Exploiting Hidden Resource Contention in Selective Speculation Defenses

Xiaoyu Cheng[1], Fei Tong[1,2,3,*], Zhenyu Lei[1], Fang Jiang[1], Zhe Zhou[1], Guang Cheng[1], Trevor E. Carlson[4]

[1]*School of Cyber Science and Engineering, Southeast University, China*
[2]*Purple Mountain Laboratories, China*
[3]*Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China*
[4]*National University of Singapore*

Abstract

Transient execution attacks continue to evolve beyond cache-centric channels, motivating selective speculation defenses that aim to provide comprehensive protection with low overhead by delaying only transmit instructions. In this work, we show that several state-of-the-art selective-speculation defenses rest on shared assumptions that overlook important microarchitectural behaviors, leaving systematic blind spots that admit secret-dependent reservation-station (RS) contention.

Our analysis identifies three limitations in optimized selective-speculation designs. First, existing transmit taxonomies emphasize post-issue execution effects and miss dispatch-phase channels, such as operand-dependent μop expansion in instructions (e.g., `REP`-prefixed string operations) that create operand-dependent RS occupancy before execution. Second, the delay-until-resolution strategy focuses on redirect-based control leakage, but predicated instructions (e.g., x86 `CMOV` and RISC-V Zicond) enable secret-dependent selection without branch resolution, allowing secrets to steer operand-dependent μop expansion (e.g., `REP` iteration counts). Finally, the older-μop-first allocation strategy assumes unsafe contention is prevented by prioritizing non-transient μops, yet undelayed arithmetic and cache-hit memory μops can still lead to secret-dependent RS pressure through latency-amplifying dependency chains.

Guided by these findings, we construct Spectre-v1-style gadgets that bypass STT/DOLMA on x86 and RISC-V gem5 models. We further validate RS-contention effects on real CPUs, including a `REP MOVSB`- and `REP STOSB`-based proof-of-concept and a real-world RS-contention pattern identified by our LLVM pass, demonstrating realistic gadget structure and measurable signal. Finally, we propose strengthened STT mechanisms that close both existing and newly exposed gaps at moderate performance overhead.

*Corresponding author: ftong@seu.edu.cn

1 Introduction

Transient execution attacks, epitomized by Spectre [25] and Meltdown [29], continue to pose a critical threat to modern computing systems, which can leak sensitive data by exploiting microarchitectural side channels exposed during transient execution. While significant effort has been invested in developing cache-centric defenses [3, 10, 28, 47] to mitigate these vulnerabilities, attackers continue to discover new exploitation vectors beyond traditional cache-based side channels [4, 7, 8, 13, 18, 35, 37–39, 51]. To counteract this evolving landscape, selective speculation schemes [6, 11, 23, 33, 46, 50] aim to provide comprehensive protection by preventing the propagation of sensitive data through potential side channels. A number of these selective schemes, such as STT [50], SPT [11], DOLMA [33], and SpecTerminator [23], are designed to minimize performance overhead by delaying only those instructions that can transmit secret data (known as transmit instructions), rather than suppressing all speculation. These schemes rely on a shared set of "safe" assumptions about what constitutes transmission and where secret-dependent effects can surface, and many of their optimizations have gradually become default design choices adopted across defense solutions. This raises a fundamental question:

Do these assumptions capture all of the microarchitectural behaviors that can leak information?

In this paper, we show that these assumptions are incomplete. While these assumptions provide a useful starting point, they fail to model several microarchitectural effects that can induce a reservation-station (RS) contention side channel. Prior work [7] noted that speculative load misses can create RS contention and thereby delay frontend fetch. We revisit and generalize this phenomenon, showing that RS pressure can arise from multiple sources beyond load μops, and that the resulting fetch throttling can remain exploitable even when modern selective-speculation defenses are deployed. As illustrated in Figure 1, RS contention affects frontend fetch behavior during mis-speculation. When speculative instructions expand into enough μops to create measurable RS contention,

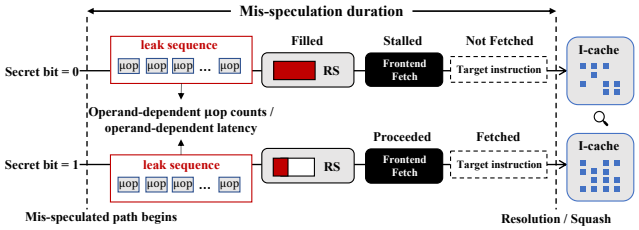

Figure 1: RS-contention-based leakage mechanism. Secret-dependent RS pressure can throttle frontend fetch during a branch-misprediction window. If fetch is throttled, the target instruction on the wrong path is not fetched into the I-cache before the squash; otherwise, it is. This difference in post-recovery I-cache state forms a measurable side channel.

backend pressure throttles fetch, preventing the target instruction from entering the I-cache. Since the target instruction lies on the mis-speculated path, it is never fetched into the I-cache once the misprediction is resolved. Conversely, when RS pressure is low, fetch proceeds normally, allowing the target instruction to be cached. This secret-dependent difference in I-cache state forms a measurable side channel. Our analysis reveals three limitations in selective-speculation defenses that enable secret-dependent RS contention.

First, current defenses adopt a transmit-instruction taxonomy that implicitly emphasizes execution-phase effects and therefore prevents tainted transmit μops from being issued for execution. This focus on execution-phase transmit effects misses instruction classes whose leakage-relevant behavior can arise before issue, such as `REP`-prefixed string operations with operand-dependent microcode expansion. The individual μops themselves do not induce timing variation during transient execution; instead, **the expanded μop sequence creates operand-dependent RS occupancy prior to execution**, thereby exposing a previously unmodeled contention surface.

Second, STT, SPT, and DOLMA adopt a delay-until-resolution strategy to mitigate leakage from secret-dependent control flow by deferring speculative fetch redirects (i.e., misprediction recovery) for tainted branches. This design implicitly assumes that leakage from control dependence is adequately captured by fetch redirects and the resulting wrong-path divergence. However, predicated instructions (e.g., x86 `CMOV` [22], and `czero`-class instructions in RISC-V Zicond [41]) enable secret-dependent value selection without branch resolution. As a result, **attackers can encode secrets into values that control operand-dependent μop expansion** (e.g., `REP` iteration counts) under the delay-until-resolution strategy, allowing the exposed contention surface to be directly exploited.

Third, defenses such as DOLMA and SpecTerminator adopt an older-μop-first allocation strategy that prioritizes older, non-transient μops to prevent backend contention, thereby allowing some instructions that the defenses themselves classify as transmit instructions, such as arithmetic instructions and cache-hit loads/stores, to proceed without delay. This optimization assumes that, in a single thread, unsafe contention is unidirectional: only younger transient μops can interfere with non-transient μops. However, **unprotected arithmetic and cache-hit load μops can still introduce observable operand-dependent execution latency** that, when amplified by dependency chains, creates secret-dependent RS pressure and throttles the fetch of younger transient instructions.

Our Contributions. In this paper, we make the following key contributions:

- We systematically analyze representative selective-speculation defenses [11, 23, 33, 50] and uncover security limitations in their underlying strategies that enable secret-dependent RS contention.
- Guided by these limitations, we construct Spectre-v1-style gadgets that induce secret-dependent RS contention under state-of-the-art selective-speculation defenses. Our gadgets exploit (i) transmit behaviors that fall outside of existing taxonomies via operand-dependent μop expansion (e.g., `REP MOVSB` combined with predicated selection), and (ii) taxonomy-recognized transmit μops that are not delayed (e.g., arithmetic and memory μops), whose operand-dependent latency, amplified through dependency chains, creates RS pressure.
- We demonstrate attack practicality. Using a custom LLVM-based analysis pass [32], we identify RS-contention-relevant patterns in real-world code and demonstrate one such pattern as a transient timing channel with a measurable signal in the commonly used libsodium cryptographic library. We then demonstrate bypasses of STT/DOLMA on x86 and RISC-V gem5 models and validate the μop expansion-based RS-contention effects on Intel Raptor Cove and AMD Zen CPUs.
- We propose and evaluate strengthened STT mechanisms that close both existing and newly exposed gaps while maintaining a moderate performance overhead.

Paper Organization. The rest of the paper is structured as follows: Section 2 introduces the relevant background references that relate directly to this work. Section 3 presents existing selective-speculation schemes and discusses their limitations. Section 4 describes attacks that exploit the identified limitations. Section 5 demonstrates our proof-of-concept (PoC) attacks and real-machine validation results. Section 6 introduces our enhanced STT defense. Section 7 discusses the broader implications and generality of μop-expansion-induced contention across instruction families and microarchitectures. Finally, Section 8 concludes the paper.

2 Background

OoO Processor Pipeline. Modern high-performance processors employ out-of-order execution to exploit instruction-level parallelism (ILP). In the frontend, instructions are fetched, decoded into µops, and dispatched to backend structures. Each µop is inserted into the Reorder Buffer (ROB), which maintains µops in program order to ensure in-order retirement. The µops are also dispatched to the reservation station (RS), which holds them until their operands become ready. Once operands are available, a µop can be issued from the RS to an appropriate execution unit, potentially out of program order. Although execution can proceed out-of-order, the ROB ensures a correct architectural state by committing µops in program order.

Fetch Redirect. Control flow instructions (e.g., conditional branches, indirect jumps, and function calls) alter the sequential flow of program execution, determining the next instruction to fetch. Executing these instructions often depends on operands (e.g., condition flags, target addresses) that may not be immediately available, causing pipeline stalls due to unresolved control dependencies. Data dependencies between instructions can also stall the pipeline. To mitigate these stalls, modern OoO processors employ speculative execution. This involves predicting control flow outcomes and dynamically handling data dependencies to hide stalls. When speculation is incorrect (e.g., due to branch or memory dependency misprediction), the processor must recover. This recovery involves a fetch redirect: the backend signals the fetch unit to resume fetching from the correct program counter (PC). Simultaneously, a squash signal is broadcast to remove invalid speculative instructions from intermediate pipeline stages, ensuring a precise architectural state and correct program execution.

Transient Execution Attacks. Transient execution attacks exploit microarchitectural side effects left during misspeculation, particularly operand-dependent patterns of hardware resource usage, to leak sensitive data. **Spectre-type** attacks [7, 8, 13, 19, 24–26, 34, 35, 39, 51] mistrain the Branch Prediction Unit (BPU) or abuse the Memory Dependency Unit (MDU) or memory-consistency mechanisms to steer victims into semantically incorrect control or data flows. **Meltdown-type attacks** [29, 40, 43–45] exploit delayed exception handling to transiently bypass privilege checks and illegally access kernel data before the fault is architecturally observed. Although transient instructions are ultimately squashed, they can leave measurable traces (e.g., cache state or other resource-usage variations). With suitable gadgets, attackers can observe these microarchitectural traces and infer secrets from transiently accessed memory or from registers holding non-speculative results [23, 50].

Speculative Side Channels. Speculative side channels arise when secret-dependent transient execution leaves observable microarchitectural traces, such as variations in hardware resource usage [50]. Depending on whether secrets directly or indirectly affect instruction behavior, speculative side channels are categorized as explicit or implicit.

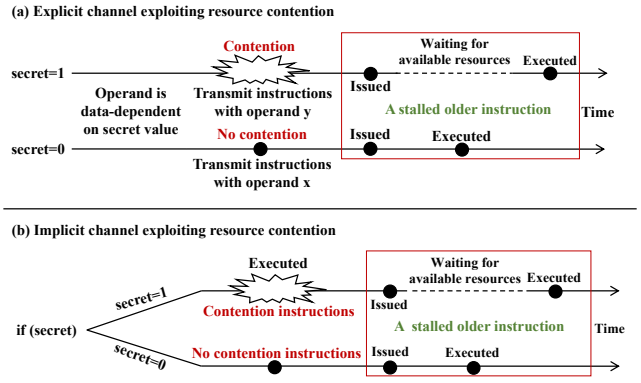

Figure 2: Example transient execution attack utilizing (a) explicit or (b) implicit channel to exploit backend resource contention. This contention delays the execution of an older, already-stalled instruction. The resulting timing interference leaks information about (a) the transmit instruction's operand or (b) the control predicate.

In the case of **explicit channels**, secrets directly influence transmit instructions whose operand-dependent resource usage modifies microarchitectural states. For example, in cache-based side channels [9, 12, 16, 17, 42, 49], memory access instructions with their addresses dependent on the secret serve as the transmit instructions. Their speculative execution alters the cache state, leading to secret-dependent timing variations. Similarly, in backend contention channels [7, 48], as shown in Figure 2, secret-dependent arithmetic or memory instructions contend for limited resources (e.g., MSHR entries). Their operand-dependent resource usage patterns delay the retirement of older instructions.

In the case of **implicit channels**, the secret indirectly affects the execution state of one or more instructions. Existing defenses identified the following two variations in hardware resource usage that encode the secret: (1) **Side channels based on secret-dependent prediction outcomes:** Attackers induce secret-dependent prediction outcomes to steer instruction streams along different transient paths. For control flow prediction, secrets are encoded into control predicates that determine execution paths [4, 8, 13, 39, 51]. For example, in resource contention-based channels, as illustrated in Figure 2 (b), secret-dependent control predicates determine whether a code path containing resource-contending instructions is executed. For memory address prediction, secrets could be encoded into outcomes such as store-to-load forwarding, determining whether a memory access issues to the cache hierarchy. (2) **Side channels based on secret-dependent predictor state:** Secrets can be used to manipulate the internal

Table 1: Taxonomy and coverage of state-of-the-art selective speculation defenses.

Side channel	Defense strategy	Target / Vulnerability		NDA	SpecShield	STT	SPT	DOLMA	SpecTerminator
-	Limit the propagation of speculative data	Speculative-data-dependent µops		✓	✓	-	-	-	-
Explicit	Delay the issue of transmit instructions	Tainted-data-dependent memory access µops	Loads under cache/TLB hits	-	-	✓	✓	✗	✗
			Loads under cache/TLB misses	-	-	✓	✓	✓	✓
			Stores under TLB hits	-	-	✗	✗	✗	✗
			Stores under TLB misses	-	-	✗	✗	✓	✓
		Tainted-data-dependent arithmetic µops		-	-	✓	✓	✗	✗
		REP-prefixed string µops		-	-	✗	✗	✗	✗
	Older-µop-first resource allocation	Tainted-data-dependent resource contention	Younger-to-older resource contention	-	-	-	-	✓	✓
			Older-to-younger resource contention	-	-	-	-	✗	✗
Implicit	Delay the tainted-data-dependent prediction resolution (Delay-until-resolution)	Tainted-data-dependent memory address prediction outcomes		-	-	✓	✓	✓	✓
		Tainted-data-dependent predicated-execution outcomes		-	-	✗	✗	✗	-
	Delay the issue of tainted-data-dependent branch instruction	Tainted-data-dependent control prediction outcomes		-	-	✓	✓	✓	✓
				-	-	-	-	-	✓
		Tainted-data-dependent predicated-execution outcomes		-	-	-	-	-	✗
	Block tainted data dependent predictor update	Tainted-data-dependent BPU/MDU state		-	-	✓	✓	✓	-

* - indicates that the defense does not employ the corresponding defense strategy. ✗ indicates that the corresponding defense strategy cannot protect against the associated target/vulnerability, whereas ✓ indicates that it can. Blue text highlights the newly identified limitations uncovered in this work.

state of various prediction units (e.g., BPU, MDU), influencing subsequent speculation [18, 27, 30, 31, 35]. For instance, a secret used during mis-speculation may train the BPU to favor a particular path, or modify MDU entries to produce secret-dependent speculation behavior in later memory operations.

Selective Speculation Defenses. Such defense schemes [6, 11, 23, 33, 46, 50] aim to prevent sensitive data leakage across speculative side channels by selectively blocking the propagation of output values from speculative µops, i.e., speculative data. The source of speculative data varies across threat models: For Spectre-type attacks, speculative µops originate from unresolved control flow or memory dependencies. Their output values become non-speculative only after the corresponding branch or dependency resolves, determining when the blocking should end. For Meltdown-type attacks, speculative µops are typically data-dependent on transient loads or load-like privileged register reads that may raise exceptions resolved only at retirement. Consequently, these µops remain speculative until they retire. This means the corresponding µops must be delayed for a longer period, which inevitably incurs significant performance overhead. Meltdown-type attacks are typically modeled using the conservative variants of these schemes, such as DOLMA-Conservative or the Futuristic model of STT. Due to differing interpretations of speculative side-channel threats, each scheme selectively blocks certain propagation paths of speculative data and delays different sets of µops. These differences are further examined in our following analysis.

3 Systematic Analysis and Uncovered Limitations of Selective Speculation Defenses

As summarized in Table 1, early selective speculation defenses, such as NDA [46], enforced a strict data-propagation policy that completely blocks the propagation of speculative data, effectively preventing leakage through both registers and memory. Later permissive variants, such as SpecShield [6], relaxed this rule by blocking only memory-speculative data, i.e., data derived from speculative loads, to reduce the performance penalty. However, both strict and permissive variants disrupt ILP to varying degrees and impose substantial performance overheads. Schemes including STT, SPT, DOLMA, and SpecTerminator, refined this principle by allowing partial propagation of speculative data. They share a common design philosophy: only the instructions that may transmit secret data, referred to as transmit instructions, are delayed. They exploit hardware taint tracking techniques to follow speculative data propagation and delay transmit µops that consume tainted operands. Yet, these schemes differ in their taint initialization policies: STT, DOLMA (M), and SpecTerminator mark only speculatively accessed data as a tainted source, while SPT and DOLMA (M+R) conservatively initialize all program data as tainted sources to further protect secrets residing in registers. These policies differ in which values are initially marked as tainted, but once an operand is tainted, the relevant question for our analysis is how each defense classifies and delays transmit µops that consume it. Since our following analysis focuses on these downstream taxonomy and scheduling assumptions, we do not distinguish between taint-initialization policies in the remainder of the paper unless necessary.

For **explicit channels**, STT and SPT delay the issue of loads and arithmetic µops with tainted operands, as these µops act as transmit instructions with operand-dependent resource usage. To reduce performance overhead, DOLMA and SpecTerminator narrow the set of delayed instructions to loads and stores with tainted data that incur TLB or cache misses, leaving other memory access and arithmetic µops unguarded. But they extend protection to speculative stores that may trigger TLB-based side effects. To mitigate operand-dependent out-of-order resource contention from these overlooked transmit µops, DOLMA and SpecTerminator adopt

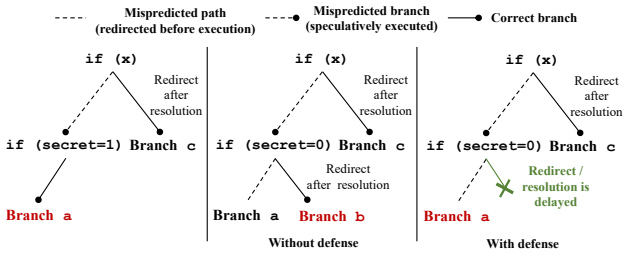

Figure 3: An example of defending against speculative side channels based on secret-dependent control flow. The red labels indicate the branch taken under different secret values. Without defense, a secret value of 1 leads to the execution of branch `a`, while a value of 0 leads to branch `b`. With defense enabled, the resolution of the secret-dependent branch is delayed until the relevant speculation is resolved; therefore, any fetch redirect caused by that branch outcome is also delayed. As a result, during the mis-speculation, execution remains on the same nested branch `a`, regardless of the secret.

an older-μop-first resource allocation strategy, which grants backend resources to the older, non-transient μop when contention occurs, ensuring that younger speculative micro-ops cannot delay non-transient ones.

For **implicit channels**, since secrets indirectly affect the execution state of μops without direct data dependencies, these defenses cannot rely solely on delaying specific tainted data-dependent transmit instructions. These schemes employ two complementary strategies: (1) For **side channels based on secret-dependent prediction outcomes**, STT, SPT, and DOLMA delay the resolution of predictions that depend on tainted data (referred to as delay-until-resolution in this paper). This allows nested speculative execution to proceed using the current predictor state rather than dependent operands. DOLMA adopts the same principle but provides a concrete microarchitectural implementation: it delays fetch redirects triggered by speculative operands, thereby deferring related prediction resolutions. For control flow predictions, SpecTerminator adopts a stricter policy: it directly delays the issue of branch instructions whose predicates depend on tainted data, which inherently also prevents speculative updates to the BPU. For memory address predictions, SpecTerminator still employs the delay-until-resolution policy. (2) For **side channels based on secret-dependent predictor state updates**, except for SpecTerminator, other schemes block speculative updates to hardware predictors (e.g., BPU, MDU) when such updates depend on tainted data, preventing speculative modification of predictor states that could encode secrets.

Limitation I: A blind spot in the taxonomy of transmit instructions. State-of-the-art selective speculation defenses define their transmit taxonomies at different granularities. Some operate at the instruction level, identifying instructions that cause operand-dependent execution resource usage; others operate at the μop level, identifying μops that induce observable operand-dependent timing variation during transient execution. For example, the transmit taxonomy introduced by STT classifies instructions as transmit if "their execution creates operand-dependent resource usage that can reveal the operand." DOLMA operationalizes transmit μops through "speculative micro-ops whose operand values can be transmitted during transient execution via corresponding timing variations." Despite these differences, both methodologies restrict transmit instructions as those having operand-dependent execution effects that can begin only after the instructions or μops are issued. Consistently, these defenses delay the issue of tainted transmit μops until resolution, preventing them from executing. Furthermore, μop-level descriptions implicitly assume that operand-dependent timing variation arises from the transmit μop's transient execution.

However, these assumptions overlook transmit channels that arise during the decode and dispatch phases, where operand values can influence μop expansion and thus the degree of RS occupancy before execution. For instance, microcode-expanded instructions, such as `REP`-prefixed string instructions, generate operand-dependent μop sequences, increasing RS occupancy at dispatch. The individual μops need not exhibit operand-dependent execution latency during transient execution; rather, the elevated RS occupancy directly delays frontend fetch of subsequent transient μops. Such instructions therefore fall outside the current transmit taxonomies. Delaying the issue of transmit instructions is insufficient to prevent their pre-execution microarchitectural footprint.

Limitation II: Delay-until-resolution misses predicated secret-dependent selection. To mitigate leakage from secret-dependent control-flow predicates, state-of-the-art selective speculation schemes uniformly rely on delaying prediction resolution to prevent speculative path divergence. In their gem5 implementations, all defenses defer fetch redirects and flush signals caused by secret-dependent mispredictions. This approach increases only the latency of recovering from a tainted branch misprediction [50], compared to delaying all tainted branches, but it implicitly treats redirect-based path divergence as the primary manifestation of control-dependent leakage. For example, in nested control flows (Figure 3), an attacker can extend the outer speculation window such that an inner branch resolves early. When the secret is 0, the predictor initially fetches along path `a`; but the inner branch resolves to path `b` before the window closes, causing a secret-dependent fetch redirect that leaks the nested control flow predicate. Existing defenses correctly block such explicit control flow divergence, ensuring that the nested control flow remains on branch `a` until resolution.

However, this mechanism does not account for predicated instructions, which realize secret-dependent control semantics (conditional updates/value selection) without any branch resolution or fetch redirection. Such instructions derive their predicates from flags or registers set by compare/set-

flags operations (e.g., `CMP`) and may execute transiently even when tainted branches are constrained. The stricter approach in SpecTerminator, which delays the issue of tainted data-dependent branch instructions, likewise fails to cover such predicated instructions. As a result, predication can inject secret-dependent values into operands that steer contention primitives (e.g., selecting `REP` µop-expansion paths or driving long dependency chains), enabling the predicate (and thus the secret) to be inferred without any fetch redirect.

Limitation III: Bidirectional resource contention beyond older-µop-first allocation. To preserve performance, DOLMA and SpecTerminator do not delay a subset of transmit instructions, including arithmetic instructions and speculative load/store hits. Since such µops can still alter the retirement timing of older µops through out-of-order contention for core-local resources, both schemes adopt older-first resource allocation to obviate such unsafe out-of-order resource contention in a single thread. This design, however, assumes that tainted-data-dependent contention only occurs between a younger (transient) µop and an older (non-transient) µop.

In practice, older speculative µops can also exploit RS contention to delay the fetch of younger µops. Although the speculative RS occupancies are eventually squashed, their transient resource-usage effects alter the fetch timing of younger instructions, effectively converting transient contention into persistent and observable I-cache occupancy variations. Unlike prior assumptions that contention is observable only when it affects the retirement timing of older instructions, these I-cache state differences no longer rely on committed instructions. As a result, speculative-data-dependent contention becomes measurable even when the affected µops remain transient. This breaks the unidirectional assumption behind the older-µop-first allocation policies used in DOLMA and SpecTerminator: older speculative arithmetic and cache-hit memory-access µops can still induce observable younger-fetch effects, even when younger-to-older contention is prevented.

Table 2: Performance counter measurements conducted on the Intel Core i9-13900K.

RCX	IDQ. MITE_UOPS[1]	IDQ. MS_UOPS[2]	UOPS. _ISSUED	UOPS_RETIRE .SLOTS	RECOVERY_ CYCLES[3]	MACHINE_ CLEARS[4]
"LEA RSI, src_addr; LEA RDI, dst_addr; MOV RCX, test_value; REP MOVSB"						
0	7.00	18.01	25.01	25.01	0	0
64	7.00	18.01	25.01	25.01	0	0
128	7.00	18.01	25.01	25.01	0	0
256	7.00	48.01	55.01	55.01	0	0
320	7.00	65.01	72.01	72.01	0	0
512	7.00	75.01	82.01	82.01	0	0
832	7.00	99.00	106.00	106.00	0	0
1536	7.00	107.00	114.00	114.00	0	0
2048	7.00	123.00	130.00	130.00	0	0
2560	7.00	139.00	146.00	146.00	0	0
3008	7.00	167.00	174.00	174.00	0	0
"LEA RSI, src_addr; LEA RDI, dst_addr; MOV RCX, test_value; REP MOVSW"						
0	7.00	78.01	85.01	43.01	5.00	0
"LEA RSI, src_addr; LEA RDI, dst_addr; MOV RCX, test_value;"						
-	3.00	0	3.00	3.00	0	0
"std; lfence; cld;"						
-	0	44	44	32	9.28	0

[1] IDQ.MITE_UOPS: Number of µops delivered to IDQ via the legacy decode pipeline.
[2] IDQ.MS_UOPS: Number of µops dispatched from MS to IDQ.
[3] RECOVERY_CYCLES: Cycles spent recovering from branch mispredictions or pipeline flushes.
[4] MACHINE_CLEARS: Number of machine clears due to memory ordering conflicts.

4 Exploiting Hidden Resource Contention in Selective Speculation Defenses

Attacker model. Our attacker operates within the threat model adopted by state-of-the-art selective-speculation defenses [11, 23, 33, 50] and is no stronger than the adversaries considered in these works. Specifically, the attacker can (i) induce or extend speculative/transient execution so that victim code transiently accesses secrets; (ii) mistrain branch predictors or otherwise steer speculative execution along attacker-influenced paths; and (iii) observe and exploit arbitrary microarchitectural timing-based covert channels to recover secret-dependent information from the resulting transient side effects. We assume that an attacker is co-located with the victim on the same physical core and sharing microarchitectural resources such as frontend structures and backend queues. Because the precise scope varies across defenses (e.g., DOLMA explicitly excludes SMT-based attacks), we adopt the most restrictive setting and limit the adversary to a single hardware thread. While SMT could amplify the signal, our attacks do not require SMT capabilities.

Under this model, we instantiate Spectre-v1-style gadgets that exploit the limitations identified in Section 3. We show that secret-dependent resource contention can persist even under state-of-the-art selective-speculation defenses, either through transmit instructions overlooked by existing defense taxonomies or through transmit instructions recognized by existing defense taxonomies but left to proceed under optimized policies, resulting in measurable side-channel leakage.

4.1 Exploiting Transmit Instructions Overlooked by Existing Defense Taxonomies

As discussed in Section 3-Limitation I, operand-dependent µop expansion falls outside current execution-phase transmit taxonomies, yet can still induce measurable RS contention. One representative example is the x86 `REP`-prefixed string-instruction family. Instructions such as `REP MOVSB` and `REP MOVSW` copy memory blocks from the address in `RSI` to the address in `RDI` for a number of iterations specified by `RCX`; these instructions are expanded into multiple µops by the Microcode Sequencer (MS). Using `REP MOVSB` as a representative example, we demonstrate that operand-dependent µop expansion in `REP`-prefixed string instructions can induce RS contention that survives state-of-the-art selective-speculation defenses, even when defenses such as STT and SPT delay all

transmit instructions under their execution-phase taxonomies.

Exploitable characteristics of `REP MOVSB` under selective speculation defenses. Prior work [51] has noted that the string operations with a repeat prefix consume varying amounts of hardware resources, depending on the number of repeat iterations (e.g., RCX value). Although the exact microarchitectural impact was unclear, it was generally assumed that repeat-string operations contend for memory-access resources, indirectly influencing subsequent memory operations and enabling D-cache side channels.

Conventionally, the microcode implementation of `REP MOVS`-class instructions is believed to include internal micro-branches that manage the repetition loop. If so, the delay-until-resolution strategy could block speculative, secret-dependent micro-branches, forcing `REP MOVSB` to follow a single execution path during speculation regardless of the RCX value. However, our experiments on Intel Raptor Cove P-cores reveal otherwise. We find that `REP MOVSB` exhibits properties that not only induce RS contention but also suggest a microcode implementation without mispredicted micro-branches. Using nanoBench [1], we collect performance counter data, including the number of dispatched/retired µops, and misprediction-related metrics (e.g., recovery cycles and machine clears), after executing the sequence "`LEA RSI, src_addr; LEA RDI, dst_addr; MOV RCX, test_value; REP MOVSB`". The results, partially summarized in Table 2, yield two key observations:

(1) **The number of µops dispatched from the MS to the Instruction Decode Queue (IDQ) varies with the `RCX` value.** For example, for RCX values smaller than 128, the MS_UOPS count remains around 18, whereas it increases to around 75 when RCX=512.

(2) **Execution of `REP MOVSB` involves no mispredictions.** As shown in Table 2, the number of issued µops equals the number retired, indicating no transiently issued operations, and both misprediction-related counters (RECOVERY_CYCLES and MACHINE_CLEARS) remain zero. In contrast, the `CLD` instruction, whose microcode has been reverse-engineered [36] and shown to contain a micro-branch, exhibits nonzero transiently issued operations and recovery cycles, as shown in Table 2. Our experiments show that `CLD` consistently exhibits mispredictions, while `REP MOVSB` does not exhibit any mispredictions across thousands of iterations. Therefore, we conclude that `REP MOVSB` executes without speculation, rendering delay-until-resolution defenses ineffective against this class of instructions. Although it remains challenging to precisely characterize Intel's internal implementation of `REP MOVSB` without conditional micro-branches, our results suggest that the observed behavior is likely related to the Enhanced `REP MOVSB` and `STOSB` (ERMSB) [21] mechanism to accelerate string copy operations. In contrast to `REP MOVSB`, `REP MOVSW` without ERMSB optimization incurs mispredictions, as shown in Table 2. Furthermore, our observations suggest that the number of dispatched µops remains constant within specific RCX ranges (e.g., 25 µops for RCX<128 and 55 µops for RCX<256). This piecewise-constant behavior implies that ERMSB may use fixed µop paths for certain copy sizes to avoid the performance penalty of mispredicted micro-branches in `REP MOVSB`.

```
Pseudocode:     if condition
                   do A
                else
                   do B
```

(b) Predication

```
CMP condition
Predicate(condition)do A
Predicate(not condition)do B
```

(a) Conditional branch

```
CMP condition
// if condition is false, jump to label1
JNE label1
  do A     // condition is true
JMP end    // branch always to end
label1:
  do B     // condition is false
end
```

Figure 4: Comparison of conditional branch vs. predicated execution for identical pseudocode. (a) illustrates conditional branch pseudo-assembly. (b) shows predicated execution pseudo-assembly.

Attackers can exploit this operand-dependent microarchitectural behavior in Observation (1) during mis-speculation to increase RS occupancy and create contention for subsequent instructions. A `REP MOV` instruction crafted with a secret-dependent RCX value can cause secret-dependent RS contention, which in turn delays the fetch of subsequent instructions. This delay forms a side channel that leaks the secret.

Bypassing delay-until-resolution. The remaining challenge is how to construct a secret-dependent selection of RCX in the presence of delay-until-resolution. Since arithmetic encodings of secret-dependent selections would likewise be classified as transmit µops and delayed by selective-speculation defenses such as STT and SPT, such encodings cannot be used to drive the `REP MOVSB` operand. Limitation II in Section 3 becomes relevant here: delay-until-resolution suppresses secret-dependent speculative fetch redirects, but does not account for predicated data selection.

ISAs such as x86, ARM [5], and RISC-V support predicated instructions to enable conditional execution without explicit control-flow divergence. As illustrated in Figure 4, predication associates each instruction in a control-dependent region with the control predicate that determines whether it should execute and update architectural state. For example, Intel x86 provides the conditional-move instruction family, such as `CMOVE` (move if equal) and `CMOVNE` (move if not equal), which test the relevant condition codes in `EFLAGS` (e.g., `ZF`) before performing the move. RISC-V's Zicond extension introduces predicated instructions such as `czero.eqz` (clear if equal to zero) and `czero.nez` (clear if not equal to zero), which determine execution directly based on the value of a source register. Crucially, predicated execution enables secret-dependent data selection without triggering control-flow redirection, thereby bypassing delay-until-resolution and encoding the secret into RCX, which in turn selects different `REP MOVSB` microcode/µop-expansion paths.

Gadget 1: Secret-dependent repeat move string instructions. In the gadget shown in Figure 5, we use `CMOVNE` to

Gadget 1

```
01 if(condition){// boundary check
02 // misprediction
03 MOV large_value,RDX
04 MOV small_value,RCX
05 CMP secret_bit,0
06 CMOVNE RDX,RCX // conditionally move RDX to RCX
07 REP MOVSB // copy RCX bytes from RSI to RDI
08 target inst // target instruction }
```

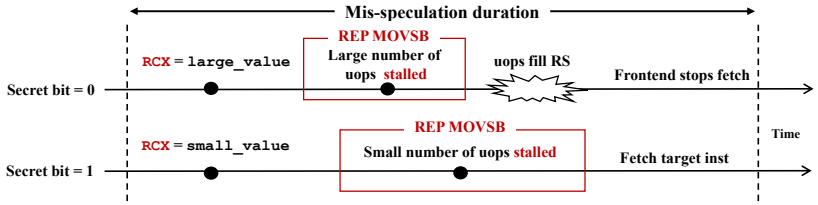

Figure 5: Left: Gadget 1 exploiting `REP MOVSB` instruction. Right: Timeline of secret-dependent RS contention caused by `REP MOVSB` during mis-speculation. The `RCX` value (controlling `REP MOVSB` iterations) depends on the secret bit. Large `RCX` values generate excessive µops that saturate RS entries, stalling frontend fetch and creating an I-cache side channel.

make the repeat count of `REP MOVSB` depend on the secret bit. In our example, `RCX` is initialized to a small iteration count, while a large iteration count is prepared in another register. The `CMP` instruction sets `ZF` depending on whether the secret bit equals 0. If `ZF` is not set, corresponding to secret bit 1, `CMOVNE` copies the large iteration count into `RCX`; otherwise, `RCX` remains at the small iteration count. The speculative `REP MOVSB` instruction then uses this secret-dependent `RCX` value as its implicit repeat-count operand:

(1) When the secret bit is 1, `RCX` is set to a large value in our experiment, e.g., 3008, causing `REP MOVSB` to generate a larger number of MS_UOPS. Since the source and destination addresses have been flushed, these µops cannot issue promptly and remain in the scheduler for an extended period, exhausting RS entries. The resulting RS pressure stalls frontend fetch during misprediction, preventing the target instruction from entering the I-cache before resolution.

(2) When the secret bit is 0, `RCX` is set to a small value in our experiment, e.g., 4, resulting in a small number of MS_UOPS generated by the `REP MOVSB` instruction. These µops do not exhaust RS entries during the mis-speculation, and frontend fetch proceeds normally.

4.2 Exploiting Transmit Instructions Recognized by Existing Defense Taxonomies

As discussed in Section 3-Limitation III, modern selective speculation defenses such as DOLMA and SpecTerminator adopt an older-first resource allocation strategy to avoid explicit out-of-order backend contention introduced by arithmetic and cache-/TLB-hit memory µops that are recognized as transmit instructions by existing defense taxonomies. This strategy assumes that the explicit resource contention is unidirectional in single-thread execution, i.e., a younger (transient) µop contends with an older (non-transient) µop.

However, older speculative µops that exhibit operand-dependent occupancy of dispatch resources can delay the fetch of younger transient µops, which in turn alters the observable I-cache state. Below, we demonstrate this idea with two gadgets exploiting arithmetic and memory-access transmit instructions that cause a data leakage under the older-µop-first allocation strategy of selective speculation defenses:

Gadget 2

```
01 if(condition){// boundary check
02 // expand secret bit to full-width mask
03 mask = extend(secret_bit);
04 // bitwise selection
05 tmp = (slow_value&mask) | (fast_value&~mask) ;
06 // leak sequence
07 tmp = MULSD(tmp,tmp);
08 tmp = SQRTSD(tmp); tmp = MULSD(tmp,tmp);
09 ...
10 tmp = SQRTSD(tmp); tmp = MULSD(tmp,tmp);
11 target inst; // target instruction }
```

Figure 6: Gadget 2 exploiting variable-time arithmetic instructions. A secret bit selects fast or slow operands via bitwise operations, creating a speculative arithmetic chain with secret-dependent RS pressure.

Gadget 2: Secret-dependent, variable-time arithmetic instructions. Certain floating-point instructions on x86 exhibit input-dependent execution latencies: some operand values execute substantially slower than others (so-called "fast" vs "slow" values; e.g., the values 65536 and 2.34e-308 as discussed in [37]). Prior work [51] constructs a leakage sequence (e.g., `SQRTSD` followed by `MULSD`) whose overall latency depends on whether the operands are chosen as fast or slow values under a secret-dependent branch (e.g., those introduced by `if` statements or ternary expressions). Instead of relying on branches, we implement operand selection using arithmetic instructions that are not delayed by DOLMA/SpecTerminator. As illustrated in Figure 6, the secret bit is first broadcast into a full-bit mask, which is then used to perform a branchless, bitwise conditional selection between slow and fast values. Selecting the slow value prolongs the dependent arithmetic chain and increases RS occupancy, whereas selecting the fast value allows the chain to progress with lower backend pressure. In short, this gadget realizes secret-dependent operand readiness entirely through arithmetic instructions that remain undelayed in DOLMA/SpecTerminator, and yields measurable RS contention as in Gadget 1.

Gadget 3: Secret-dependent, variable-time memory-access instructions. Defense schemes such as DOLMA and SpecTerminator only delay loads under cache misses to optimize performance overhead. However, speculative

Gadget 3

```
01 if(condition){// boundary check
02 // array[0] is pre-cached
03 // array[miss_index] is pre-flushed
04 addr = miss_index*secret_bit;
05 tmp = array[addr];
06 // leak sequence
07 tmp = tmp + tmp; tmp = tmp + tmp;
08 ...
09 tmp = tmp + tmp; tmp = tmp + tmp;
10 target inst; // target instruction }
```

Figure 7: Gadget 3 exploiting memory-access instructions. The secret bit selects between a cached and a flushed array index, producing hit/miss-dependent operand readiness that propagates through the dependent arithmetic chain and results in secret-dependent RS contention.

loads whose address depends on the secret can still influence operand readiness time. As illustrated in Figure 7, the secret bit selects between two `array` indices through a simple multiplication, yielding either a cache hit or miss. When `secret_bit = 1`, the selected load from `array[miss_index]` misses in the cache. The miss delays the readiness of `tmp`, which stalls the dependent arithmetic chain and prolongs RS occupancy. Furthermore, because selective speculation defenses block the issue of speculative miss requests, the effective miss latency is further amplified. In contrast, when `secret_bit = 0`, the selected load from `array[0]` hits in the cache. Operand readiness is reached earlier, so fetch proceeds normally. As a consequence, the target instruction exhibits a different I-cache residency across the two cases.

Predicated value-selection variants of Gadget 2 and Gadget 3. In both gadgets, the secret is ultimately encoded through a value-selection step: fast vs. slow operands (arithmetic) or hit vs. miss addresses (memory). The same semantics can also be expressed as semantically conditional statements (e.g., `if-else`). However, compilers, depending on optimization settings and backend (e.g., GCC vs. Clang/LLVM), may realize this selection using either branch-based or predicated lowering. These two lowerings are semantically equivalent but differ microarchitecturally under selective-speculation defenses. Under delay-until-resolution, the branch-based selector is stalled until resolution and therefore fixes the selected value during speculation, eliminating downstream timing variation. In contrast, predicate-based selectors (e.g., `CMOV`-class on x86 or `czero`-class under RISC-V Zicond) perform secret-dependent data selection without control-flow redirection, thereby bypassing delay-until-resolution. Consequently, the secret can still be encoded into fast/slow operands (arithmetic) or hit/miss addresses (memory), preserving the RS leakage channel. This compiler-induced, microarchitectural divergence means that source-level code that appears safe under selective speculation can compile into gadgets that bypass the delay-until-resolution strategy.

Summary. The constructions above illustrate how the optimized assumptions in selective-speculation defenses are incomplete. Gadget 1 combines Limitation I and Limitation II: predicated selection injects the secret into the `RCX` operand without triggering a fetch redirect, and the `REP`-prefixed instruction then turns that operand into pre-execution μop expansion and RS pressure. Gadgets 2 and 3 exercise Limitation III: arithmetic and memory-access μops left undelayed under older-μop-first allocation can still create secret-dependent operand-readiness differences, which are amplified by dependency chains into observable RS pressure. Table 3 summarizes the broken assumptions and corresponding bypass mechanisms.

4.3 Existence of RS Contention Gadgets

To validate that such gadgets can occur in real-world software, we develop an LLVM-based leakage-detection pass. The pass performs lightweight taint tracking from function arguments and maintains a set of tainted registers. It models both *explicit* propagation, where taints flow through explicit operands, and *implicit* propagation, where taints flow through implicit operands such as condition flags across predicated instructions (e.g., flag-setting operations followed by conditional moves). We intentionally do not attempt to detect purely variable-time arithmetic gadgets, since the fast vs. slow operand values used in prior work are highly microarchitecture-specific and do not generalize well for large-scale searching. Instead, our pass finds candidate functions that satisfy the following rules.

(1) Speculation trigger. The function either (i) contains at least one conditional branch that can be mistrained in a Spectre-v1-style setting, or (ii) is reachable as an indirect-call/jump target in contexts that enable Spectre-v2-style control-flow manipulation (e.g., after predictor training).

(2) Transmit primitive: Variable-time memory access. Along at least one path, tainted data is propagated into a memory access address; the loaded value is then used by a sequence of arithmetic instructions to amplify operand-readiness differences and induce RS contention.

(3) Transmit primitive: `REP-MOVSB`. Tainted data is implicitly propagated to the arguments of a `memcpy`-style routine.

(4) Post-primitive observable footprint. After the transmit primitive, there exists at least one subsequent control-transfer, such as a `call`, `ret`, or indirect `jmp`, so that secret-dependent fetch throttling can translate into measurable I-cache effects.

Note that because we do not have any a priori method for determining which values are secret, the gadgets found do not necessarily leak actual secrets. However, they indicate locations where leakage could potentially occur under speculation. Table 4 reports the results of this search.

As an illustrative example, our pass finds the libsodium function `ristretto255_frombytes` as a candidate RS-contention gadget. In the compiled code, most field-

Table 3: Summary of defense assumptions and bypass techniques

Lim.	Defense assumption	Bypass technique	Instantiated by
I	Transmit effects arise only after issue.	Use operand-dependent µop expansion to create pre-execution RS pressure.	`REP`-prefixed string instructions.
II	Control leakage is captured by fetch redirection.	Use predicated selection to inject secrets into operands without redirecting fetch.	`CMOV`/Zicond-based operand selection.
III	Older-µop-first allocation prevents unsafe contention.	Use older speculative µops to create RS pressure that delays younger fetch.	Variable-time arithmetic and memory-load dependency chains.

element helpers (e.g., `fe25519_frombytes`, `fe25519_sq`, `fe25519_add/sub`) are inlined, forming a long contiguous arithmetic chain whose operand readiness depends on the input `s`. The first out-of-line boundary that follows this chain is an explicit `call` to `fe25519_mul`, providing a concrete post-primitive probe point.

We build a PoC that turns this pattern into a 1-bit transient timing channel on a real Intel Xeon Gold 6248R system running Ubuntu 22.04. The `ristretto255_frombytes()` function begins with a canonicality check `ristretto255_is_canonical(s)`; we choose a non-canonical `s` and mistrain the branch so that execution transiently follows the valid path. During this transient window, the inlined arithmetic chain executes and can create RS pressure that throttles frontend fetch. To encode a single bit, two microarchitectural states are prepared that differ only in the cache residency of selected cache lines backing `s`. When `s` is cold, the chain experiences longer operand-ready latencies, increasing RS occupancy and reducing the likelihood that the core fetches the subsequent `fe25519_mul` call within the transient window. Since `fe25519_mul` is an internal helper function, both training and timing use normal calls to `ristretto255_frombytes`. The reported signal is the end-to-end timing difference of a subsequent invocation of `ristretto255_frombytes`. In this controlled setup, the main post-chain difference between the two classes is whether `fe25519_mul` is fetched into the I-cache within the transient window: the flush of `ristretto255_is_canonical` is symmetric across classes, whereas hot vs. cold `s` changes the amount of backend pressure and thus the likelihood of fetching `fe25519_mul`. Across 1000 trials per class, the two states are distinguishable with a median latency gap of 549 cycles. A single-threshold classifier achieves 71.25% accuracy, confirming an exploitable signal.

Importantly, our PoC does not assume that `s` is a long-term secret in libsodium's intended usage; rather, it shows that if an attacker can transiently steer or select `s` (e.g., making it secret-tainted or controlling its cache residency), the resulting `s`-dependent arithmetic chain can induce measurable RS pressure. This case study demonstrates that real-world cryptographic code can contain concrete RS-pressure primitives whose effects are externally observable, even when the function is architecturally constant-time.

```
01 int ristretto255_frombytes(ge25519_p3 *h, const unsigned char *s){
02     ...
03     if (ristretto255_is_canonical(s) == 0) {
04         return -1;
05     }
06 //The following fe25519_* helpers are inline-expanded by the compiler
07     fe25519_frombytes(s_, s);
08     fe25519_sq(ss, s_);                  /* ss = s^2 */
09     fe25519_1(u1);
10     fe25519_sub(u1, u1, ss);             /* u1 = 1-ss */
11     fe25519_sq(u1u1, u1);                /* u1u1 = u1^2 */
12     fe25519_1(u2);
13     fe25519_add(u2, u2, ss);             /* u2 = 1+ss */
14     fe25519_sq(u2u2, u2);                /* u2u2 = u2^2 */
15 //The following fe25519_mul is invoked as a regular function call
16     fe25519_mul(v, d, u1u1);             /* v = d*u1^2 */
17     ...}
```

Figure 8: `ristretto255_frombytes` in libsodium 1.0.20.

5 PoC Demonstration

In this section, we evaluate the PoC demonstrations introduced in Section 4. The goal is to show that speculative RS contention remains inducible under the strongest security-oriented configurations of state-of-the-art selective-speculation defenses, thereby exposing the three security limitations identified in Section 3. Our evaluation is based on the publicly released code of representative defenses, including STT and DOLMA. The evaluation uses OoO cores modeled for both x86 and RISC-V ISAs, with detailed configuration parameters summarized in Table 4. All PoCs follow the core principle of Spectre-V1 [25]: the Pattern History Table (PHT) is mistrained to induce a branch misprediction and open a transient window on a mis-speculated path.

Three sets of simulation-based experiments and one real-hardware validation are conducted. In simulation, (i) a `REP MOVSB`-based gadget is evaluated under STT to demonstrate that taxonomy-overlooked µop expansion and predicated selection can still induce RS contention (Limitations I–II); (ii) a variable-latency load gadget is evaluated under DOLMA to show that older-µop-first allocation does not eliminate RS-contention leakage in a single thread (Limitation III); and (iii) branch-based and predicated selectors are compared to highlight that the effectiveness of delay-until-resolution depends on the compiler realization. Finally, the `REP MOVSB`-induced RS-pressure primitive is validated on real Intel processors with Raptor Cove P-cores, confirming that operand-dependent

Table 4: Number of gadgets found in common software libraries.

	OpenSSL 1.1.1q	Libgcrypt 1.11.0	libsodium 1.0.20	OpenBLAS 0.3.0	musl 1.2.5	zlib 1.3.1	wolfssl 5.8.4
variable-time memory access gadgets	52	77	46	24	26	5	15
`REP-MOVSB` gadgets	28	13	0	0	10	9	10

Table 5: gem5 parameters.

Parameter	gem5
ISA	x86_64 & RV64
Processor type	8-decode 8-issue DerivO3 CPU
ROB/LDQ/STQ	192/32/32 entries
L1I Cache	32KB, 8-way, 64B line, 4 MSHRs
L1D Cache	32KB, 8-way, 64B line, 4 MSHRs
L2 Cache	256KB, 16-way, 64B line, 20 MSHRs

µop expansion can create substantial RS pressure and a robust I-cache side effect on real hardware.

Simulator extensions. To support our evaluations, we make two targeted simulator-side extensions. First, gem5's default x86 microcode for `REP MOVS`-class instructions uses an explicit microcode loop that introduces micro-branching (and thus potential mispredictions), which does not match the behavior of `REP MOVSB` with ERMSB optimization observed on real Intel processors (Section 4.1). We therefore replace the loop-based `REP MOVSB` microcode with a branch-less implementation that selects fixed µop paths for different `RCX` ranges, better reflecting the observed behavior. Second, as the gem5 versions used by the STT and DOLMA artifacts lack support for the RISC-V Zicond extension, we implement `czero`-class instructions in the simulator to evaluate predicate-based gadgets under RISC-V.

Exploiting taxonomy-overlooked µop expansion under STT (Limitation I & II). To demonstrate that selective-speculation defenses can miss RS-contention channels beyond their transmit taxonomy, we evaluate Gadget 1 (Section 4.1) under STT, which delays both arithmetic and memory-access µops. This experiment targets two assumptions simultaneously. First, existing transmit taxonomies emphasize post-issue execution effects and overlook operand-dependent µop expansion during decode and dispatch. Second, delay-until-resolution mechanisms prevent control-predicate leakage mainly by suppressing secret-dependent speculative fetch redirects, but do not constrain predicated data selection.

Concretely, we use predicated instructions to encode a secret bit into the iteration count of `REP MOVSB`, choosing different RCX values for bit 0 and bit 1, thereby selecting lower- and higher-µop expansion paths, respectively. We place a target instruction later on the mis-speculated path: a `call` through a function pointer `dummy` whose entry point has been evicted from the I-cache. Whether `dummy`'s entry point is fetched into the I-cache during the transient window depends on the

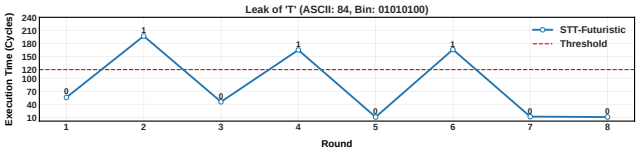

Figure 9: Secret bits recovery for the predication-based gadget exploiting RS Contention induced by `REP MOVSB` instruction under STT.

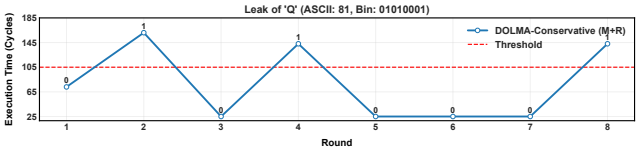

Figure 10: Secret bits recovery for the gadget exploiting RS contention induced by variable-time memory access instructions under DOLMA.

RS pressure created by `REP MOVSB`, and is later reflected in the timing of a subsequent call to `dummy`. As shown in Figure 9, this produces measurable timing differences even under STT's strongest Futuristic threat model, enabling recovery of an 8-bit ASCII character in eight rounds.

Bypassing older-µop-first with taxonomy-recognized transmit µops under DOLMA (Limitation III). DOLMA reduces overhead by leaving arithmetic and cache-hit loads undelayed and relies on prioritizing older µops to avoid explicit backend contention; however, this optimization still leaves Gadget 3 (Section 4.2) exploitable. We instantiate Gadget 3 in gem5 under DOLMA Conservative (M+R) to show that older speculative µops can still create measurable RS contention under older-µop-first allocation. Specifically, we keep `array2[0 * L1_BLOCK_SZ_BYTES]` cached and flush `array2[40 * L1_BLOCK_SZ_BYTES]`. A secret-dependent index computation, e.g., `addr = secret * 40 * L1_BLOCK_SZ_BYTES`, selects between the hit and miss locations, producing secret-dependent load readiness. A dependent arithmetic chain amplifies this readiness difference, prolonging RS occupancy when the miss is selected. As in Gadget 1, we use a function-call-based I-cache probe to test whether a subsequent call target is fetched into the I-cache during the transient window. The results in Figure 10 confirm that secret-dependent RS pressure remains observable even under DOLMA's strongest threat model, Conservative (M+R).

Implementation-dependent effectiveness of delay-until-resolution. Semantically equivalent value-selection code

```
01 if (condition){
02  idx = secret_bit?40:4;
03 *t = &array2[idx*L1_BLOCK_SZ_BYTES];
04  asm volatile(
05  "MOV (%0),%%r15\n"
06  ".rept 120\n"
07 "ADD %%r15,%%r15\n"
08  ".endr\n"
09 "CALL dummy\n"   // target instruction
10  :"+r"(t)
11  :
12  :"%r13","%r15","memory"
13  );}
```

Figure 11: Gadget variants exploiting RS contention induced by a variable-latency memory access: the secret controls a selector that chooses between hit and miss indices.

(a) Secret dependent predication under X86

```
MOVL %[secret_bit],%%ebx
// mov hit address offset to %rdx
MOV $0x40,%%rdx
// mov miss address offset to %rcx
MOV $0xA00,%%rcx
MOV $0,%%r10d
CMP %%ebx,%%r10d
// if secret_bit = 0, overwrite %rcx with 0x40
CMOVE %%rdx,%%rcx
// add rcx to base address
ADD %[base_address],%%rcx
```

(b) Secret dependent predication under RISC-V

```
LW t2,0(%[secret_bit])
LI t3,0x40
LI t4,0xA00,%%rcx
// secret_bit==0 -> t5=t3; else t5=0
czero.eqz t5, t3, t2
// secret_bit !=0-> t4=t4; else t4=0
czero.nez t4, t4, t2
OR t4, t5, t4
ADD t5,t4,%[base_address]
```

Figure 12: Assembly-level predicated implementations of the gray-shaded snippet in Figure 11: (a) x86 and (b) RISC-V.

(e.g., if-else or ternary expressions) can be realized either as branches or predicated moves, depending on the compiler backend and optimization level. As illustrated in Figure 11, the gadget encodes a secret by selecting either `hit_index` or `miss_index`, so that a subsequent load becomes a cache hit or miss and introduces distinct operand-readiness delays in the leakage sequence. We then evaluate two semantically equivalent realizations of this selector: a branch-based selector and a predicated selector. On x86, the predicated selector is implemented using `CMOV`-class instructions (e.g., `CMOVE`); on RISC-V, the predicated selector is implemented using Zicond instructions (`czero.eqz`/`czero.nez`). Figure 12 (a), (b) shows the corresponding x86 and RISC-V realizations. The results (Figure 13) confirm that delay-until-resolution is effective for the branch-based selector: by suppressing secret-dependent speculative redirection, it effectively fixes transient execution on a single path and keeps timings below the decision threshold. In contrast, the predicated selector performs secret-dependent data selection without any fetch redirect, thereby bypassing delay-until-resolution and producing secret-dependent timing differences. This divergence suggests that source-level code that appears safe under selective speculation can compile into gadgets that bypass delay-until-resolution.

Contention thresholds of REP-prefixed string instructions on Raptor Cove. We evaluate whether operand-dependent µop expansion in REP-prefixed string instructions can induce RS pressure to delay the fetch of a subsequent target instruction on real Intel processors. We implement Gadget 1 from Section 4.1 on Intel Raptor Cove P-cores (Core i9-13900K and Core i7-14650HX).

Figure 14 shows the measured post-transient execution

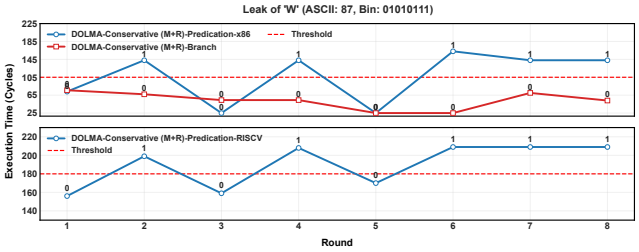

Figure 13: Secret bits recovery for the gadget exploiting RS contention via secret-dependent predication under DOLMA across different ISAs: x86 (top) and RISC-V (bottom).

time of `dummy()` (20 measurements per point) as we sweep `RCX`; the sampled `RCX` values correspond to different issued-µop levels. For `REP MOVSB`, the contention signal becomes stable at `RCX`=3008, corresponding to about 174 issued µops, where the subsequent `CALL dummy` exhibits a clear timing increase. For comparison, we also tested `REP STOSB` on the same processors and observed a lower stable threshold: clear contention appears at `RCX`=2432, corresponding to 98 issued µops. The detailed correspondence between `RCX` values and the measured µop counts for `REP MOVSB` and `REP STOSB` can be found in Table 2 and Appendix A, Table 6, respectively.

As shown in Figure 15, our PoC recovers a secret character on the Core i9-13900K. Over 2,000 measurements per bit value, a single-threshold probe achieves 85.10% (bit 0) and 85.11% (bit 1) classification accuracy (Figure 16). These results confirm that `REP`-prefixed instructions can induce operand-dependent RS pressure and a robust instruction-fetch/I-cache side effect on real CPUs, supporting the realism of the primitive assumed in our simulated defense-bypass evaluations.

We emphasize that these thresholds are not fixed architectural constants. In noisier conditions, we also observe contention effects at smaller `RCX` values (e.g., when the issued-µop count reaches roughly 80). The difference between `REP MOVSB` and `REP STOSB` further suggests that the threshold is jointly determined by the total µop pressure, the distribution of dispatched µops across backend ports, and the organization of backend resources, rather than by the issued-µop count alone. However, the thresholds reported here correspond to relatively clean operating conditions and therefore serve as conservative operating points.

6 Defense

As discussed in Section 3, schemes such as NDA and SpecShield avoid the optimized defense strategies that we identify as insecure. However, they adopt a coarse-grained policy that blocks speculative data propagation wholesale, effectively disabling ILP during speculation and incurring prohibitive overhead. We align with the design philosophy of state-of-

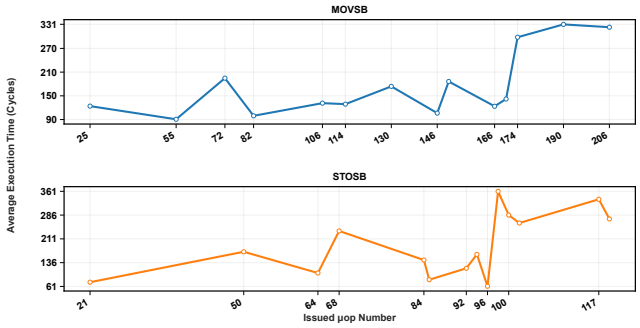

Figure 14: Execution time of target instruction vs. the corresponding issued-µop count, obtained by sweeping `RCX`, for `REP MOVSB` and `REP STOSB` on Intel Core i9-13900K.

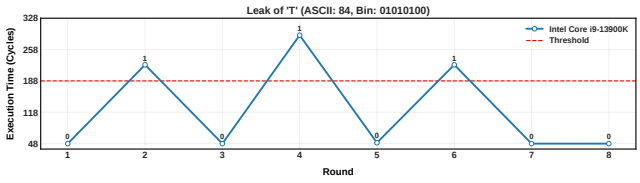

Figure 15: Secret bits recovery for a gadget exploiting the `REP MOVSB` instruction on Intel Core i9-13900K.

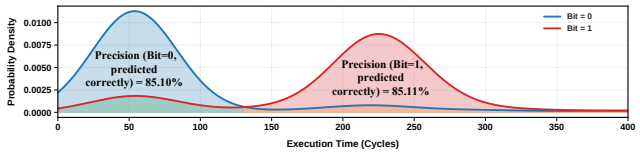

Figure 16: Execution time distributions (via kernel density estimates) of the target instruction for secret bits 0 and 1 on Intel Core i9-13900K.

the-art selective speculation defenses, which permit limited speculative data propagation while blocking only security-critical cases. Our goal is to strengthen this line of defenses by addressing the security limitations revealed in this paper.

Addressing Limitation I: Refining the transmit taxonomy beyond post-issue effects. Existing schemes typically define transmit instructions by execution-phase effects, implicitly assuming that (i) transmission must be caused by effects after issue, and (ii) observable timing variation must originate from the transmit µop's execution. Our results contradict both assumptions: secrets can influence microarchitectural state via decode and dispatch behavior, e.g., operand-dependent µop expansion, without requiring per-µop variable-time execution. We therefore redefine transmit behavior to capture pre-execution channels:

A speculative instruction is a transmit instruction if operand-dependent variation in its speculative behavior can induce attacker-observable microarchitectural state changes.

Under this definition, microcode-expanded instructions, such as `REP`-prefixed string instructions, become transmitters when their operand-dependent expansion alters RS occupancy and subsequently modulates I-cache state. A conservative defense should therefore cover such pre-execution transmit behavior. Moreover, for channels that manifest at decode and dispatch, delaying the issue alone is insufficient: the defense must prevent the speculative instruction from entering the backend scheduling structure in the first place.

This extension is necessarily ISA-specific, because it depends on identifying instructions whose speculative behavior exhibits operand-dependent microcode expansion under the target ISA and microcode model. In our implementation, we perform a conservative survey of gem5's x86 ISA implementation and identify instructions matching this pattern, including `REP/REPE/REPNE` string and string-I/O instructions as well as `ENTER`. We then prevent speculative instances of these instructions with tainted expansion-controlling operands from entering the backend scheduling structure.

Addressing Limitation II: Predicated selection under delay-until-resolution. Delay-until-resolution defers fetch redirects for tainted branches, assuming that redirect-based path divergence is sufficient to capture control-dependent leakage. Predicated selection breaks this assumption by enabling secret-dependent control semantics such as conditional updates and value selection, without any branch resolution or fetch redirection.

In the pattern studied in this paper, predication is not itself the primary source of contention; rather, it serves as a secret-injection mechanism that encodes the secret into operands later consumed by contention-inducing primitives, such as the repeat count of a `REP`-prefixed instruction. Thus, delaying tainted predicated instructions blocks this secret-injection step and stops the concrete gadget pattern demonstrated in this paper. While the direct Limitation-I treatment already blocks the identified operand-dependent microcode-expanding transmitters, predicate hardening is complementary because it limits downstream exposure of secret-dependent values to operand-sensitive primitives that may not yet be characterized by existing selective-speculation defenses.

Addressing Limitation III: Bidirectional contention defeats older-µop-first allocation. The motivation behind this strategy is to further narrow the set of delayed µops. However, our results show that unprotected transmit µops, specifically variable-time arithmetic instructions and cache-hit memory µops, can still be exploited to construct RS-contention channels even under older-µop-first allocation. When contention becomes bidirectional between older and younger µops, simply prioritizing either side cannot prevent leakage. Thus, the only principled solution is to continue delaying tainted-data-dependent arithmetic and memory-access µops, as implemented by STT and SPT.

In summary, we extend the STT framework in gem5's x86 O3 model and evaluate performance overhead using the SPEC2017 benchmark suite. First, although STT already de-

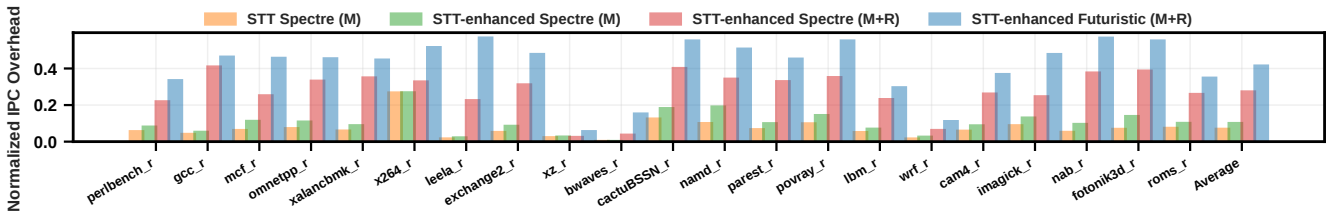

Figure 17: Performance overheads of STT and STT-enhanced on SPEC2017 in gem5 simulator.

lays tainted-data-dependent arithmetic and load µops, it overlooks store-based TLB side channels which exploit speculative stores potentially triggering TLB misses. Since stores do not generate any memory-side effects during speculative execution beyond TLB lookups, they can affect occupancy state only through TLB behavior. Following DOLMA, we additionally delay tainted-data-dependent stores that may incur TLB misses, classifying them as transmit µops. Second, to directly address Limitation I, we identify x86 instructions with operand-dependent microcode expansion and prevent speculative instances with tainted expansion-controlling operands from being dispatched into the instruction queue in gem5's x86 O3 model. This provides direct coverage for the identified pre-execution transmitters under our refined transmit taxonomy. Third, to address Limitation II, we extend STT to delay the use of tainted-data-dependent predicate sources, such as condition flags, as operands in non-control µops. This blocks the secret-injection step and more broadly reduces exposure to downstream primitives whose operand-sensitive behavior may go beyond the specific gadgets characterized in this work.

As shown in Figure 17, the original STT incurs a performance overhead of 6.86%, while our enhanced STT increases to 10.01% under the Spectre model. We further extend STT to mitigate register-based leakage channels, denoted as STT-enhanced (M+R). Unlike SPT, which conservatively treats all destination registers of completed instructions as potential taint sources, STT-enhanced (M+R) marks only the destination register of the first speculative µop as tainted, and propagation follows the same rules as STT-enhanced (M). Like NDA's strict data propagation policy, this defense cannot prevent attacks that leak secrets through a single speculative µop, a scenario that is unlikely to occur in practical systems [46]. Under this configuration, the improved mitigation incurs an average normalized IPC overhead of 27.29% under the Spectre model and 41.44% under the Futuristic model.

7 Discussion

Systematic discovery of µop-expansion patterns. The REP-prefixed string instruction case is not an isolated hand-crafted example, but a representative instance of a broader class of operand-sensitive µop-expansion behaviors. More generally, similar patterns can be identified through a systematic workflow: selecting candidate complex or microcoded instruction families, sweeping their key controlling operands, profiling operand-dependent µop counts with PMU events, and then constructing transient test cases to determine whether a stable contention threshold can be reached.

Our follow-up exploration further suggests that operand-sensitive µop-expansion behavior is not unique to REP-prefixed string instructions. We also observe such behavior in other microcode-assisted instruction families, including complex x87 floating-point instructions for transcendental functions such as `FSIN`, `FCOS`, `FPTAN`, `FSINCOS`, and `FPATAN`. However, these cases fall outside the main scope of bypassing selective-speculation defenses considered in this paper: in PMU measurements, they are accompanied by nonzero recovery-related indicators, suggesting the presence of internal micro-branches. More broadly, operand-dependent variation in µop counts alone is insufficient to imply the same attack surface. Whether such a pattern becomes exploitable depends on at least three additional factors: the strength of the resulting backend pressure, the residence time of the corresponding µops in the relevant backend structures, and the availability of a probe that can reliably observe the induced contention on the target microarchitecture. These factors explain why different instructions can exhibit different effective thresholds and different probe sensitivity, even when they all show operand-sensitive internal behavior. In this sense, REP-prefixed string instructions are best viewed as representative gadgets rather than the only possible ones.

Generalization of µop-expansion-induced contention to AMD. The same workflow can also be applied to AMD Zen-family processors. However, the way the resulting pressure becomes observable is strongly microarchitecture-dependent. Prior work, such as SQUIP [15] and its extension [14], studies scheduler-queue contention on Zen under a distributed, type-specific scheduler organization, where different instruction classes map to different scheduler queues, and observability often depends on the probe stressing the same scheduler queue as the leakage sequence. Their focus is on scheduler contention itself, especially in SMT settings and queue-specific priming/probing strategies.

Our experiments on AMD Zen 1 and Zen 3 are consistent with this architectural picture. Both `REP MOVS`- and `REP`

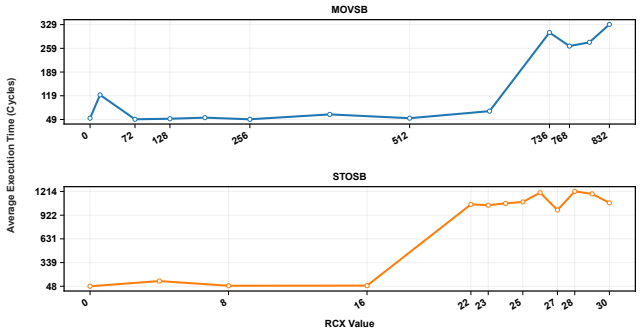

Figure 18: Execution time of the memory-access-based probe across different `RCX` values for `REP MOVSB` and `REP STOSB` on AMD Ryzen 7 5800H.

`STOS`-based constructions produce measurable scheduler contention effects within a single thread, but the probe design that works on Intel does not transfer directly. In particular, a call-based downstream probe is much less effective on AMD, whereas a memory-access-based probe yields a substantially clearer signal because it is better aligned with the scheduler queue stressed by the `REP`-prefixed string instructions. Figure 18 shows the threshold plots measured on Zen 3, where we sweep `RCX` and measure the post-transient execution time of the target memory access (20 measurements per point). For `REP STOSB`, contention becomes visible once `RCX` exceeds 22, at which point the dispatched load and store µop counts are approximately 2 and 22, respectively. The signal therefore aligns with the store-side scheduler pressure: the roughly 22 dispatched store µops correspond to an effective occupancy of about 22 entries in the relevant scheduler queue, matching the capacity reverse-engineered in prior work [14]. `REP MOVSB` produces a signal at substantially larger operand ranges (around `RCX`=736 in our setup), where the dispatched load and store µop counts reach approximately 31 and 32. This suggests a broader and less directly probe-aligned scheduler footprint than in the more store-concentrated `REP STOSB` case. Overall, what generalizes across architectures is the methodology for identifying operand-sensitive patterns, whereas the effective probe, threshold, and visibility of the signal remain architecture-specific. Supporting performance-counter measurements across different `RCX` values are provided in Appendix B (Table 7).

Distinction from AMD Transient Scheduler Attacks (TSA). This distinction is also important for separating our mechanism from AMD's documented Transient Scheduler Attacks (TSA) and their corresponding mitigations [2]. TSA is described as arising from false completion of loads, where invalid data from the L1 cache path or store queue affects the timing of dependent instructions; AMD further distinguishes TSA-L1 and TSA-SQ as load-related variants. By contrast, our mechanism does not rely on false completion or invalid forwarded load data. Instead, it arises from operand-dependent internal instruction behavior that creates secret-dependent backend pressure before the transient window closes. Thus, while both lines of work concern timing effects in backend execution, TSA and the mechanism studied here stem from different microarchitectural causes and should not be conflated.

8 Conclusion

Selective-speculation defenses promise comprehensive mitigation of transient-execution attacks with low overhead by delaying only a subset of transmit instructions. In this paper, we show that widely adopted assumptions behind these optimized designs leave systematic blind spots. We identify a broader RS-contention leakage mechanism in which secret-dependent RS pressure throttles frontend fetch during mis-speculation and creates measurable post-recovery I-cache footprint differences. Through a systematic analysis, we pinpoint three root causes: taxonomy blind spots for decode-phase µop expansion, inadequate coverage of predicated selection under delay-until-resolution, and the failure of older-µop-first allocation under RS-driven fetch throttling. Guided by these insights, we construct Spectre-v1-style gadgets that bypass state-of-the-art defenses in simulation and demonstrate the signal on real CPUs, and we propose strengthened STT mechanisms that close both prior and newly exposed gaps with moderate overhead.

Ethical Considerations

Disclosure Statement. We responsibly disclosed the newly identified Spectre-class gadget patterns presented in this paper to Intel on July 14, 2025, and to AMD on May 18, 2026. Intel acknowledged the observed behavior and indicated that it falls under existing software-level mitigation guidance, such as constant-time programming [20]. AMD acknowledged our report on May 29, 2026, and informed us that it would publish an AMD security brief to coincide with the conference date. AMD indicated that the brief would not assign a CVE and would contain existing guidance previously provided in response to similar reports. We did not identify evidence that the academic defenses evaluated in this paper, including STT and DOLMA in their published forms, have been deployed as commercial mitigations or incorporated into vendor products. For that reason, apart from vendors on whose hardware we directly validated related behavior, we did not treat other unspecified vendors as subjects of coordinated vulnerability disclosure. Our study focuses on a Spectre-class attack mechanism and on published research defenses rather than on an undisclosed product-specific flaw in a deployed commercial mitigation.

Affected Stakeholders. The stakeholders potentially affected by this work include CPU vendors whose processors may exhibit related speculative-execution behaviors; developers of security-sensitive software, especially cryptographic and constant-time code; operators and users of affected systems, particularly in shared environments; designers and adopters of speculative-execution defenses; and the broader security research community. These groups may be affected in different ways: vendors and defense designers may need to reassess assumptions in mitigation designs, software developers may need to revisit constant-time engineering practices, and system operators may need to better understand residual risk even when selective-speculation defenses are assumed to be in place. During the research process, these stakeholders were not directly exposed to harm because experiments were restricted to private systems and released artifacts. Upon publication, however, vendors and defense designers may need to reassess mitigation assumptions, software developers may need to revisit constant-time engineering practices, and system operators may need to reconsider residual deployment risk.

Potential Harms and Mitigations. The main ethical risk is that publishing concrete gadget patterns and bypass mechanisms could help attackers reason about overlooked transient-execution channels. We mitigated this risk in several ways. All real-hardware experiments were performed only on private, isolated systems owned and managed by our research group. We did not test third-party systems, production environments, public infrastructure, or shared cloud resources. Defense-bypass evaluations were conducted only in open-source simulation environments or on released research artifacts, not on deployed vendor mitigations. We did not access unreleased hardware information or proprietary defenses. At the same time, we conducted this research because selective-speculation defenses are increasingly proposed as practical mitigations, and blind spots in their assumptions have direct implications for system security. Importantly, this paper does not merely demonstrate bypass paths; it also identifies the underlying design limitations and proposes corresponding defense extensions to close the exposed gaps. In our judgment, the defensive benefit of identifying these blind spots and proposing corresponding mitigations outweighs the incremental publication risk, because the results can inform improved mitigation design, threat modeling, and constant-time engineering practice.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 625B2047 and No. 62472088), and in part by the Big Data Computing Center of Southeast University. The authors thank the anonymous reviewers and shepherd for their comments and suggestions that help us improve the quality of the paper. Xiaoyu Cheng conducted this research partly during her visit to the National University of Singapore, hosted by Professor Trevor E. Carlson.

Open Science

Artifact provided. We are making the artifact of our study publicly available to the research community at https://doi.org/10.5281/zenodo.20364721. It contains: (i) the nanoBench-based measurement code and scripts used in Section 4.1 to characterize `REP MOVSB` (including all code needed to reproduce Table 2); (ii) the LLVM-based analysis pass used in Section 4.3 to identify RS-contention-relevant patterns in real-world code (supporting the results in Table 4), together with a PoC timing channel instantiated from a libsodium function identified by this pass and its evaluation scripts; (iii) the STT/DOLMA artifact-based simulation code used in Section 5, including the required simulator-side extensions for our evaluations, such as the branch-less `REP MOVSB` implementation and RISC-V Zicond `czero`-class instruction support; (iv) PoC gadgets and experiment scripts to reproduce the simulation results and figures in Section 5; and (v) the gem5 implementation of our strengthened STT mechanisms from Section 6.

References

[1] Andreas Abel and Jan Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. In *International Symposium on Performance Analysis of Systems and Software*, pages 34–46, 2020.

[2] Advanced Micro Devices, Inc. Technical Guidance for Mitigating Transient Scheduler Attacks, July 2025. AMD whitepaper on TSA-L1/TSA-SQ and recommended mitigations.

[3] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *International Symposium on Computer Architecture*, pages 132–144, 2020.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *Symposium on Security and Privacy*, pages 870–887, 2019.

[5] Arm Ltd. *Arm Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE)*. 2022. Defines the predication and scalable vector mechanisms of SVE. URL: https://developer.arm.com/documentation/ddi0584/latest.

[6] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 151–164, 2019.

[7] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1046–1060, 2021.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.

[9] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security Symposium*, pages 1967–1984, 2020.

[10] Xiaoyu Cheng, Fei Tong, Hongyu Wang, Zhe Zhou, Fang Jiang, and Yuxing Mao. SpecLFB: Eliminating cache side channels in speculative executions. In *USENIX Security Symposium*, pages 631–646, 2024.

[11] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *International Symposium on Microarchitecture*, pages 607–622, 2021.

[12] Yujie Cui, Hongwei Cui, and Xu Cheng. Information leakage attacks exploiting cache replacement in commercial processors. *Transactions on Computers*, 72(9):2536–2547, 2023.

[13] Jacob Fustos, Michael Bechtel, and Heechul Yun. Spectrerewind: Leaking secrets to past instructions. In *Workshop on Attacks and Solutions in Hardware Security*, pages 117–126, 2020.

[14] Stefan Gast, Jonas Juffinger, Lukas Maar, Christoph Royer, Andreas Kogler, and Daniel Gruss. Remote scheduler contention attacks. In *International Conference on Financial Cryptography and Data Security*, pages 365–383. Springer, 2024.

[15] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. Squip: Exploiting the scheduler queue contention side channel. In *2023 Symposium on Security and Privacy*, pages 2256–2272, 2023.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, 2016.

[17] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial prefetch: New cross-core cache side channel attacks. In *Symposium on Security and Privacy*, pages 1458–1473, 2022.

[18] Ali Hajiabadi and Trevor E. Carlson. Conjuring: Leaking control flow via speculative fetch attacks. In *Design Automation Conference*, pages 1–6, 2024.

[19] Jann Horn. *Speculative execution, variant 4: Speculative store bypass, 2018*. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[20] Intel Corporation. *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. URL: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

[22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[23] Hai Jin, Zhuo He, and Weizhong Qiang. SpecTerminator: Blocking speculative side channels based on instruction classes on RISC-V. *Transactions on Architecture and Code Optimization*, 20(1):1–26, 2023.

[24] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy*, pages 1–19, 2019.

[26] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies*, 2018.

[27] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor. In *USENIX Security Symposium*, pages 2137–2154, 2024.

[28] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *International Symposium on High-Performance Computer Architecture*, pages 264–276, 2019.

[29] 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. In *USENIX Security Symposium*, pages 973–990, 2018.

[30] Chang Liu, Yongqiang Lyu, Haixia Wang, Pengfei Qiu, Dapeng Ju, Gang Qu, and Dongsheng Wang. Leaky MDU: ARM memory disambiguation unit uncovered and vulnerabilities exposed. In *Design Automation Conference*, pages 1–6, 2023.

[31] Chang Liu, Dongsheng Wang, Yongqiang Lyu, Pengfei Qiu, Yu Jin, Zhuoyuan Lu, Yinqian Zhang, and Gang Qu. Uncovering and exploiting AMD speculative memory access predictors for fun and profit. In *International Symposium on High-Performance Computer Architecture*, pages 31–45, 2024.

[32] LLVM. *LLVM's Analysis and Transform Passes*. URL: https://llvm.org/docs/Passes.html.

[33] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *USENIX Security Symposium*, pages 1397–1414, 2021.

[34] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[35] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *USENIX Workshop on Offensive Technologies*, 2019.

[36] Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. Analyzing and Exploiting Branch Mispredictions in Microcode. *arXiv preprint arXiv:2501.12890*, 2025.

[37] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium*, pages 1451–1468, 2021.

[38] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Symposium on Security and Privacy*, pages 1852–1867, 2021.

[39] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299, 2019.

[40] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative Dereferencing: Reviving foreshadow. In *Financial Cryptography and Data Security*, pages 311–330, 2021.

[41] Philipp Tomsich. *The RISC-V Integer Conditional (Zicond) Operations Extension, Version 1.0.1*. 2023. Ratified specification. URL: https://riscv.org/spec-state.

[42] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[43] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient Out-of-Order execution. In *USENIX Security Symposium*, pages 991–1008, 2018.

[44] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *Symposium on Security and Privacy*, pages 54–72, 2020.

[45] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Symposium on Security and Privacy*, pages 88–105, 2019.

[46] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *International Symposium on Microarchitecture*, pages 572–586, 2019.

[47] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *International Symposium on Microarchitecture*, pages 428–441, 2018.

[48] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural modeling for security evaluation. In *International Symposium on Computer Architecture*, pages 1–15, 2023.

[49] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[50] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *International Symposium on Microarchitecture*, pages 954–968, 2019.

[51] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *USENIX Security Symposium*, pages 7125–7142, 2023.

A Additional Intel Measurement Results

Additional nanoBench measurements for `REP STOSB` on Intel Raptor Cove are shown in Table 6.

Table 6: Performance-counter measurements for `REP STOSB` across different `RCX` values on Intel Core i9-13900K.

RCX	IDQ. MITE_UOPS[1]	IDQ. MS_UOPS[2]	UOPS. _ISSUED	UOPS_RETIRE .SLOTS	RECOVERY_ CYCLES[3]	MACHINE_ CLEARS[4]
`LEA RDI, dst_addr; MOV AL, 0x41; MOV RCX, test_value;REP STOSB"						
0	6.99	14.00	21.00	21.00	0.00	0.00
64	7.00	14.00	21.00	21.00	0.00	0.00
128	7.01	14.00	21.00	21.00	0.00	0.00
256	7.00	43.00	50.00	50.00	0.00	0.00
320	7.00	57.00	64.00	64.00	0.00	0.00
512	7.00	61.00	68.00	68.00	0.00	0.00
832	7.00	78.00	85.00	85.00	0.00	0.00
1536	6.99	77.00	84.00	84.00	0.00	0.00
2176	7.00	87.00	94.00	94.00	0.00	0.00
2432	7.00	90.99	98.00	98.00	0.00	0.00
2560	7.00	93.00	100.00	100.00	0.00	0.00
3008	6.99	112.00	119.00	119.00	0.00	0.00

B Supplementary Experimental Details for AMD Zen

We constructed a nanoBench performance-counter configuration for the AMD Ryzen 7 5800H platform. Table 7 reports the resulting measurements for `REP STOSB` and `REP MOVSB` across different `RCX` values. These measurements provide the detailed sweep data underlying the discussion in Section 7.

Table 7: Performance counter measurements conducted on the AMD Ryzen 7 5800H.

RCX	EX_RET _INSTR[1]	EX_RET _OPS[2]	LS_DISPATCH _LD[3]	LS_DISPATCH _STORE[4]	EX_RET _BRN_MISP[5]
`LEA RDI, dst_addr; MOV AL, 0x41; MOV RCX, test_value;REP STOSB"					
0	4.00	6.00	2.00	0.00	0.00
4	4.00	46.00	2.00	4.00	0.00
8	4.00	58.00	2.00	8.00	0.00
16	4.00	82.00	2.00	16.00	0.00
17	4.00	85.00	2.00	17.00	0.00
18	4.00	88.00	2.00	18.00	0.00
19	4.00	91.00	2.00	19.00	0.00
20	4.00	94.00	2.00	20.00	0.00
21	4.00	97.00	2.00	21.00	0.00
22	4.00	100.00	2.00	22.00	0.00
24	4.00	106.00	2.00	24.00	0.00
32	4.00	130.00	2.00	32.00	0.00
64	4.00	226.00	2.00	64.00	0.00
"LEA RSI, src_addr; LEA RDI, dst_addr; MOV RCX, test_value; REP MOVSB"					
0	4.00	6.00	3.00	1.00	0.00
16	4.00	75.00	11.00	11.00	0.00
72	4.00	75.00	11.00	11.00	0.00
128	4.00	102.00	15.00	16.00	0.00
184	4.00	150.00	24.00	25.00	0.00
256	4.00	114.00	19.00	20.00	0.00
384	4.00	126.00	23.00	24.00	0.00
512	4.00	138.00	27.00	28.00	0.00
640	4.00	150.00	31.00	32.00	0.00
736	4.00	150.00	31.00	32.00	0.00
800	4.00	162.00	35.00	36.00	0.00
832	4.00	162.00	35.00	36.00	0.00

[1] EX_RET_INSTR: Number of retired architectural instructions.
[2] EX_RET_OPS: Number of retired macro-ops.
[3] LS_DISPATCH_LD: Number of load operations dispatched to the load/store unit.
[4] LS_DISPATCH_STORE: Number of store operations dispatched to the load/store unit.
[5] EX_RET_BRN_MISP: Number of retired branch instructions that were mispredicted.