POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Automated Assembly-Level Mitigation of Transient Execution Attacks using Diversified Code Variants and Genetic Search

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Gabriele Giordanelli

Advisor: Prof. Luca Maria Cassano

Co-advisor: Elia Lazzeri

Academic year: 2025-2026

1. Introduction

Transient Execution Attacks (TEAs) exploit speculative and out-of-order execution to transiently process data that is architecturally inaccessible and to encode it into microarchitectural state [5][4]. Although architectural state is rolled back on misprediction or faults, the microarchitectural footprint can persist long enough to be recovered through timing side channels [4]. As a consequence, TEAs break the assumption that architectural isolation alone (process/container/microVM boundaries) implies confidentiality on shared hardware [2]. From a deployment standpoint, practical defenses must satisfy three constraints. First, they must be software-deployable on already-deployed processors, complementing hardware mitigations if present. Second, they must be selective: global strategies such as "fence everywhere" can introduce large overheads, making them unsuitable for performance-sensitive contexts [3]. Third, they should support a tunable security–performance trade-off [1], because the effective mitigation mix depends on workload characteristics and on which TEA variant dominates the threat model.

This thesis proposes an end-to-end compile-time mitigation framework that performs assembly-to-assembly rewriting of compiler-emitted x86-64 code. The framework detects candidate transient windows for multiple TEA classes, rewrites only those regions into diversified hardened variants, and optionally uses genetic search to select a single global configuration meeting an overhead budget (or, conversely, minimizing overhead under a protection target). The focus is on providing a deployable mitigation layer compatible with standard toolchains, without requiring source-code annotations or manual assembly patching.

1.1. Scope and contributions

The framework targets Spectre V1 (bounds-check bypass) [4], Spectre V4 (speculative store bypass) [4], and Meltdown-like faulting-load patterns [5], focusing on gadget hardening (disrupting secret-dependent encoding) instead of eliminating side channels in general.

Main contributions:

- An assembly rewriting pipeline that detects transient windows and rewrites only affected regions while preserving correctness and toolchain compatibility.
- Variant-based mitigation: each window is

replaced by a lightweight selector and N semantically equivalent variants, enabling diversification and transformation composition.
- A composable transformation catalog (fences/ordering barriers, masking/sandboxing, dependency injection, and lightweight structural perturbations) applied conservatively with applicability checks.
- A dataset-driven genetic optimizer that searches the configuration space and outputs a single global configuration balancing protection across the three attacks under chosen constraints.
- An experimental campaign on three TEA PoCs quantifying leakage reduction vs. cycle overhead and clarifying which mitigation families dominate trade-offs for each TEA variant.

Overall, the thesis aims to bridge a practical gap between manual patching (hard to scale and error-prone) and monolithic defenses (broadly applicable but expensive in terms of performance), by offering a selective, configurable, and optimizable rewriting stage.

2. Background and positioning

2.1. Transient execution in a nutshell

A TEA can be summarized as: *trigger* → *transient window* → *secret-dependent access* → *microarchitectural encoding* → *recovery* [2][3]. The unit of mitigation in this thesis is the *window*: the framework detects and rewrites code regions likely to host secret-dependent encoding, with the goal of reducing the reliability of leakage.
The targeted variants differ mainly by how the transient window is created: Spectre V1 uses control-flow speculation past a conditional branch (e.g. bounds checks)[4], Spectre V4 exploits memory disambiguation (speculative store bypass), creating a store–load window [4], and Meltdown-like patterns use faulting loads that transiently forward data to dependent operations before the fault is architecturally handled [5]. These differences matter for mitigation because V4 often yields compact windows where many rewrites are unsafe, while V1 commonly offers more room for data sanitization and structural perturbations. On the contrary, Meltdown-like patterns often require stopping transient propagation (cutting the dependent chain) or constraining the dereference target.

2.2. Related work: where this thesis fits

Software defenses typically fall into: barrier-based, data hardening, or selective rewriting guided by analysis/detection [2][3]. Barrier-based defenses are broadly applicable but can be expensive, data hardening can be cheaper but pattern-specific, selective rewriting reduces overhead but depends on detection precision and conservative safety rules [2].
This thesis combines these directions with two additional ingredients: diversified variants per window and search-based configuration. Instead of committing to a single policy, variants implement different mitigation mixes, and an optimizer can select one global configuration that performs well across multiple TEA classes. In practice, this makes the framework usable when the attacker model is uncertain or when a deployment wants one mitigation profile that remains robust across multiple TEA triggers.

3. Threat model and security objectives

3.1. Deployment scenario

The target deployment is a multi-tenant platform with strong architectural isolation (process/container/microVM) but shared microarchitectural resources (cache hierarchy, predictors, memory subsystem). The framework is deployed as a compile-time hardening stage: it outputs rewritten assembly that is assembled and linked into the shipped binary, so it integrates into existing build pipelines. Target workloads handle secrets (e.g., keys, tokens) and are exposed to attacker-controlled inputs.

3.2. Attacker capabilities

The attacker is an unprivileged tenant who can repeatedly trigger victim code (e.g., via inputs/APIs) and perform cache-based timing measurements under occasional co-residency. Shared pages are not assumed; cross-domain attacks such as Prime+Probe are in scope. For Spectre variants, the attacker can influence branch history or store/load timing so the gad-

get executes repeatedly under favorable microarchitectural conditions. The attacker cannot control compilation, disable platform mitigations, or read victim memory architecturally; the advantage comes from repetition and microarchitectural observation.

3.3. Security goal and non-goals

The goal is not to eliminate speculation globally, but to prevent or substantially degrade reliable secret-dependent encoding by rewriting vulnerable patterns (e.g., sanitizing indices/pointers, enforcing ordering, and reducing repeatability via diversification). Non-goals include complete TEA coverage (e.g., BTI/V2-heavy attacks), removal of side channels in general, and constant-time guarantees for arbitrary programs.

4. Method overview

The pipeline transforms compiler-emitted assembly into a hardened file while preserving toolchain compatibility:

1. **Parse**: build a representation of functions/instructions and preserve labels/ABI.
2. **Detect**: find candidate transient windows for V1, V4, and Meltdown-like behavior.
3. **Rewrite**: replace each window with a selector and N variants (Variant 0 unchanged; others transformed).
4. **Emit**: generate standard x86-64 assembly and output a structurally equivalent hardened file.

The design is selective: only regions matching detector patterns are rewritten, limiting overhead and perturbations to unrelated hot paths. Detectors return window boundaries so the rewrite affects a minimal region and preserves surrounding labels and function structure.

4.1. Runtime selector mechanism

Each rewritten window is guarded by a compact selector that saves clobbered registers, derives a pseudo-random index from `rdtsc`, maps it to $[0, N-1]$, and dispatches to the chosen variant. Correctness measures include avoiding red-zone clobbering in leaf functions (stack padding when needed) and ensuring variants restore the expected architectural state and rejoin a common continuation label. The "duplication-only" baseline isolates selector overhead from transformation overhead.

4.2. Why diversified variants

Variants enable *mechanism composition* (different mitigations per clone) and *diversification* (clones at different addresses reduce repeatability and can degrade cache-based recovery). In the framework, diversification is controlled by N, while k controls how many transformations are applied per clone. Empirically, large N with small k often yields better budgeted trade-offs than aggressive per-variant rewriting, since some transformations quickly dominate cycle cost.

5. Transformation space

Transformations are applied at window scope only when safety checks pass (e.g., temporaries available, liveness and ABI preserved). If unsafe, the generator falls back to leaving the variant unchanged, prioritizing semantic preservation and fully automated, toolchain-compatible output.

5.1. Speculation-stopping and ordering primitives

Fences and ordering barriers are direct but often costly. The framework supports fences after conditional branches, store–load ordering fences, and control-dependent "cut" fences to stop transient execution past sensitive points. They serve as robust fallbacks when cheaper rewrites are inapplicable, especially for compact windows where correctness constraints limit higher-level transformations.

5.2. Data sanitization and masking

Data hardening ensures transient execution cannot use secrets as meaningful addresses/indices. The framework includes index masking (map indices to a safe range) and pointer sandboxing (redirect sensitive dereferences to a safe region). These rewrites are applied conservatively due to register pressure, but when applicable they often achieve strong leakage reductions at moderate overhead compared to fence-only strategies.

5.3. Dependency injection and structural perturbations

Lightweight mechanisms introduce dependencies (delay sensitive operations without full fences) and perturb structure (reordering where safe, small padding, address-arithmetic vari-

ants). Alone they rarely eliminate leakage, but they can improve trade-offs when combined with at least one direct mechanism and increase diversity in microarchitectural footprints across variants.

5.4. Configuration surface

Key parameters include N, k, enabled transformation families, and sampling weights. The resulting space is large, making manual tuning impractical when protecting multiple TEA variants under explicit overhead constraints. Additional switches separate selector overhead from transformation overhead (e.g., forcing identical variants), supporting fair baselines and surrogate learning.

6. Genetic optimization of configurations

6.1. Problem formulation

Configuration selection is a constrained trade-off problem: each configuration (e.g., N, k, and transformation-weight profile) induces both cycle overhead and leakage suppression, and must perform across multiple TEA variants. The optimizer therefore searches the configuration space and outputs a *single global configuration* applied consistently to Spectre V1, Spectre V4, and Meltdown-like windows.
Two equivalent deployment modes are supported:

- **Budget-driven**: maximize protection under an overhead budget (typically enforcing worst-case across attacks).
- **Security-driven**: minimize overhead under a protection target (typically enforced on the worst-case attack).

The user specifies either an overhead ceiling or a protection floor, and the optimizer returns the best configuration under that policy.

6.2. Surrogate-based fitness evaluation

To avoid expensive loops for every candidate, the framework trains lightweight regressors on the experimental dataset to predict leakage and overhead from configuration parameters. Separate per-attack predictors are learned, and fitness aggregates the predicted metrics across attacks. Penalties discourage budget violations and unreliable extrapolation (configurations far from measured regions), enabling multiple optimization profiles without re-running the full campaign.

6.3. Genetic algorithm design

A GA searches mixed discrete/continuous chromosomes (e.g., N, k, a "same variants" flag, and weight vectors) using standard operators (selection, crossover, mutation). Selected solutions often exhibit large N with small k and multi-mechanism mixes. Since V4 is typically the limiting case, solutions allocate probability mass to ordering primitives while preserving lighter components that remain effective for V1 and Meltdown-like patterns.

7. Experimental evaluation

7.1. Methodology and metrics

We evaluate three TEA Proof of Concepts (PoCs) microbenchmarks (Spectre V1, Spectre V4, Meltdown-like). Results are reported relative to the unprotected baseline of each PoC using: **Bytes stolen (%)** as an end-to-end leakage metric (lower is better), and **Cycle overhead (%)** (lower is better). When needed, **Protection (%)** is defined as 100 − Bytes stolen (%).

7.2. Per-attack results

Tables 1–3 summarize representative configurations per attack. Rows are selected policies/configurations, and columns report N (variants), leakage, and overhead.

Spectre V1. Table 1 shows that diversification alone is insufficient at moderate N (e.g., duplication-only at $N = 75$ still leaks 86.80% with 32.69% overhead). Adding a direct mechanism achieves much stronger suppression at small N: at $N = 5$, index masking reduces leakage to 10.80% (23.55% overhead), while fence-only reaches 0% (27.70% overhead). A hand-picked mix achieves low residual leakage (2.32%) within a similar overhead range (≈22–23%).

Spectre V4. Table 2 highlights V4 as the budget-limiting case: duplication-only remains high even at $N = 1000$ (55.63% leakage, 16.02% overhead) and reaches 0% only at extreme N (e.g., $N = 5000$, 48.34% overhead). In con-

Table 1: Spectre V1: representative configurations (leakage vs overhead).

N	Policy	Bytes stolen (%)	Ovhd (%)
1	Unprotected	97.80	0.00
75	Duplication-only	86.80	32.69
5	Idx-only (k=1)	10.80	23.55
5	Fence-only (k=1)	0.00	27.70
10	Hand-picked mix	2.32	22.63

Table 2: Spectre V4: representative configurations (leakage vs overhead).

N	Policy	Bytes stolen (%)	Ovhd (%)
1	Unprotected	100.00	0.00
1000	Duplication-only	55.63	16.02
500	Fence-only (k=1)	7.82	17.65
1000	Hand-picked mix	25.25	19.03
5000	Duplication-only	0.00	48.34

trast, ordering primitives dominate: fence-only already reduces leakage to 7.82% at $N = 500$ with 17.65% overhead, and mixes are effective and have similar overhead range mainly because they include ordering/SSB-related primitives.

Meltdown-like. Table 3 shows that duplication-only barely helps (83.21% leakage even at $N = 5000$). Strong reduction requires a path-breaking mechanism: pointer sandboxing reaches 10.89% leakage at $N = 100$ with 14.37% overhead while fence-only achieves 1.67% leakage at $N = 100$ with 25.49% overhead. A hand-picked mix instead obtains 1.34% leakage at 24.57% overhead.

7.3. Global optimization results and comparison

Hand-picked and fence-oriented baselines implicitly assume *attack-specific tuning* (different choices for V1, V4, or Meltdown-like). In contrast, the genetic optimizer outputs a *single global configuration* that targets all attacks simultaneously without requiring the user to know in advance which TEA variant is relevant.
Table 4 compares, for each attack (row block), the GA global configuration against the best hand-picked per-attack configuration and the strongest fence baseline, reporting N, k, protection, and overhead. Under an average overhead

Table 3: Meltdown-like: representative configurations (leakage vs overhead).

N	Policy	Bytes stolen (%)	Ovhd (%)
1	Unprotected	100.00	0.00
5000	Duplication-only	83.21	11.28
100	Sand-only (k=1)	10.89	14.37
100	Fence-only (k=1)	1.67	25.49
1000	Hand-picked mix	1.34	24.57

budget of about 24%, the GA remains within budget and achieves high protection on all three PoCs, it is competitive on V1 and Meltdown-like, while improving substantially over manual choices on V4. Fence-only can reach maximal protection but is often less budget-friendly, supporting mixed global configurations as a more deployable default, especially if additional TEA variants are considered.

7.4. Interpretation

- Diversification helps, but robust suppression generally needs at least one direct mechanism (ordering/barriers or sanitization/sandboxing), purely structural changes tend to saturate early.
- Applicability constraints, especially for V4, shape the optimal mix and often drive global configurations, explaining why the optimizer assigns weight to ordering primitives under budgets.
- Under budgets, mixed policies allocate overhead more efficiently than fence-only strategies, achieving high protection on multiple attacks without committing to maximal barriers everywhere.

7.5. Security analysis and limitations

The proposed defense is selective and software-only, so its guarantees are bounded by detection coverage and by the attack model. First, pattern-based detectors can miss gadgets (false negatives) or conservatively skip transformations due to applicability constraints, in particular, compact V4 windows often restrict which rewrites are safe. Second, diversification reduces repeatability but does not remove side channels: a sufficiently persistent attacker may increase samples, attempt to learn the selector behavior, or exploit alternative microarchitectural channels not targeted by the PoCs. Third,

Table 4: Representative comparison (24% overhead budget): GA global configuration vs best per-attack baselines.

Attack	Strategy	N	k	Prot. (%)	Ovhd. (%)
Spectre V1	GA (global)	1027	1	95.74	24.02
	Best hand-picked	10	6	97.68	22.63
	Best fence baseline	5	1	100.00	27.70
Spectre V4	GA (global)	1027	1	95.93	24.02
	Best hand-picked	2500	6	87.57	29.21
	Best fence baseline	1000	1	100.00	24.53
Meltdown	GA (global)	1027	1	96.86	24.02
	Best hand-picked	1000	6	98.66	24.57
	Best fence baseline	100	1	98.33	25.49

large N increases code size and can also introduce instruction-cache pressure. Finally, the optimizer's output is guided by a surrogate model trained on measured configurations, this accelerates search but predicted trade-offs should be validated when transferring to different CPUs or runtime conditions.

8. Conclusions and future developments

This thesis shows that selective assembly-level rewriting can provide a practical, deployable mitigation layer for TEA gadgets on x86-64 systems. The framework detects candidate windows, rewrites them into diversified variants guarded by a selector, and can automatically tune configurations via genetic optimization driven by empirical data. Results indicate that diversification reduces leakage but is rarely sufficient alone, effective points typically combine a direct mechanism with cheaper perturbations. Under strict overhead budgets, GA-selected global configurations achieve high protection simultaneously for Spectre V1, Spectre V4, and Meltdown-like patterns, improving over manual tuning on the limiting case (V4) while staying within budget and achieving a lower overhead than the best fence baselines only at the cost of a few percentage points. Future work includes LLVM integration, broader TEA coverage, portability to other ISAs/dialects, and additional validation on realistic workloads and microarchitectures.

References

[1] Intel Analysis of Speculative Execution Side Channels. Technical report, 2018.

[2] Claudio Canella, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Jo Van Bulck, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. *A Systematic Evaluation of Transient Execution Attacks and Defenses.*

[3] Luís Fiolhais and Leonel Sousa. Transient-Execution Attacks: A Computer Architect Perspective. *ACM Computing Surveys*, 56(3), 3 2024.

[4] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. 1 2018.

[5] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown: Reading Kernel Memory from User Space.*