaction test records the highest privilege-level bypass at which a load can cross a privilege boundary. The reported privilege levels are ordered as follows:

$$\texttt{excluded} < \texttt{not\_detected} < \texttt{same\_space} < \texttt{user} < \texttt{kernel} \tag{4.3}$$

A PHT score, recorded from user space, reported as `kernel` indicates that the load executed in a transient window induced by a branch misprediction can be detected from kernel memory, even after the window is closed. This represents the highest level security breach possible and, as such, is the highest reported by TEA-checker. On the opposite end of the scale, ignoring both `excluded` and `not_detected` since they don't report any vulnerability, there is `same_space` that indicates a load inside a transient window can reach any memory that is accessible in the same process.
Together, the three levels produce the continuous, context-sensitive exploitability measure described in the chapter introduction: not merely whether a processor is vulnerable to a named CVE, but how strongly each required component manifests and how severely it manifests. The full empirical results of applying this model to the target platforms are presented in Chapter 5.

4.2. Validation Methodology

Validating an architecture-agnostic timing framework presents a fundamental problem: the system being measured is not directly observable, so there is no independent oracle against which to check TEA-checker's output. A nonzero δ between two structurally identical code paths confirms just that something produced a timing difference, but it cannot tell what contributed in its production. The same observable timing could come from the microarchitectural mechanism that the test targets, from OS scheduling noise, a hardware prefetcher, or a power state transition. Without an independent reference, the framework's output is a raw measurement rather than a validated claim. The scores it derives from that output inherits the same uncertainty, so a validated measurement base is a prerequisite for a validated metric.

A direct check would require runtime visibility into the processor's internal microarchitectural state: pipeline occupancy, predictor entries, and cache hierarchy status. This is precisely the kind of Register-Transfer Level design access that Section 3.3 identifies as unavailable for COTS hardware: architecture-aware verification methods cannot be applied to platforms whose RTL is proprietary. Since the ground truth against which the tool's output would be compared is inaccessible by definition, indirect validation is the only available option. The indirect strategy is suppression-and-collapse. Each specific timing differential, that the TEA-checker claims its recorded from a specific microarchitectural feature, it's measured twice from the same test, but compiled in two configurations: a standard variant that leaves the claimed mechanism active, and a suppressed variant that disables it through a countermeasure. Both variants run on the same platform under identical conditions. If δ collapses under suppression, the claim is considered valid, Inversely if it persists, the measurement is erroneous and, thus, discarded.

This check is applied at three levels of increasing abstraction, each corresponding to a distinct type of claim that the framework produces. At the feature level, each test's δ must collapse when its own mechanism is suppressed. This validates that the test measures what it claims to measure. At the interaction level, system interaction tests mustn't show any windows or boundary breaks, when any prerequisite feature is suppressed. This validates that the emergent behaviour depends on those specific prerequisite features and not on an unknown confound. At the attack-family level, the aggregate score must drop to near-zero when all underlying features are suppressed. This validates that the scoring model propagates the effect correctly from individual measurements to high-level assessments. Table 4.4 reports the collapse ratios for four representative tests.

Test	Unmitigated S	Mitigated S
Cache	97.2%	3.8%
PHT	38.1%	0.1%
ROB	100.0%	0.0%
Stale code execution	46.4%	0.0%

Table 4.4: Collapse ratios under suppression for representative tests.

cache-line flush. The PHT test drops from 38.1% to 0.1%. The ROB test drops from 100% to 0% when out-of-order execution is constrained. The stale code interaction test drops from 46.4% to 0.0%.
All four follow the same pattern: a clear timing differential collapses to near-zero when the claimed mechanism is suppressed. The collapse is evidence that each test's δ measures what it claims to measure. With this foundation, the severity scores derived in Section 4.1.3 rest on verified measurements.

4.3. Microbenchmark Structure

This section describes the internal organisation of a test module and the lifecycle it follows from the moment the orchestrator loads it to the moment a validated result is produced. Understanding this structure is a prerequisite for reading the individual test descriptions in Section 4.3.1, each of which is presented as a concrete implementation of the pattern established here.

Module Anatomy

Each module lives in its own directory under `modules/<test_name>/` and consists of two main files with well-defined roles. The microbenchmark containing the measurement function implements the timing loop that the orchestrator compiles and executes against the target runner. This is the only file that gets compiled for each target runner, so the same benchmarking code can be executed as user space shared library, a user space program, a Linux kernel module, or a bare-metal target. The manager is the component that handles both the execution life-cycle of the microbenchmark and handles the first pass of evaluation, just enough to perform some post-processing on the readings and to determine if a test has passed or failed. This component is compiled into a shared library that exposes a `setup` function that can handle argument fine tuning the benchmark execution, and a `diagnostics` function that handles the aforementioned first-phase evaluation

The Lifecycle

Figure 4.2 shows the lifecycle of a module execution. The six steps are as follows.

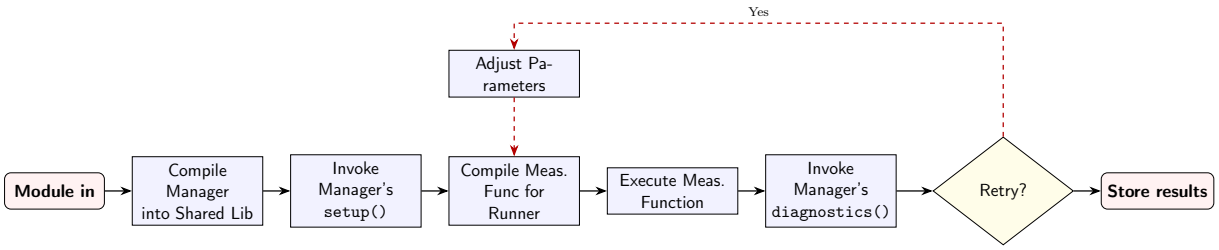

Figure 4.2: Life cycle of a testing module execution

Manager compilation. The manager is compiled into a shared library and loaded by the orchestrator. This step occurs once per module regardless of how many times the measurement function is subsequently executed.

Setup. The orchestrator calls `setup` passing the result structs of every declared dependency. This function serves as a pre-processing step that can determine the parameters for an upcoming measurement such as iteration count or the amount of instructions to generate to hide a transient window.

Measurement-function compilation. The orchestrator compiles `<name>_test.c` for the target runner. The manager's parameters are injected at compile time through preprocessor definitions, so each runner can access these parameters in the same way from the compiled binary.

Execution. The compiled measurement function runs on a core pinned by the runner and populates the result fields defined by the test itself. Each test averages multiple readings in order to minimise the execution noise. No dynamic parameter adjustment occurs during execution as all decisions are already taken in setup phase.

First phase evaluation The manager's `diagnostics` function interprets the raw result of the microbenchmark. If the measurement has not yet converged, this may happen when a binary search has not yet determined the speculative window size, the function returns `RETRY` and the cycle repeats with new parameters until `diagnostics` accepts the result or the retry limit is reached. This mechanism allows modules to implement binary search and parameter sweep algorithms without the orchestrator understanding their internal logic. This is also where the first pass of analysis happens by averaging and cleaning the

results when appropriate using the methodologies highlighted in Section 4.4, and finally determining whether a test has passed or failed.

Result storage. The accepted result is serialised to a binary file together with the module metadata. These are the values that the offline analyser will use to apply the scoring model described in Section 4.1.3.

4.3.1. Feature tests

Cache hierarchy

The data cache hierarchy 2.1.2 is the essential covert channel for every transient execution attack. This test measures the L1 load latency against the latency created by a cache miss, if this δ is statistically relevant the CPU will report to have cache memories. The first two results are fundamental for defining the bounds of every system interaction test.

Mitigation To hide this feature TEA-checker makes sure that every load experiences a cache miss by performing a `cache_line_flush` upon every load, thus collapsing the differential.

Pattern History Table

The Pattern History Table (PHT) 2.1.2 makes branch prediction possible. This test checks for branch prediction structures by training the same branch evaluations in either a predictable or random way. If this is enough to create a latency δ between the two execution paths, then it's evidence of the presence of a branch predictor.

Mitigation To hide this feature TEA-checker makes every branch result unpredictable, equalising the misprediction rate on both paths.

Branch Target Buffer

The Branch Target Buffer (BTB) 2.1.2 allows for indirect jump predictions, making the CPU potentially vulnerable to branch target injection attacks. This test trains the predictor on target `A` and compares the latency of calling `A` against calling it after training on a different target `B`. If the result shows that the former case is faster, then the BTB is found.

Mitigation To hide this feature every prediction has to manifest same result, to this end the test issues random indirect calls to other targets before each training call, preventing

the training pattern from persisting.

Load Address Predictor

Load Address Predictor (LAP) [13] is a microarchitectural mechanism that speculatively resolves a load's address by extrapolating stride patterns from prior executions of the same load instruction resulting in the opening of a transient window on data that was never architecturally accessed. This test detects the LAP by traversing an array via index indirection, first under random indices and then under a fixed stride of eight words, the predictor trained on the stride pattern completes the fixed-stride traversal faster. If timings diverge beyond a threshold, the predictor is considered present.

Mitigation To hide this feature TEA-checker replaces the fixed-stride path with a second random sequence, equalising both measurements.

Return Stack Buffer

The RSB (§2.1.2) is a dedicated hardware stack that mirrors the call stack in the microarchitectural state. Its depth is determined by stuffing over increasing recursion depths. Each entry in the recursive call chain pushes to the RSB, and as the chain unwinds, returns pop entries until the RSB underflows. When the recursion depth exceeds the buffer's capacity, the target's `ret` finds an empty entry and mispredicts, producing a latency jump; the depth at which this jump occurs defines the RSB size:

```
noinline void poison(int depth) {
    if (depth > 0) {
        poison(depth - 1);
    } else {
        serialise();
        memory_barrier();
        u64 start = get_cycle();
        target_func();
        serialise();
        read_memory_barrier();
        results.latency[depth] = get_cycle() - start;
    }
}
```

Listing 4.5: RSB depth detection via recursive poison chain.

Mitigation To hide this feature TEA-checker fills the RSB to capacity before every measured call (2.2.2), ensuring no underflow occurs.

Reorder Buffer

The Reorder Buffer (2.1.2) defines the upper bound on the maximum number of in-flight instructions and therefore the length of the transient execution window for every speculation-based (2.1.2) attack. Its capacity is measured by sweeping over an increasing number of `nop` instructions between two cache-miss loads [39]: for small N the OoO engine hides the first miss behind the second load, but when N exceeds the ROB depth the second load stalls and total latency jumps:

```
for (int n = 1; n <= 512; n++) {
    cache_line_flush(ptr1); cache_line_flush(ptr2);
    serialise();
    memory_barrier();
    u64 start = get_cycle();
    load(ptr1);
    for (int i = 0; i < n; i++) nop();
    load(ptr2);
    for (int i = 0; i < n; i++) nop();
    serialise();
    read_memory_barrier();
    results.latency[n] = get_cycle() - start;
}
```

Listing 4.6: ROB capacity sweep: two cache-missing loads separated by N NOP instructions.

Mitigation To hide this feature TEA-checker inserts a `serialise` after every NOP and load, forcing in-order execution and causing latency to grow linearly with N.

Translation Lookaside Buffer

The TLB caches virtual-to-physical address translations and is the target of Meltdown-family (2.2.2) attacks that exploit stale page-table entries. Its reach is measured by sweeping over an increasing working set of distinct 4 KiB pages, if the accesses are within TLB capacity, then every access will hit a cached translation, but beyond, it misses resulting in a full page-table walk and average latency jumps.

Mitigation To hide this feature TEA-checker issues a `tlb_flush()` and per-page `tlb_flush_page()` before every access, forcing a page walk regardless of working set size.

Line Fill Buffer

The Line Fill Buffer (§2.1.2) tracks outstanding cache miss requests while data is fetched from lower memory levels. Under certain fault or assist conditions, a dependent load can be serviced from stale data already resident in the LFB rather than from the correct address. Because this buffer is also shared between hyperthreads, this sampling can extend across the SMT boundaries. Its capacity is measured by sweeping over N concurrent cache-missing loads to distinct pages, within the LFB capacity misses are serviced in parallel and total time stays constant, but beyond it some misses serialise and total time jumps:

```
for (int n = 1; n <= MAX_BUFFERS; n++) {
    for (int i = 0; i < n; i++) cache_line_flush(pages[i]);
    serialise();
    memory_barrier();
    u64 start = get_cycle();
    for (int i = 0; i < n; i++) load(pages[i]);
    serialise();
    read_memory_barrier();
    results.latency[n] = get_cycle() - start;
}
```

Listing 4.7: LFB capacity sweep: concurrent cache-missing loads to distinct pages.

Mitigation To hide this feature TEA-checker inserts a `serialise()` between each load in the parallel set, forcing sequential miss handling and eliminating the concurrency gain.

Out-of-order execution

Out-of-Order (OoO) execution (§2.1.2) reorders independent instructions to hide the latency of a slow instruction. It serves as the prerequisite for Meltdown-type transient windows (§2.2.2). This test compares the execution time of a cache-miss followed by N independent `xor` instructions against the sum of their latencies measured in isolation. If due to an instruction reorder the combined sequence completes faster than the sum of its

parts, then the test passes.

Mitigation To suppress this feature TEA-checker inserts a `memory_barrier()` between the load and the independent computation, preventing reordering and matching the in-order expectation.

Pipelining

Superscalar pipelined execution (§2.1.2) determines how many independent instructions can be dispatched per cycle, bounding the rate at which speculative instructions can be issued during a transient window. This test compares three dependent arithmetic instructions executed sequentially against the same instructions interleaved with independent operations. A superscalar pipeline achieves higher throughput on the interleaved form.

Mitigation To suppress this feature TEA-checker inserts a `serialise()` between every arithmetic operation, draining the pipeline between instructions and eliminating any throughput gain.

Store-to-load forwarding

The Store to Load (STL) forwarding path allows a young load to obtain data from an older, uncommitted store without traversing the cache hierarchy. The test measures STL latency across a 64×64 grid of alignment offsets [40], so that when addresses share the same microarchitectural alignment the forwarding fast path activates and the load completes measurably faster, thus revealing the underlying mechanism:

```
for (int src = 0; src < 64; src++) {
    for (int dst = 0; dst < 64; dst++) {
        serialise();
        memory_barrier();
        u64 start = get_cycle();
        *(volatile u64 *)(dst_ptr + dst) = 0;
        load(src_ptr + src);
        serialise();
        read_memory_barrier();
        results.heatmap[src][dst] = get_cycle() - start;
    }
}
```

Listing 4.8: STL forwarding alignment sweep: 64×64 offset combination heatmap.

The resulting heatmap is analysed for the characteristic diagonal of fast forwarding where load and store offsets match, together with the latency penalties for partial overlaps and cache-line boundary crossings that reveal the microarchitecture's forwarding unit design.

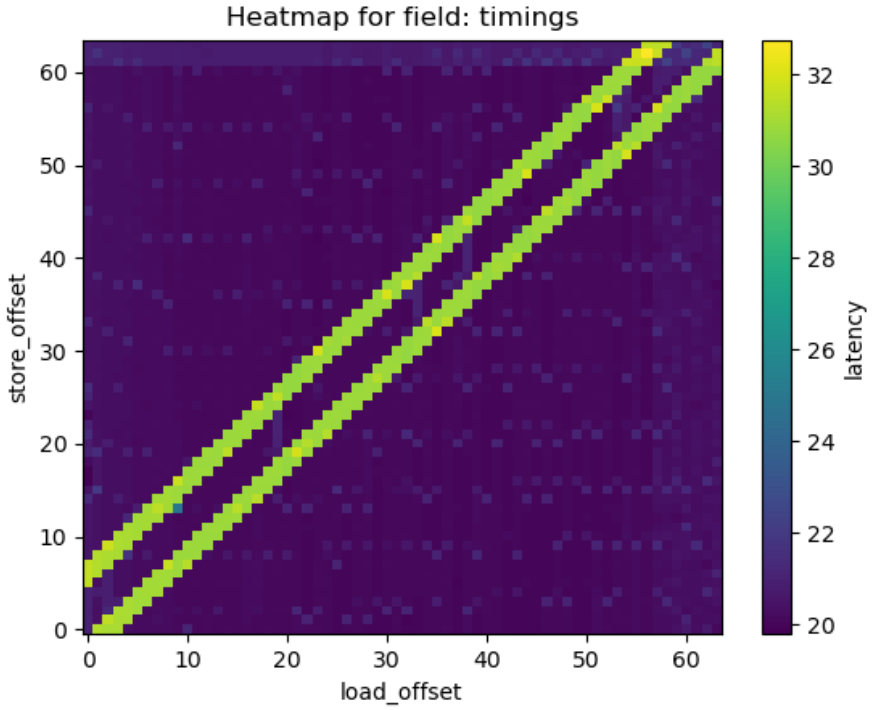

Figure 4.3: STL measurement results.

Mitigation To hide this feature TEA-checker inserts a `serialise` between every store and its paired load, forcing the store to drain to the cache first and flattening the heatmap.

Simultaneous Multi-Threading

Simultaneous Multi-Threading(§2.1.2) increases the severity of MDS type attacks since the shared LFB, store buffer, and load ports create a cross-thread observation channel. A CPU-bound busy loop is timed single-threaded, with two threads pinned to sibling logical cores, and with two threads on different physical cores, thus resource contention on sibling cores measurably increases the loop time, exposing SMT.

Mitigation To suppress this feature TEA-checker pins the sibling thread to a different physical core, removing the contention entirely.

ISA feature detection

ISA-level features, such as SIMD or Intel's SGX are detected through compile-time macros and `CPUID` queries rather than by probing with signal handlers or executing test instructions at runtime. This is due to the limitations of the proposed approach having to run code in a kernel environment, where there is no recovery measure for illegal instructions.

4.3.2. System interaction tests

Speculative memory access

As stated in Section 2.1.2 there are two main ways to create a transient window, the first is to cause a misprediction, and the second is to raise an exception due to a faulting instruction.
This test characterises the former by stretching the transient window by stalling its resolution on DRAM, thus generating a misprediction. The resulting window $W = n_{nops}$ is measured by stuffing the execution with filler instructions, until the transient result is not serialised to the microarchitecture anymore using a Flush+Reload (§2.2.1) probe.

Mitigation To hide this feature TEA-checker trains the predictor in a predictable way, thus removing the misprediction entirely and proving that the wrongly executed code path is the sole reason for the microarchitectural trace.

Out-of-order memory access

Inversely, this test uses the latter method to create a window. It uses an exception caused by missing load, thus confirming that a speculative load inside a mispredicted branch completes before the ROB flush and updates cache state.
Since, as stated before, exception handlers cannot be used by TEA-checker, the window stuffing and serialising load is put inside a mispredicted branch to void the code path to be architecturally executed. This does not generate the same misprediction transient window as the previous test since the branch uses a cached value.
Same as before, a binary search over the filler length identifies the largest window $W = t_{\text{nops}}$ that still permits a hit.

Mitigation To hide this feature TEA-checker inserts a `memory_barrier` inside the speculative path before the load, ensuring that the previous window closes before encoding the load probe into the microarchitecture.

Stale code execution

Self-modifying code creates an OoO window in which the processor transiently executes stale instruction bytes from the instruction or micro-op cache before the modification becomes globally visible. The first byte of `test_access` is overwritten with a return instruction, then a subsequent call to the modified function transiently executes the stale original bytes, which include `load(ptr)`, caching the probe address. A `flush_reload` measurement then detects whether the cache line was filled.

```
void test_access(unsigned long addr) { load(addr); }

void func(void) {
    // Build pointer chain to enlarge transient window
    // chain1 -> chain2 -> chain3 -> ret opcode
    build_chain(chain1, chain2, chain3, opcode);
    cache_line_flush(chain1, chain2, chain3);
    memory_barrier();

    // Overwrite test_access's first byte with ret
    *((volatile char *)test_access) = **chain1;

    // Call test_access -- stale micro-op cache may keep
       original bytes
    test_access((usize)ptr);

    // Check whether the stale load cached ptr
    result.hit = flush_reload(ptr);
}
```

Listing 4.9: Stale code execution test.

Mitigation To mitigate this feature TEA-checker inserts a `memory_barrier` after the code modification, forcing global visibility before any fetch can observe the stale bytes.

Branch target injection

The BTB makes it possible to predict the jump target of indirect branches, allowing to steer speculative execution to an attack-chosen path. The test trains the BTB on a target performing a load, and then, when performing another indirect call to a different target,

it causes a misprediction potentially resulting in performing the same load yet again.

Mitigation TEA-checker interleaves a random indirect call during training, polluting the BTB entry and preventing the poisoned target from persisting.

Load address prediction transient bypass

Differently from the feature test, to check the bounds bypass for the LAP the test traverses a linked list with a fixed stride, then issues a load with a deviating stride, making the LAP transiently accesses the wrong predicted address before the correct one resolves, thus loading memory into the cache hierarchy across the different address spaces

Mitigation TEA-checker replaces the fixed-stride traversal with random strides, preventing any predictable pattern from forming.

TLB stale-translation bypass

The TLB caches page table translations, but a stale entry can persist after its PTE is modified, allowing a load to transiently complete through the old translation before the page walker detects the change. The test clears the Present bit on a PTE without flushing the TLB, then loads from the affected address to transiently access data through the stale translation.

Mitigation TEA-checker flushes the TLB after the PTE modification, removing the stale translation before the load executes.

Return-stack-buffer underflow

The RSB predicts return addresses from the call stack, but a deep call chain exhausts its entries, causing a subsequent `ret` to mispredict and diverting the execution flow of the program to transiently load a cache line.

Mitigation TEA-checker fills the RSB with benign entries before every measured `ret`, preventing underflow.

```
no_inline void victim(void) { return; }
no_inline void load_ptr(void) {
    if (take_branch[idx]) load(cache_line);
}

no_inline void rsb_poison(int depth) {
    if (depth <= 0) return;
    rsb_poison(depth - 1);
}

void func(request_dependencies_t *args) {
    // ...
    for (idx = 0; idx < tries; idx++) {
        cache_line_flush(cache_line);
        for (int j = 0; j < 1000; j++) load_ptr();
        cache_line_flush(cache_line);
        cache_line_flush(&take_branch[idx]);
        serialise(); memory_barrier();

        rsb_poison(rsb_depth);
        // Under -DMITIGATE: rsb_stuff(READINGS)
        victim();  // ret mispredicts;
                   // fallback executes load_ptr
        // Measure cache_line access time
    }
}
```

Listing 4.10: RSB underflow interaction test.

Line-fill buffer leakage

The LFB holds outstanding cache-miss requests shared across SMT threads, allowing a sibling thread to observe stale data left by another thread. The test fills the LFB from one thread and probes it from the sibling to detect the leakage.

Mitigation TEA-checker inserts a `serialise` between parallel LFB fills, preventing stale data from persisting across the context boundary.

Store-to-load forwarding leak

The STL predictor forwards data from an older uncommitted store to a younger load to the same address, but a misprediction can forward stale data from the wrong address. The test issues a store to address A followed by a load to B, the result is that the predictor incorrectly forwards A's data, and the transiently received stale value is used for a secret-dependent cache access.

Mitigation TEA-checker inserts a `serialise` between the store and the dependent load, forcing the store to drain before the load issues.

This chapter established three core components for a structured microarchitectural evaluation of transient execution security. First, the modular, multi-runner test infrastructure that compiles a single measurement function against user space, kernel, and simulation targets without source modification. Second, the validation methodology that uses software mitigation as a negative control to establish causality between a measured timing differential and the microarchitectural feature under test. Third, the three-level scoring model that aggregates raw δ measurements into feature severities, attack-family scores, and privilege-level assessments.

5 | Experimental Results

This chapter evaluates the security framework presented in Chapter 4 on three COTS processors: an AMD Ryzen 9 7900X, an Intel Core Ultra 7, and an Intel Core i5-8250U. We start by checking whether the timing measurements are reproducible across repeated executions, then ask whether the framework can characterise CPUs from different vendors without per-platform calibration. With these foundations in place, we examine which speculative primitives are present on each platform, how far speculation reaches across privilege boundaries, and how the resulting scores compare against the manufacturers' own published vulnerability status. The Ryzen 9 7900X and Core Ultra 7 are recent processors from the two major x86-64 vendors, both incorporating the hardware-level mitigations introduced in response to the transient execution disclosures of Section 2.2.2. The Core i5-8250U predates those disclosures entirely, which lets us test whether the framework correctly distinguishes hardened from unhardened silicon and whether its measurements generalise across two independently designed microarchitectures rather than being specific to one vendor.

5.1. Experimental Setup

All measurements come from user space execution, with each test pinned to a fixed physical core for the duration of the run, following the isolation procedure described in Section 4.1.1. The system was otherwise idle during all campaigns to minimise background noise. The value reported for each test is the severity score S_f defined in Section 4.1.3, which for some tests is represented by a binary value. These report only whether a given microarchitectural structure is present or absent, with no intermediate score.

The physical evaluation covers x86-64 only. The framework's RISC-V runner, described in Section 4.1.1, was developed for a simulated environment to aid development but does not contribute data to this chapter.

Table 5.1: Attack families, required feature tests, corresponding system interaction tests, and score multipliers.

Attack family	Feature tests	Interaction test	Multipliers
Spectre v1 (BCB)	Cache (4.3.1), PHT (4.3.1)	Spec Window (4.3.2)	—
Spectre v2 (BTI)	Cache (4.3.1), BTB (4.3.1)	BTI Window (4.9)	—
Spectre-RSB	Cache (4.3.1), RSB (4.3.1), BTB (4.3.1)	RSB Window (4.9)	—
Spectre-STL	Cache (4.3.1), STL (4.7)	STL Window (4.10)	—
SLAP	Cache (4.3.1), LAP (4.3.1)	LAP Window (4.9)	—
Meltdown	Cache (4.3.1)	TLB Window (4.9)	—
L1TF	Cache (4.3.1), TLB (4.6), STL (4.7), BTB (4.3.1)	TLB Window (4.9)	×2 SGX
RIDL	Cache (4.3.1), LFB (4.6)	LFB Window (4.10)	×2 SMT
Fallout	Cache (4.3.1), STL (4.7), TLB (4.6)	STL Window (4.10)	—
ZombieLoad	Cache (4.3.1), LFB (4.6), SMT (4.8)	LFB Window (4.10)	×2 SGX
CacheOut	Cache (4.3.1), LFB (4.6), SGX (4.8)	LFB Window (4.10)	×2 SMT
LVI	Cache (4.3.1), STL (4.7), LFB (4.6)	STL Window (4.10)	—
SCSB	Cache (4.3.1)	Stale Code Window (4.3.2)	—

Table 5.1 maps each attack family to the feature tests and system interaction test it depends on, together with the score multipliers applied when relevant platform-level features are detected, as explained in Section 4.1.3. Table 5.2 summarises the three privilege-level variants under which every interaction test is executed.

Table 5.2: Security boundaries under which each system interaction test is executed.

Security boundary	Privilege level	Target location
`same`	Ring 3 (user)	Attacker's own address space
`user`	Ring 3 (user)	Protected user space addresses
`kernel`	Ring 0 (kernel)	Kernel memory via probe module

5.2. Experimental Results

5.2.1. Measurement Stability

Before interpreting any per-platform score, we first need to know whether the measurements themselves are reproducible. If predictor tests oscillate between zero and near-maximum readings across identical runs due to scheduler jitter, thermal state, or initial predictor state, then no single measurement can be treated as the processor's definitive state. To test this, we ran the full test suite nine times on the platform that showed the widest run-to-run variation during initial measurements, the AMD Ryzen 9 7900X. Table 5.3 reports the per-execution values for the three features that exhibited the largest spread. The BTB, the STL forwarder, and the RSB.

Table 5.3: AMD Ryzen 9 7900X per-execution variance.

Run	BTB	STL	RSB
1	0.0%	24.7%	3.4%
2	0.0%	31.3%	5.5%
3	33.6%	25.9%	98.2%
4	33.3%	99.8%	3.4%
5	0.0%	25.0%	6.7%
6	0.0%	25.3%	0.0%
7	33.0%	41.3%	7.9%
8	0.0%	25.5%	0.0%
9	0.0%	23.4%	98.1%
Average	11.1%	35.8%	24.8%

The RSB test alternates between near-zero readings and near-maximum readings 98.2% without any change to the underlying hardware. The microarchitecture's underflow fallback behaviour depends on the microarchitectural state resident at the moment of measurement. Some processors fall back to a BTB prediction on underflow, others stall,

making the measured latency depend on whether collateral predictor entries have been evicted. The BTB test follows a similar pattern 0.0% vs 33.0% because the predictor is indexed by partial address bits, making detection depend on how the test's code layout interacts with the hardware's set-associative mapping and with the OS memory layout. The STL test also produced a 99.8% reading in run 4, likely an outlier consistent with the elevated variance the AMD platform exhibits across all predictor tests. More instructive than the outlier is the spread of typical STL readings, which span 23.4% to 41.3% and confirm that no single run captures the platform's range of behaviour.

The wide run-to-run variation across all three tests confirms that no single execution can be treated as the processor's definitive state. The framework therefore reports both the central tendency and the dispersion as first-class outputs. The spread itself is equally informative, and a processor whose score oscillates between extremes presents a fundamentally different evaluation target than one that produces a stable value.

5.2.2. Architectural Feature Detection

A processor's vulnerability flags say which attacks the vendor claims to have mitigated, but they reveal nothing about which speculative primitives remain present on the silicon or how strongly they manifest. The fourteen feature tests described in Section 4.3.1 address this gap by measuring each primitive directly through timing differentials, without relying on any hardware documentation. Six of these tests (marked with † in the table) produce a binary outcome. They report whether a given microarchitectural structure is present or absent, with no intermediate severity. The remaining eight report a continuous severity score S_f reflecting how consistently the timing signal exceeds the detection threshold. Table 5.4 collects all fourteen results across the three platforms, each test run in both an unmitigated and a mitigated configuration.

Three questions were investigated: (i) can the framework detect speculative execution primitives on commercial processors, or do vendor mitigation flags conceal the hardware that remains present? (ii) can it differentiate between independently designed microarchitectures, or are the measured differences merely artefacts? (iii) can it detect primitives whose exploitability depends on subtle timing effects, which is the hardest case for a timing-based approach?

Table 5.4: Feature test severity scores across the three platforms.

	AMD Ryzen 9 7900X		Intel Core i5-8250U		Intel Core Ultra 7	
Feature tests	Normal	Mitigated	Normal	Mitigated	Normal	Mitigated
Cache	95.9%	0.1%	93.1%	1.9%	97.8%	7.4%
PHT	30.5%	1.9%	44.2%	12.2%	39.4%	0.6%
ROB†	100%	100%	100%	100%	100%	0%
LAP	0%	0%	0%	0%	0%	0%
SIMD†	100%	0%	100%	0%	100%	0%
Pipeline†	100%	0%	100%	0%	100%	0%
LFB	48.0%	16.0%	24.0%	16.0%	56.0%	16.0%
SMT†	100%	0%	0%	0%	100%	0%
TLB	16.3%	3.3%	73.5%	28.7%	59.9%	20.5%
SGX†	0%	0%	100%	0%	0%	0%
OoO†	100%	0%	100%	0%	100%	0%
BTB	11.1%	0%	21.2%	0%	12.5%	0%
STL	35.8%	0%	87.8%	0%	88.7%	0%
RSB	24.8%	0%	10.5%	0%	31.6%	24.8%
Feature Detection	54.5%		61.0%		63.3%	

(i) The aggregate Feature Detection scores confirm that all three platforms expose a broad and measurable speculative surface. The Core Ultra 7 scores highest despite being the most heavily mitigated system, which illustrates a central point of the methodology. The score measures the breadth of detectable features, and a richer feature set produces a higher aggregate regardless of whether individual primitives are mitigated.

(ii) The TLB and STL tests reveal a consistent vendor split, with both structures scoring substantially higher on Intel than on AMD across both Intel generations. Section 5.2.4 confirms that both tests collapse correctly under mitigation across all three platforms, so these differences reflect genuine design divergence rather than a measurement artefact. This is information that no vendor disclosure provides.

(iii) Branch predictor structures such as the BTB, PHT, and RSB fall into this category because the timing difference between a predictor hit and a predictor miss is smaller than the difference between a cache hit and a cache miss. All three tests produce moderate severity scores, demonstrating that the framework registers these primitives despite the inherently low observable differential. Doing so without hardware documentation and purely through user-space timing validates the architecture-agnostic approach directly.

Taken together, these measurements support two findings. The framework detects and differentiates speculative primitives across three commercially available processors from two vendors, using only user-space timing measurements and no privileged hardware information. At the same time, feature presence alone does not determine whether a primitive can be weaponised across privilege boundaries. The scores in Table 5.4 measure what is present on the silicon. Whether those same primitives can be made to leak data across

security boundaries is the subject of the following section.

5.2.3. Security Boundary Analysis

Raw feature presence does not imply exploitability of the system as a whole. Knowing that a processor implements an exploitable branch predictor or a reorder buffer is not the same as knowing whether an attacker can use those structures to leak data across privilege boundaries. To move from characterisation to assessment, we executed the nine system interaction tests described in Section 4.3.2 under both unmitigated and mitigated configurations, across each of the three privilege-level variants defined in Table 5.2. For each test, the Space column in Table 5.5 records the highest boundary crossed.
Two questions were investigated: (i) can the framework detect cross-boundary speculative leakage on these platforms, or does feature presence alone determine exploitability? (ii) can it identify which specific primitives are responsible for each boundary crossing and how far they reach, moving beyond a simple binary verdict? (iii) does the framework's unified score reveals structure that the disaggregated results alone cannot express, exposing information about mitigation effectiveness that the component scores do not show?

Table 5.5: System interaction test results and privilege-boundary crossings.

	AMD Ryzen 9 7900X			**Intel Core i5-8250U**			**Intel Core Ultra 7**		
Interaction tests	Normal	Mitigated	Space	Normal	Mitigated	Space	Normal	Mitigated	Space
Spec Window	0%	0%	safe	32.5%	0%	same	45.2%	0%	same
OoO Window	0%	0%	safe	32.5%	0%	safe	20.6%	0%	safe
BTI Window	0%	0%	safe	6.0%	0%	kernel	1.2%	2.1%	same
STL Window	0%	0%	safe	47.7%	0%	kernel	10.2%	0%	user
RSB Window	0%	0%	safe	100%	0%	kernel	28.4%	0%	safe
Stale Code Window	0%	0%	safe	32.5%	0%	kernel	45.2%	0%	safe
TLB Window	0%	0%	safe	4.1%	0%	safe	0.6%	0.6%	safe
LFB Window	0%	0%	safe	82.7%	0%	safe	7.1%	0%	safe
LAP Window	0%	0%	safe	0%	0%	safe	0%	0%	safe
Speculative Overview	0.0%		safe	37.6%		kernel	17.6%		user

(i) The Space column shows that AMD Ryzen 9 7900X crosses no boundary in any interaction test, while both Intel platforms leak across multiple privilege levels. This divergence is the central result. Feature presence alone does not determine exploitability, and the framework captures that distinction.
(ii) The interaction tests produce a range of signals across the two Intel platforms, with some primitives reaching kernel space on the older platform while the same primitives are contained to user or same space on the newer one. These results show that the

framework distinguishes not only whether leakage occurs but which mechanism drives it and how deeply it penetrates. The comparison also reveals a further capability: both Intel platforms exhibit similar feature breadth, yet the newer model reduces the reach of each primitive without removing the underlying mechanism. The framework therefore separates hardware capability from mitigation effectiveness, a distinction that vendor flags collapse into a single verdict.

A complete security assessment must account for both the features present on the silicon and their reachability across boundaries. Table 5.6 computes this combined view by averaging all twenty-three component scores, specifically the fourteen feature detection scores from Section 5.2.2 and the nine speculative overview scores from Table 5.5, into a per-platform aggregated vulnerability.

Table 5.6: Aggregated vulnerability scores per platform.

	AMD Ryzen 9 7900X	**Intel Core i5-8250U**	**Intel Core Ultra 7**
Feature Detection (14 tests)	54.5%	61.0%	63.3%
Speculative Overview (9 tests)	0.0%	37.6%	17.6%
Aggregated vulnerability (23 tests)	33.1%	51.6%	45.4%

(iii) The Feature Detection scores alone give each platform a similar rating, leaving the differences in boundary containment invisible. The aggregate exposes this hidden dimension. The two Intel platforms diverge based on how effectively their mitigations contain the same underlying speculative structures, a separation that neither the feature detection scores nor the interaction tests alone provide.

These scores measure the presence and strength of detectable microarchitectural features and boundary crossings, not real-world exploitability. A higher score indicates a richer attack surface, but turning that surface into a working exploit requires engineering effort the framework does not attempt to quantify. The scores are best read as a relative comparison across the three evaluated platforms rather than as an absolute measure of risk.

5.2.4. Cross-Platform Portability

An architecture-agnostic methodology must work across platforms without per-vendor calibration. Portability within a single ISA is a necessary condition before we can pursue cross-ISA portability. To test this, we executed the identical test suite on all three platforms, running each feature test both unmitigated and with its corresponding software mitigation active, applying the validation methodology described in Section 4.2 uniformly. Of the fourteen feature tests, thirteen collapsed correctly under mitigation on all three

platforms.

The ROB feature test is the exception, and it exposes a limitation in the portability of the detection algorithm itself. On AMD Ryzen 9 7900X and Intel Core i5-8250U the mitigated reading remains at 100%, identical to the unmitigated value, while the same test collapses correctly to 0% on Intel Core Ultra 7. Figures 5.1 and 5.2 show the underlying cause. Further analysis indicates that the fault lies in the detection function rather than in the mitigation itself. The function does not generalise correctly across different execution environments during mitigated runs, and the sudden spikes visible in the traces are attributable to operating system scheduling interference rather than to the ROB mechanism. The mitigation does suppress the timing jump that normally indicates ROB capacity, as Figure 5.2 shows, but the detection algorithm fails to register this suppression on two of the three platforms.

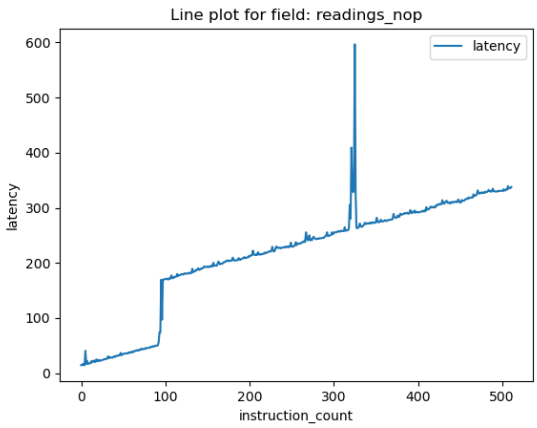

Figure 5.1: ROB readings under normal load.

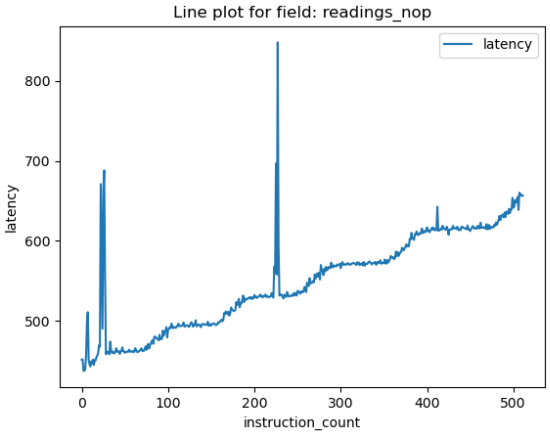

Figure 5.2: ROB readings with mitigations.

This exception highlights the portability limit. Detection thresholds calibrated on one platform cannot be blindly transferred to another. The remaining thirteen tests validate correctly across all three platforms, demonstrating that the framework's measurement principle is portable even if the signal-processing parameters require per-platform tuning.

5.3. Attack-Family Scoring and Vendor Comparison

A framework that disagrees with known ground truth has no practical value, but the ground truth available here warrants care. Vendor-reported vulnerability flags do not represent such truth in the usual sense as they report whether a hardware mitigation is active, not whether the underlying speculative mechanism is present on the silicon. The asymmetry is critical. When a vendor flags a platform as vulnerable for a given

attack family, the flag is trustworthy. The required microarchitectural mechanisms are present and the exploit has been demonstrated. When a vendor reports a platform as not vulnerable, the flag may mean the hardware mitigation is in place, but the historical record of transient execution research has repeatedly shown that a mitigated status can coexist with a working bypass variant [2]. The comparison in this section is therefore a one-sided validation. A false negative, where the framework misses something the vendor confirms, would be a genuine failure. A discrepancy, where the framework finds something the vendor reports as mitigated, is expected and informative.

For each attack family listed in Table 5.1, we computed the per-family score S_a per Equation 4.2 from the validated feature and interaction test results, then cross-referenced the outcome against each platform's vendor-reported vulnerability status. Comparing the framework's score against vendor-reported status yields one of four outcomes. True positive where both sides agree on vulnerability, true negative where both sides agree on no vulnerability, discrepancy where the framework finds features but the vendor reports mitigated, or false negative where the framework misses a vendor-confirmed vulnerability. Two questions were investigated: (i) does the framework ever contradict a vendor-confirmed vulnerability, which would be a direct falsification of its correctness? (ii) what do the seven discrepancies reveal about the evaluation gap between binary vendor flags and continuous hardware capability?

Table 5.7: S_a versus vendor-reported status for each attack family and platform.

	AMD Ryzen 9 7900X		**Intel Core i5-8250U**		**Intel Core Ultra 7**	
Family	S_a	Reported	S_a	Reported	S_a	Reported
RIDL	33%	no	67%	yes	67%	no
CacheOut	33%	n/a	67%	n/a	67%	n/a
Fallout	33%	no	100%	yes	67%	no
Foreshadow	25%	no	100%	yes	75%	no
ZombieLoad	0%	no	67%	yes	33%	no
Meltdown	0%	no	100%	yes	0%	no
LVI	50%	n/a	100%	n/a	50%	n/a
Spectre-RSB	50%	n/a	100%	n/a	100%	n/a
Spectre-STL	100%	n/a	100%	n/a	100%	n/a
Spectre v1	100%	yes	100%	yes	100%	yes
Spectre v2	50%	yes	100%	yes	100%	yes
SLAP	50%	n/a	50%	n/a	50%	n/a
SCSB	100%	yes	100%	yes	100%	yes

(i) The outcome is unambiguous. Zero false negatives occur on any platform. Every attack family that a vendor flags as vulnerable receives a nonzero score from the framework,

indicating that the framework correctly models the mechanisms these families target.

Table 5.8: Outcome classification across the twenty-four comparable data points.

Outcome	Count
True positive	14
True negative	3
Discrepancy	7
False negative	**0**

(ii) In each discrepancy, the vendor reports a platform as not vulnerable while the framework assigns a nonzero score. These are not false positives. The discrepancies are the evaluation gap that Section 2.2.2 identified. The framework's continuous scores expose residual attack surface that binary vendor flags cannot represent. That the discrepancies cluster on AMD and Intel Core Ultra 7, the two platforms with the most extensive mitigation deployment, is consistent with this interpretation. These are precisely the platforms where a binary mitigated flag is most likely to conceal remaining hardware capability.

6 | Conclusions

This thesis has shown that architecture-agnostic timing measurement is able to characterise transient execution behaviour on Commercial of the Shelf processors without access to their internal design. TEA-checker was developed to test this premise directly, and the results obtained across three physical platforms in Chapter 5 support it, in fact, the framework reliably identifies which speculative execution primitives a processor implements, measures how exploitable each one is in practice, and characterises the severity of the resulting isolation breach independently of whether that breach corresponds to a previously catalogued attack. The analysis of the data collected in this work points to three findings in particular.

- **Accurate detection of speculative behaviour.** Across every case where an independent ground truth was available, TEA-checker's feature and interaction tests detected the corresponding primitives. This is not a claim that the framework is complete, but it is direct evidence that the underlying measurement principle, a structurally paired timing differential validated against its own software mitigation, correctly isolates the microarchitectural mechanisms it targets rather than producing scores that happen to agree with known answers by coincidence.

- **Exploitability quantified through window measurement.** Where prior post-silicon tools report only whether a known exploit succeeds, the system interaction tests developed in this thesis measure the actual size of the transient execution window in instruction count, derived from the timing of the underlying feature test scaled through the platform's measured execution rate. This provides a more fine grained security evaluation then simple binary classification through vendor provided flags

- **Severity characterised independently of named attack families.** The generic boundary-breach test introduced in Section 4.1.3 probes for any speculative access that crosses an isolation boundary, regardless of the specific triggering mechanism. Because it does not assume a documented exploitation primitive, it gives the framework a way to assign a severity even to undiscovered attack vectors

Taken together, these three results support the central premise of the thesis. A processor's actual speculative execution behaviour can be characterised from outside, using nothing but timing, without trusting what the processor's vendor claims about it and without waiting for a named exploit to exist for every mechanism worth measuring.
This thesis makes two contributions to that end. The first is the architecture-agnostic testing framework itself, constituted by the orchestrator, the privilege-aware execution runners, the dependency-resolved module system, and the validation methodology that together make it possible to extract this information reliably and reproducibly from unmodified commercial silicon. The second is the continuous security model built on top of it. The scoring scheme that moves from a single timing differential, through a per-feature severity score, to an exploitability measure and a boundary-crossing assessment, without collapsing any of that information into a binary verdict along the way.

6.1. Future Work

While this work has shown the effectiveness of TEA-checker at characterising the speculative properties of the CPUs used for the study, it is also true that a major limiting factor for the feature selection and interaction taxonomy was the availability of vulnerabilities on the CPU used for the framework development. As such, this thesis does not encompass the full landscape of TEA vulnerabilities.

The first step for future research is to broaden the scope of the microarchitectural tests to include more transient execution behaviors. Thanks to the modular nature of the proposed verification framework, this task is reduced to defining a new module that fits the required structure, without requiring any modification to the orchestrator, the dependency resolution logic, or any existing test. Newly disclosed primitives such as GhostRace, which targets race conditions between concurrently executing speculative paths, or Native Branch History Injection, which extends Spectre v2 across privilege boundaries previously assumed protected by eIBRS, are natural candidates for this extension, since both fit within the existing feature and interaction taxonomy and would require no changes to the underlying measurement infrastructure described in Chapter 4.

A second, closely related direction, would be the broadening of the set of supported ISAs. The portability claims of this platform rest on the set of architecture-agnostic primitives described provided by the framework itself. Currently, x86-64 is the architecture that is the most battle tested and with the most feature support, and RISC-V was used solely in a simulated environment. So, future research is suggested to:

- Perform the analysis on physical RISC-V cores
- Expand the provided ISA collection offered by providing a target for the ARM architecture

Lastly, experiments showed a flaw in the data processing step of the evaluation, indicating that statistical detection is not yet platform-agnostic. This is further shwon in the managers where the thresholds that define the boundaries of when a feature is detected or discarded, are calibrated against the noise characteristics of the processor used when developing the framework. A clear example of this is the ROB. Future work should replace these thresholds with more adaptive data analysis pipelines that can effectively cancel the target platform noise and adjusts detection sensitivity accordingly rather than relying on values tuned in advance on different hardware. The run-to-run variance observed on the AMD platform for the BTB, STL, and RSB tests reinforces this point: even within the same x86-64 ISA, two vendors with different microarchitectural implementations, and sometimes two generations from the same vendor, can produce measurement characteristics that a single calibration cannot capture. Improving portability within x86-64 is therefore a prerequisite for the cross-ISA portability the framework claims.

Each of these directions extends the same two contributions described above rather than replacing them: better statistics make the architecture-agnostic framework more reliable, broader attack coverage and ISA support make the continuous security model more complete. Neither requires rethinking the measurement principle this thesis is built on, they only require applying it more carefully, and more widely, than was possible within the scope of this work.

Bibliography

[1] L. Bancale. TEA-checker. `https://github.com/sbancuz/TEA-checker`, 2026.

[2] C. Canella, J. V. 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, 2019. URL `https://arxiv.org/abs/1811.05441`.

[3] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, 2019. doi: 10.1145/3319535.3363219.

[4] H.-V. Dang and A.-Q. Nguyen. Unicorn: Next generation CPU emulator framework. 01 2015.

[5] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi: 10.1109/JSSC.1974.1050511.

[6] B. Fu, L. Tenenbaum, D. Adler, A. Klein, A. Gogia, A. R. Alameldeen, M. Guarnieri, M. Silberstein, O. Oleksenko, and G. Saileshwar. AMuLeT: Automated design-time testing of secure speculation countermeasures, 2025. URL `https://arxiv.org/abs/2503.00145`.

[7] 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 2016)*, volume 9721 of *LNCS*, pages 279–299. Springer, 2016.

[8] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2020. doi: 10.1109/SP.2020.00005.

[9] Z. He, G. Hu, and R. B. Lee. CloudShield: Real-time anomaly detection in the cloud. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security*

and Privacy, CODASPY '23, pages 91–102, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700675. doi: 10.1145/3577923.3583639. URL https://doi.org/10.1145/3577923.3583639.

[10] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[11] B. Herzog, H. P. Reiser, and R. Kapitza. The price of meltdown and spectre: Energy overhead of mitigations at operating system level. In *Proceedings of the 14th European Workshop on Systems Security (EuroSec '21)*, pages 8–14, 2021. doi: 10.1145/3447852.3458706.

[12] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.

[13] J. Kim, D. Genkin, and Y. Yarom. SLAP: Data speculation attacks via load address prediction on Apple Silicon. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3549–3566. IEEE, 2025.

[14] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[15] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[16] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: exploiting speculative execution. *Commun. ACM*, 63(7):93–101, June 2020. ISSN 0001-0782. doi: 10.1145/3399742. URL https://doi.org/10.1145/3399742.

[17] P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[18] 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)*. USENIX Association, 2018.

[19] E. Lai, W. Xiong, E. Suh, M. Tiwari, and M. Luo. Towards reinforcement learning for exploration of speculative execution vulnerabilities, 2025. URL `https://arxiv.org/abs/2502.16756`.

[20] S. Lesimple. Reptar, downfall, zenbleed, zombieload, RIDL, fallout, foreshadow, spectre, meltdown vulnerability/mitigation checker for linux and BSD, 2026. URL `https://github.com/speed47/spectre-meltdown-checker`. GitHub Repository.

[21] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[22] 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.

[23] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. V. wagons, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. The gem5 simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020. URL `https://arxiv.org/abs/2007.03152`.

[24] A. Mambretti, P. Convertini, A. Sorniotti, A. Sandulescu, E. Kirda, and A. Kurmus. GhostBuster: understanding and overcoming the pitfalls of transient execution vulnerability checkers. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 307–317, 2021. doi: 10.1109/SANER50967.2021.00036.

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

[26] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® software guard extensions (Intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347693. doi: 10.1145/2948618.2954331. URL `https://doi.org/10.1145/2948618.2954331`.

[27] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[28] E. S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954. doi: 10.2307/2333009.

[29] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Four generations of spec cpu benchmarks: what has changed and what has not. Technical report, Technical Report TR-041026-01-1, University of Texas Austin, 2004.

[30] *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*. RISC-V International. Document version 20191213.

[31] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, 2019. doi: 10.1145/3319535.3354252.

[32] M. Song, T. Suh, and G. Koo. Vizard: Passing over profiling-based detection by manipulating performance counters. *IEEE Access*, 11:48099–48112, 2023. doi: 10.1109/ACCESS.2023.3260179.

[33] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan. RTL verification for secure speculation using contract shadow logic, 2024. URL `https://arxiv.org/abs/2407.12232`.

[34] C. Trippel, D. Lustig, and M. Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960. IEEE, 2018. doi: 10.1109/MICRO.2018.00081.

[35] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys

to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008. USENIX Association, 2018.

[36] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019. doi: 10.1109/SP.2019.00087.

[37] B. L. Welch. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947. doi: 10.2307/2332510.

[38] J. Wikner and K. Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2787–2804. USENIX Association, 2022.

[39] H. Wong. Measuring reorder buffer capacity, May 2013. URL `https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/`.

[40] H. Wong. Store-to-load forwarding and memory disambiguation in x86 processors, Jan. 2014. URL `https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/`.

[41] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[42] J. Xu, Y. Zhou, X. Zhang, Y. Li, Q. Tan, Y. Zhang, Y. Zhou, R. Chang, and W. Shen. DejaVuzz: Disclosing transient execution bugs with dynamic swappable memory and differential information flow tracking assisted processor fuzzing, 2025. URL `https://arxiv.org/abs/2504.20934`.

[43] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732. USENIX Association, 2014.

[44] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, May 2020.

List of Acronyms

BTB Branch Target Buffer. 18, 19

CISC Complex Instruction Set Computer. 10

COTS Commercial of the Shelf. 4, 35

GHR Global History Register. 18

HPC Hardware Performance Counters. 4, 34, 36, 37

IBPB Indirect Branch Predictor Barrier. 42

ILP Instruction Level Parallelism. 1, 2, 11

IPC Instructions Per Cycle. 11, 12

ISA Instruction Set Architecture. 2, 6, 9–12, 15, 17, 20, 21, 23, 39

L1TF Level 1 Terminal Fault. 27

LRU Least Recently Used. 14

MMU Memory Management Unit. 26

OoO Out-of-Order. 15, 20, 26, 33, 34

OS Operating System. 4, 15

PC Program Counter. 18

PHT Pattern History Table. 17, 18, 28

PoC Proof of Concept. 4, 5, 35–37, 40

RISC Reduced Instruction Set Computer. 10

ROB Reorder Buffer. 15–17, 28

RSB Return Stack Buffer. 18, 29

RTL Register-Transfer Level. 4, 7, 31, 32, 37

SGX Software Guard Extension. 3

SMT Simultaneous Multi-Threading. 19, 20, 27

STL Store to Load. 16

TEA Transient Execution Attacks. 2–6, 21, 22, 25, 31, 37–40, 42

TLB Translation Lookaside Buffer. 15

List of Figures

List of Tables