ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage

Sungkeun Kim†, Farabi Mahmud†, Jiayi Huang†, Pritam Majumder†, Neophytos Christou§
Abdullah Muzahid†, Chia-Che Tsai† and Eun Jung Kim†
†Texas A&M University, §University of Cyprus
{ksungkeun84,farabi,jyhuang,pritam2309,abdullah.muzahid,chiache,ejkim}@tamu.edu, nio_christou@hotmail.com

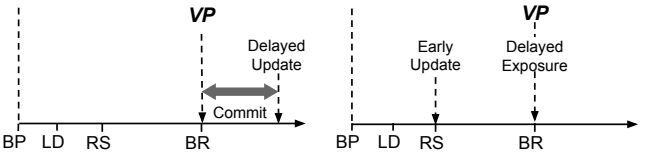
***Abstract*—Spectre and Meltdown attacks reveal the perils of speculative execution, a prevalent technique used in modern processors. This paper proposes ReViCe, a hardware technique to mitigate speculation based attacks. ReViCe allows speculative loads to update caches early but keeps any replaced line in the victim cache. In case of misspeculation, replaced lines from the victim cache are used to restore the caches, thereby preventing any cache-based Spectre and Meltdown attacks. Moreover, ReViCe injects jitter to conceal any timing difference due to speculative lines. Together speculation restoration and jitter injection allow ReViCe to make speculative execution secure. We present the design of ReViCe following a set of security principles and evaluate its security based on shared-core and cross-core attacks exploiting various Spectre variants and cache side channels. Our scheme incurs 2-6% performance overhead. This is less than the state-of-the-art hardware approaches. Moreover, ReViCe achieves these results with minimal area and energy overhead (0.06% and 0.02% respectively).**

***Keywords*—Transient-Execution Attacks, Cache Side-channel Attacks, Hardware Mitigation**

I. Introduction

Speculative execution [1]–[10] is integral to the performance optimization of modern out-of-order CPUs. A CPU can increase utilization by speculating uncertainties such as branching, memory dependency or exception handling. Recent Spectre [11] and Meltdown [12] vulnerabilities show the danger of combining side channels with speculative execution. Spectre attacks exploit speculative execution that violates the program logic whereas Meltdown attacks bypass hardware protections. Attackers can manipulate the CPU to bypass security checks or jump to malicious gadgets in order to trigger *transient execution*, a transient period before squashing the execution, that leaks secrets via a side channel. Even more devastating, various speculation can be combined with different side channels, multiplying the variants of Spectre and Meltdown [13]–[20].

Mitigation of Spectre and Meltdown is challenging. Existing software or firmware-level solutions are ad-hoc and expensive [21]–[28]. Several recent papers proposed more systematic hardware-based solutions. For example, InvisiSpec [29] is the first work to offer a solution by isolating cache updates from transient execution. Follow-up works focus on reducing the latency and space overhead of recovering from squashed updates [30]–[32]. Other solutions are based on either disabling speculation for leaky instructions or data access [33]–[37] or blocking side channels between security contexts [38], [39]. Existing solutions [33]–[39] either incur high performance overhead, require software modification, or are unable to prevent attacks within the same context or core. As a more efficient approach, InvisiSpec and its follow-up works are based on either **Redo**'ing or **Undo**'ing the side effects of transient execution in a microarchitectural structure (e.g., caches). Both approaches need to ensure that microarchitecture updates are only exposed to outside of the transient execution at **Visibility Point (VP)** [29], i.e., when the instructions cannot be squashed by prior operations. The Redo approach, as in InvisiSpec [29] and Selective Delay [31] propose to delay the update of caches after the VP. On the other hand, the Undo approach, as in CleanupSpec [30], applies the updates early but undo them at VP.



(a) Redo (InvisiSpec, Select. Delay) (b) Undo (CleanupSpec, ReViCe)

Figure 1: After branch prediction (BP), the load (LD) and response (RS) can occur before branch resolution (BR). Both Redo and Undo approaches assume the cache update is safe at the visibility point (VP), which is BR, but Redo'ing delays the actual update beyond the VP.

The distinction between Redo and Undo causes a significant difference in performance. Redo suffers higher overheads than Undo as a Redo'ing architecture such as InvisiSpec penalizes execution that is confirmed to be *safe*. The distinction is demonstrated in Figure 1. Penalizing confirmed execution is counter-intuitive since CPUs are generally optimized for executions where predictions are mostly correct and exceptions are rare. For instance, in PARSEC, at least 82% of branches are predicted correctly, and ∼80% of the load instructions become safe without getting squashed. Both InvisiSpec and Selective Delay procrastinate the update until the VP; however, before exposing a cache line after the VP, the CPU needs to reacquire the data from the cache hierarchy to check for potential invalidation from other cores. The delayed exposure may stall subsequent instructions that depend on the cache line. InvisiSpec also increases network traffic in the cache hierarchy for reacquiring data. InvisiSpec incurs 13% and 16% slowdown to PARSEC [40] for mitigating Spectre and Meltdown, respectively.

The Undo approach, as in CleanupSpec [30], incurs a lower performance overhead as compared to all other solu-

tions, and thus, can be appealing to CPU vendors. However, the design of the CleanupSpec has two major security flaws. **First**, it can leak secrets to shared-core and cross-core attackers. CleanupSpec cannot hide eviction of an L1 cache line since the data is evicted for real. This allows a shared-core attacker to launch an eviction-based attack such as Prime+Probe. Moreover, a cross-core attacker can observe a speculative load in CleanupSpec using invalidation. **Second**, attacks to cache address randomization of CleanupSpec are possible. CleanupSpec adopts CEASER's [41] scheme to randomize a mapping between address and cache sets. Bodduna, et al. [42] shows that the block cipher of CEASER does not randomize address mappings properly. Randomization does not prevent an attacker from detecting the access of a shared cache block by the victim. The security flaws in CleanupSpec demonstrated the complexity of implementing the Undo'ing approach. In this paper we focus on re-examining the principles for Undo'ing scheme to ensure that a low-overhead, secure Undo'ing scheme can be implemented in architecture.

This paper presents **ReViCe**, a microarchitecture that *safely* hides and restores speculative cache updates in both shared-core and cross-core scenarios. When an instruction queue issues a load, ReViCe determines if the load is speculative. If the load is speculative and causes a cache miss, a new line is brought in to replace an existing line, changing the cache state. The change is safe if the speculation is later deemed correct. However, if the speculation is incorrect, an attacker who can observe the change, can infer secrets from access patterns. To prevent such leakage, the cache needs to restore any change caused by a misspeculated load. In ReViCe, we use a *victim cache* [43] to keep the lines replaced by speculative loads. If a load is deemed misspeculated, ReViCe can restore the replaced line from the victim cache to its original location. Moreover, if multiple accesses occur to the cache line while the line is speculative, ReViCe adds *jitters* to prevent attackers from observing any timing difference. With restoration and jitter, ReViCe hides speculative cache changes to the attackers. To make ReViCe free from security flaws, we propose four security principles.

We simulate ReViCe on Gem5 [44], for both x86 and ARM. We show that with a relatively small victim cache, ReViCe can mitigate Spectre and Meltdown with low performance overheads. We use benchmark PARSEC in ReViCe, InvisiSpec, Selective Delay, and CleanupSpec to show the benefits of our design. We also performed a security analysis on the defenses of ReViCe in various corner cases. We attempted attacks on ReViCe with 4 Spectre variants, 3 cache side channels [45]–[48], and 3 attack scenarios.

Our contributions are summarized as follows:

- A multi-core, microarchitecture design for mitigating Spectre and Meltdown attacks by Undo'ing sensitive cache updates from misspeculated execution. ReViCe proposes a novel reuse of victim cache for hiding and restoring cache states.
- Security principles for designing an Undo'ing microarchitecture with secure speculation and security analysis based on shared-core and cross-core attacks.
- A performance analysis of ReViCe, which shows 2–6% performance overhead in PARSEC and SPEC (compared to InivisiSpec, a reduction of up to 15%).

The rest of the paper is structured as follows: The background and threat model in Section II, main idea of ReViCe in Section III followed by implementation in Section IV. Section V evaluates ReViCe. The final section concludes the paper. We also provide security analysis of CleanupSpec [30], detailed operations of ReViCe, and security analysis with proof-of-concept in Appendix A, B, and C, respectively.

II. Background and Threat Model

A. Spectre and Meltdown

Spectre attacks [11], [13]–[15], [17], [20] exploit CPU speculation (e.g., branch prediction) to execute malicious logic which is not allowed in the victim program. The attacks generally involve mistraining of the predictors to manipulate the victim. The misspeculated execution then can leak secret information via a microarchitecture side channel, before the execution is squashed by the CPU. Meltdown attacks [12], [16], [18], [19] exploit a different type of vulnerabilities, by violating hardware protection such as page table or system register protection. Meltdown exists due to inherent delay in the exception delivery to be handled until retirement of the faulty instruction; meanwhile, the faulting instructions can still obtain protected data during the transient execution.

B. Threat Model and Attack Scenarios

We assume a typical threat model for Spectre and Meltdown [11], [12], [29]. The attacker intends to retrieve secrets from a victim program by manipulating CPU speculation in the victim or bypassing hardware protection enforced by the OS. The attacker has access to the source code of the victim program and the OS. In addition, it also has knowledge about the underlying microarchitecture. We assume that the CPU and the OS are correct and trusted by the victim.

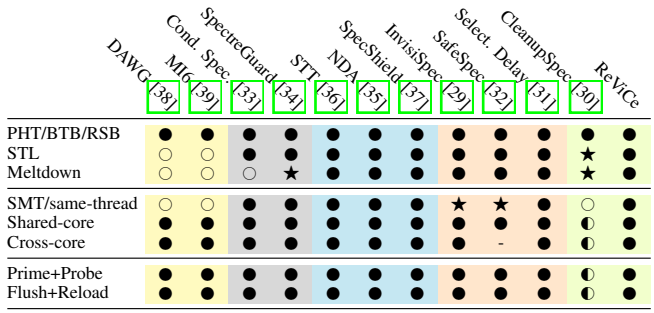
Our attack scenarios include three primary factors that can be exploited by the attackers:

1) Speculation or Squashable Operations: Variants of Spectre exploit misprediction on branch directions (PHT [11]), branch targets (BTB [11]), return targets (RSB [20]), or store-to-load dependency (STL [15]). Meltdown, on the other hand, depends on exceptions, such as page protection faults [12], device-not-available faults [16], or terminal faults [18], [19].

2) Cache-based Side Channels: This paper focuses on two types of cache timing channels as representative examples:

- **Eviction-based attacks**: Prime+Probe [45], [49]–[51] can detect eviction of lines in the same cache set(s) as the lines accessed by the victim.
- **Reload-based attacks**: Flush+Reload [46], [52]–[54] and Evict+Time [45] relies on detecting whether a shared line is recently accessed by the victim.

Cache timing attacks also include techniques that exploit cache replacement policies (e.g., PseudoLRU) [55] or replacement in the cache directory [56]. Another type of cache-based side channel attacks, such as Flush+Flush [48],



○: No defense; ◐: Partial defense; ●: Full defense; ★: Possible defense

Table I: Comparison of the existing hardware mitigations.

mainly work across cores to detect the difference of invalidation latency. ReViCe currently does not handle these side channels [48], [55], [56] and focus on inclusive caches, but we leave the mitigation of these attacks as future work.

3) Shared-core vs. Cross-core: Depending on the side channels, an attacker can potentially share the core with the victim or run on a different core, to observe side channels either before or after the malicious execution is squashed.

ReViCe only handles leakage from transient execution based on cache timing channels. Software or hardware side channels that exist without speculation is orthogonal to our solution. **Any execution that is speculated correctly or not squashed for exceptions is trusted for not leaking any secret.** Other out-of-scope cases include: (1) Potential Spectre and Meltdown variants based on other side channels, such as TLB timing [57], cache directory timing [56], and branch prediction histories [58], [59], and (2) Foreshadow [18], i.e., L1 terminal fault, that targets Intel SGX.

C. Existing Approaches

Many hardware solutions have tackled Spectre and Meltdown using various kinds of approaches (listed in Table I):

1) Blocking Side Channels: DAWG [38] and MI6 [39] partition caches or other hardware such as TLB, to block side channels through resource isolation in general. But they are limited to protecting domains across context switches.

2) Restricting Speculation: Conditional Speculation [33] and SpectreGuard [34] use software hints to disable speculation on sensitive instructions. This approach applies to all side channels, but requires software annotations.

3) Restricting Data Propagation: SST [36], NDA [35], and SpecShield [37] stall tainted, speculative instructions that may leak information. This approach applies to all side channels and requires no annotation, but incurs higher performance overhead due to restricting speculation.

4) Delaying or Hiding Speculative Update: InvisiSpec [29] and SafeSpec [32] isolate speculative loads until the operations can be safely exposed. Selective Delay [31] also uses value prediction to isolate the impact of speculative loads in caches. On the other hand, CleanupSpec [30] and ReViCe both allow speculative installation of the cache line, keeping the operation hidden from the attacker until it is safe. These approaches have low performance overhead, but requires careful defense against various side channels and attack scenarios. We observe several attacks against CleanupSpec, which are describe in Appendix A.

III. Speculative Cache Side-Channel Prevention

ReViCe consists of two major ideas: (1) restoring cache updates from misspeculated loads using a victim cache (thereby, Undo'ing the effect of speculation) and (2) hiding the timing difference of accessing a speculatively loaded line by injecting jitters. In this section, we present the design of ReViCe by first defining the terminologies, followed by ReViCe's workflow and hardware. More details of ReViCe's operations are described in Appendix B.

A. Terminologies

Speculation Source: If an instruction I causes a subsequent instruction (in program order) J to execute speculatively, we refer to I as the Speculation Source (SS) for J. Whether I can behave as a speculation source for J depends on the threat model. When I is resolved and the speculation turns out to be correct, I and subsequent instructions commit from the processor pipeline as usual. However, if I is misspeculated, all subsequent instructions in the pipeline are squashed when I reaches the head of the reorder buffer (ROB). Note that each SS has its own condition of resolving the speculation. We list the SSes and the resolution conditions for four Spectre variants as well as all the Meltdown variants as below:

Attack variants	SS	Resolution
Pattern History Table (PHT)	Conditional jump	Branch resolved
Branch Target Buffer (BTB)	Indirect jump	Branch resolved
Return Stack Buffer (RSB)	Function return	SS retired
Store-to-Load (STL)	Memory store	Address resolved
Meltdown variants	Mem. or reg. access	SS retired

Table II: Speculative sources (SS) and resolution conditions.

Speculative and Non-Speculative Load: When a load is preceded according to the program order by at least one speculation source in the pipeline, the load executes and accesses memory speculatively. Such a load is referred to as a Speculative Load (SL). If all speculation sources are resolved correctly, the SL cannot be squashed from the pipeline anymore and becomes a Non-Speculative Load (NSL).

Victimize: When a newly brought cache line replaces an existing line, the replaced cache line is kept in the victim cache. We refer to this event as *victimizing* the replaced line.

Replace: When a cache line is evicted from both the main cache and its associated victim cache, this cache line is regarded as *replaced* in the current cache level. Such an event is called *replacement*.

Restore and Confirm Messages When an speculation source associated with an SL is misspeculated, ReViCe sends a *Restore* message to the cache hierarchy to restore any change caused by the SL. If all speculation sources are resolved correctly, ReViCe sends a *Confirm* message to make the change permanent i.e., visible globally (to other caches) in the cache hierarchy.

B. Workflow of ReViCe

A ReViCe operation contains the following steps:

1) Determining speculative loads: In a processor pipeline, only loads can speculatively update the cache. Therefore, ReViCe adds a flag to each load instruction entry in the pipeline to indicate whether it is speculative or not. The flag, called Speculative Load Flag (SLF) indicates a load as an SL if its value is 1. Otherwise, the load is an NSL. When a load is issued, ReViCe hardware counts how many SS's are in the pipeline before the load. If there is at least one SS before the load, its SLF is set to 1. As SS's are committed from the pipeline, SLF's of some later loads are cleared, if there are no more SS's in the pipeline before those loads.

2) Accessing the cache hierarchy: In case of an SL, the cache checks if the SL causes a cache hit. If so, the cache returns the data immediately. However, if it is a cache miss, a request is sent through the cache hierarchy to bring the line. The new line is kept in the cache and the victimized line (if any) stays in the victim cache (as long as the load remains an SL). The cache and victim cache keep information to track the speculative line and the corresponding victimized line. A NSL is handled without any alteration to the original cache protocol, unless the line hits on a speculative line. In that case, ReViCe adds some jitter to hide any timing difference. In order to keep the caches free from security flaws, ReViCe maintains the following four security principles:

Principle 1 (P1): A speculative line cannot replace another speculative line. This prevents a chain of speculative cache line replacing each other (which can potentially make the cache susceptible to security flaws in some corner cases).

Principle 2 (P2): A restoration line cannot be removed from the victim cache. This ensures that restoration from misspeculation is always possible.

Principle 3 (P3): A speculative load cannot downgrade the coherence state of either a non-speculative line or a speculative line in another cache. This hides the effect of the speculative line on other cache lines.

Principle 4 (P4): A non-speculative line cannot be replaced by a speculative line without moving to the victim cache.[1]

If ReViCe violates P1, there can be a scenario where a sequence of speculative lines replaces one another, forming a chain in the victim cache and making the restoration much harder. In case of P2, if ReViCe removes a restoration line from the victim cache, ReViCe loses the ability to restore. In case of P3, if ReViCe allows a speculative load to downgrade a non-speculative line (from Exclusive or Modified to Shared state) or to downgrade a speculative line (from Exclusive to Shared state, a speculative line cannot be Modified), not only it complicates restoration but also creates a coherence based side channel [60]. When there is possibility of violating P1, P2 or P3, the associated SL is stalled until the violation is no longer possible. Finally, P4 is necessary so that an NSL remains unaffected by an SL. These principles provide required security properties of ReViCe (See Appendix C-A).

[1]The victim cache is used for both conventional purpose and mitigation of cache side channel. SLs might indirectly evict NSLs from the victim cache. To enforce stronger security, the victim cache can be used only for security purpose with negligible performance overhead.

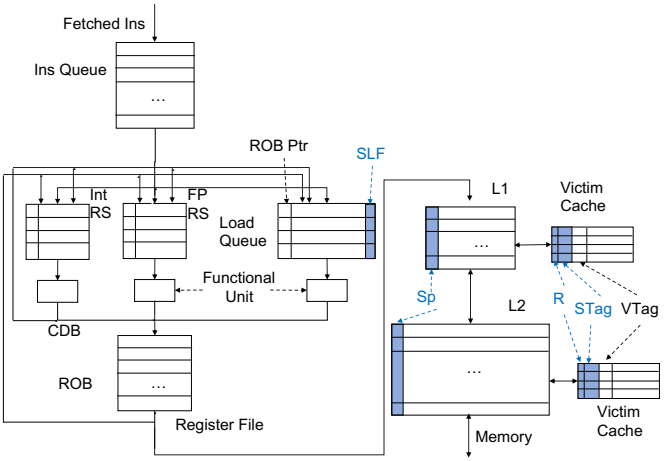


Figure 2: Additional hardware for ReViCe in a typical out-of-order machine. Additional hardware is shown in Blue. RS=Reservation Station, CDB=Common Data Bus, ROB=Reorder Buffer.

3) Restoration and confirmation: If an SL is misspeculated, it gets squashed and sends a Restore message to the cache. Upon receiving the Restore message, the cache invalidates the speculative line and restores the victimized line from the victim cache, giving an illusion that the SL had never happened. On the other hand, if the speculation is correct, the SL becomes an NSL and sends a Confirm message to the cache to clear any information related to speculation and the victimized line. The cache also delays the updates to the LRU bits until speculation is resolved.

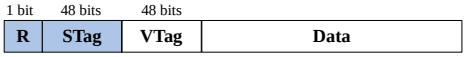
C. ReViCe Hardware

Consider a typical out-of-order multicore processor with two levels of cache. The L2 is shared among cores, with a directory implementing the cache coherence protocol. Each cache has a victim cache. Figure 2 shows the hardware extensions for ReViCe. The extensions are as follows:

1) Load Queue: Load queue has a new field in each entry - SLF. It determines if a load is speculative or not.

2) L1/L2 Cache: A Sp (speculative) bit is added in each line to indicate whether the line is brought by an SL.

3) Victim Cache: Figure 3 shows an extended victim cache line entry. In a victim cache, each line v is identified by a tag $VTag$. ReViCe extends v with a bit R and another tag $STag$; R indicates whether v has been victimized by a speculative line s (i.e., one with $Sp = 1$) and thus can be used for restoration. If v has $R = 1$, we call v a restoration line. For such a line, $STag$ contains the tag of the speculative line s that initially victimized v.



R: Restoration Bit
STag: Tag of the speculative line *s* that victimized *v*
VTag: Tag of the line *v* that is victimized

Figure 3: Victim Cache Block.

D. Supporting Different Memory Models

The operation of ReViCe is described assuming a TSO memory model. However, other memory models such as Release Consistency (RC) [61] does not require any significant change to accommodate ReViCe. Unlike TSO, RC allows reordering of loads to different cache lines. Therefore, ReViCe can support RC with the following changes:

- Whenever multiple loads are ready to send Restore or Confirm messages, they can be sent in any order.
- For TSO, even if a speculation source is correctly speculated, an SL needs to check if any earlier load is misspeculated. If so, the SL needs to send a Restore message. However, for RC, since any pair of loads can be arbitrarily reordered, the SL does not need to check misspeculation of earlier loads.

IV. Implementation Details

In this section, we first show how to find a restoration line from a speculative line. Then, we present the jitter implementation, followed by the justification of our choice of victim cache size as well the absence of deadlocks.

A. Restoration and Speculative Line Lookup

The victim cache is extended with additional tag ($STag$) to facilitate a mapping between a restoration line, r and the associated speculative line, s. To find r from s, $STag$s of the victim cache are selected to match the tag of s. r will be the one with a valid matching. To find s from r, the main cache is searched with $STag$ using a normal lookup.

B. Implementing Jitter

Jitter is added in two cases to hide the existence of a speculative cache line. *First*, a non-speculative access that hits on a speculative cache line can be observed as a latency difference in cache access time. To prevent, the response is delayed by jitter. The length of jitter is based on the level of the current cache and the state of the line (SL or NSL) in the lower level caches. For example, in L1 cache, length of jitter should be memory access latency if the corresponding cache line in the lower level cache is an NSL. Otherwise, L2 hit latency is used. *Second*, a write access to a shared cache line could be a side channel by measuring the latency difference. This is because invalidation is on the critical path of write and invalidation of SL increases the latency. This could be more prominent in a large scale system with multiple cores distributed. For example, the non-speculative sharer could be the next-hop neighbor of the writing node whereas the speculative sharer could be the one at the furthest side. ReViCe puts jitter to have the worst-case-latency of invalidation for all writes requests regardless of any sharing SL or NSL. Jitters for each case are fixed length in our experiment and a variable length jitter could be implemented by periodically sampling access latencies.

Figure 4 shows the implementation of the jitter in L2. We augment each Transaction Status Holding Register (TSHR) with a down counter for jitter. Note that TSHRs are L2 controller's internal buffers that keep track of outstanding or stalled transactions. When a non-speculative access hits on a speculative line, for example, the counter associated with the access's TSHR is initialized with the jitter length provided by the jitter generator. ReViCe decrements the jitter counter every cycle and triggers a response when the counter is 0.

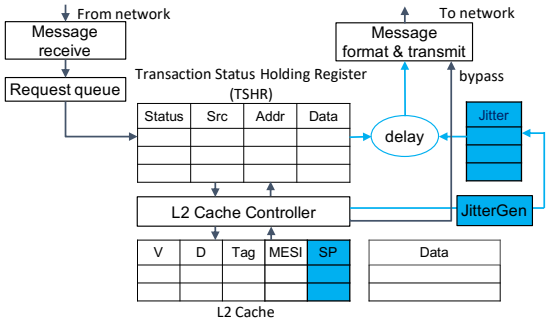


Figure 4: Implementation of Jitter mechanism.

C. Victim Cache Size

In order to find a suitable size for the victim cache, we run experiments using a large (e.g., 1k entries) victim cache, and sample victim cache occupancy every 100 cycles. We observe that L1 victim cache holds 13 restoration lines at most whereas for L2, this number is 5. 99% of the samples have less than 4 restoration lines in the victim cache. Therefore, we choose 16-entry victim cache for L1 and each L2 bank. We use this configuration for multithreaded benchmarks, PARSEC [40]. Since single threaded applications like SPEC CPU2017 [62] are sensitive to victim cache size, we use 32-entry victim cache for L1 and 64-entry victim cache for each L2 bank.

D. Deadlock Freedom

ReViCe stalls speculative load accesses in 4 cases (Figure 8) but never stalls any non-speculative memory access. To show that ReViCe is free of deadlock, we emphasize on two properties: *First*, in cases when an SL is stalled, the stalled SL never ends up stalling an NSL. That implies that each processor can still make forward progress through NSLs and other non-speculative instructions. *Second*, it is safe to stall an SL. This is because the speculation sources eventually get resolved and they either squash the stalled SL or make it non-speculative. In the later case, the stalled load is confirmed to become an NSL to proceed without any further stall. Moreover, Restore and Confirm messages from either case are never blocked inside any cache controller. Therefore, a cache controller continues to operate on requests. In a nutshell, the two properties together ensure that there will not be a circular stalling scenario among a sequence of SLs and NSLs, the processors will keep making forward progress through NSLs and will eventually either squash the stalled SLs or retry them as NSLs. Therefore, ReViCe cannot suffer from any deadlock.

Parameter	Value
Architecture	8 ARM cores for PARSEC, 1 X86 core for SPEC & PoC
Core	2GHz, Out-of-Order, no SMT, 32 Load Queue, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
L1-I Cache (Private)	32KB, 64B line, 4-way, 1-cycle round-trip lat, 1 port
L1-D Cache (Private)	64KB, 64B line, 8-way, 1-cycle round-trip lat, 3 Rd/Wr ports
L2 Cache (Shared)	inclusive, Per core: 2 MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max),
Cache Coherence	Directory based MESI
Cache Replacement	Pseudo LRU
Network	4x2 MESH, 128 link width, 1 cycle latency per hop
DRAM	Built-in memory model in Gem5 [65]
Victim Cache	64B lines, fully associative, 16 blocks in L1-D & L2 for PARSEC, 32, 64 blocks in L1-D & L2 for CPU2017, respectively.

Table III: Parameters of the simulated architecture.

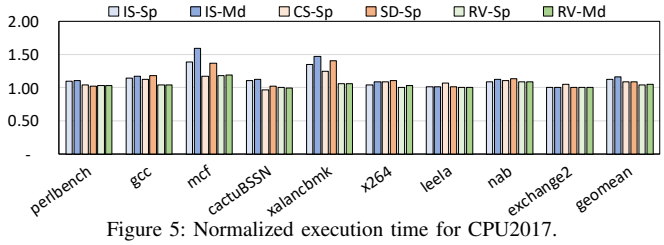


Figure 5: Normalized execution time for CPU2017.

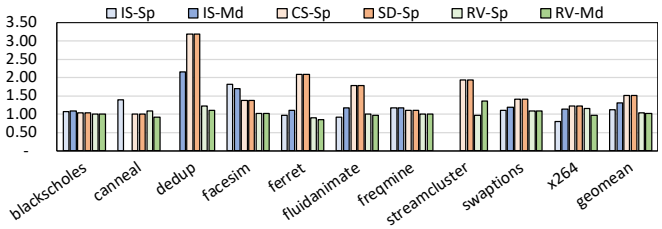


Figure 6: Normalized execution time for PARSEC.

Components	Area (mm^2)			Energy (nJ/access)		
	Base	IS	RV	Base	IS	RV
L1 buffer	0.0343	0.0343	0.0409	0.0319	0.0319	0.0373
L2 buffer	0.0379	0.0824	0.0448	0.0328	0.0440	0.0388
LSQ	0.0698	0.0698	0.0835	0.6900	0.6900	0.8171

Table IV: Area and energy overhead.

V. Evaluation

We evaluate performance, area and energy of ReViCe and other techniques using Gem5 [44] with McPAT and CACTI [63], [64] as shown in Table III. We run SPEC CPU2017 [62] on a single core and the PARSEC [40] on 8 cores. For CPU2017, we use the *reference* input size and run for 1 billions instructions after fast-forwarding 10 billions instructions. For PARSEC, we use *simmedium* input size and simulate the region-of-interest (ROI) until any thread reaches 500 million instructions.

The baseline is a conventional insecure processor. We compare ReViCe against InvisiSpec (IS) [29], [66], CleanupSpec (CS) [30], [67], and Selective Delay (SD) using Oracle value predictor with 16% prediction rate [31].[2] We use different speculation sources for Spectre and Meltdown as described in Section III-A. Speculation conditions for Spectre and Meltdown are the same as described in Section III-B. Therefore, we differentiate Spectre and Meltdown using short name Sp and Md respectively.

Appendix C includes a security analysis on ReViCe, as well as an evaluation using an attack suite with different cache-based side channels and concurrency models.

A. Performance Evaluation

In this section, we summarize the results followed by an in-depth analysis of performance overheads.

1) Execution Time: Figure 5 and 6 show the execution time of different hardware defense mechanisms normalized to the *Base*. In general, ReViCe incurs less performance overhead than the other approaches. *RV-Sp* and *RV-Md* incur only 4% and 6% overhead for CPU2017 and 4% and 2% overhead for PARSEC on average compared to *Base*.

2) Comparison to Base: In ReViCe there are two factors of performance overhead. First, ReViCe stalls and adds jitter for some accesses. We profile jitter and stall cycles over the load round-trip latency. We observe that across benchmarks, it is only up to 0.01% of overhead. Second, ReViCe sends additional *Confirm* and *Restore* messages which could cause network congestion. Our experiment shows that *RV-Sp* and *RV-Md* increase the average packet latency by 13% and 9% compared to *Base*. However, the network is not congested enough to degrade the performance.

3) Comparison to InvisiSpec: IS-Sp and IS-Md incurs 12% and 16% of average overhead for CPU2017 benchmarks, and 12% and 30% for PARSEC benchmarks. This is 2.5×–15× more performance overhead compared to ReViCe. The performance difference stems from shorter processing time of Confirm compared to InvisiSpec's Expose requests. Expose message is processed as normal load request except when data is stored in Speculative Buffer (SB). Due to early update of ReViCe, it discloses the cache line by clearing the *Sp* bit without incurring any coherence state transitions, thereby reducing the overhead. Moreover, Exposure operation increases the average packet latency by 13% and 30% for SI-Sp and SI-Md, which is 3× longer than ReViCe's. This is because of more data retrieval from the memory which increase the number of packets. In addition, the delayed update of InvisiSpec causes more blocking which reduces issue rate of load operation in the processor pipeline to 81% and 73% for CPU2017 and PARSEC, respectively.

4) Comparison to CleanupSpec: CleanupSpec [30] incurs 8% and 51% average performance overhead for CPU2017 and PARSEC, respectively. The overheads are 2×–12.8× more than ReViCe. The difference is due to the different ways in which speculative accesses are handled—ReViCe uses jitter and Confirm/Restore coherence requests whereas CleanupSpec sends out Dummy requests and uses a fixed window size (200 cycles). Most of the speculation windows are less than 200 cycles. ReViCe reduces the window size by sending Confirm requests immediately. We observe that CleanupSpec has 215 average cycles of speculation window which is 7× bigger than ReViCe. The experiment also shows more coherence requests being stalled in CleanupSpec. Unlike Dummy request, jitter just occupies one TSHR entry and does not block incoming requests to the same cache line, resulting in the reduction of pipeline stalls.

[2]We use implementations of IS and CS from the authors' git repository. CS is modified because it does not support multi-core simulation. We could not finish some benchmarks with IS due to one assertion failure.

5) Comparison to Selective Delay: We re-implemented Selective Delay [31] with an oracle value predictor with 16% accuracy. Selective Delay incurs 9% average performance overhead for CPU2017, which is 2× more than ReViCe. This is because SD-Sp flushes the CPU pipeline and re-executes in case of misprediction. Selective Delay spends 22% more cycles to squash instructions compared to ReViCe.

B. Area and Energy Estimations

RV-Sp and *RV-Md* incur minimal area and energy overhead, (Table IV). The extra tag/entry (49 bits) added in victim caches results in 19% and 18.1% area overhead in L1 and L2 victim caches, respectively. Our techniques incur 16.9% and 18.28% energy overhead compared to *Base*'s victim cache. Additional bits added in load and store queue (LSQ) (17bits/entry) incur 19.69% and 18.42% area and energy overhead, respectively. As compared to the processor chip, the additional area and energy overhead is less than 0.1%.

VI. Conclusions

The discovery of Spectre and Meltdown reveals the security risk of abusing speculative execution in CPUs. This paper shows that microarchitecture can be designed with both performance and defense in mind. ReViCe mitigates speculation based attacks by allowing early updates while delaying the actual exposure. ReViCe outperforms other hardware solutions with minimal additional hardware. We believe that our proposed Undo based solution lays down a general foundation toward defending any microarchitecture states against speculation based attacks in the future.

Acknowledgment

This work is supported by the startup package provided by Texas A&M University and NSF under Grant No. 1652655. The authors would like to thank the shepherd and anonymous reviewers for their helpful comments and suggestions.

References

[1] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[3] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[4] Jason RC Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Notices*, volume 30, pages 67–78. ACM, 1995.

[5] Thomas Ball and James R Larus. *Branch prediction for free*, volume 28. ACM, 1993.

[6] James E Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

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

[8] Andreas Moshovos and Gurindar S Sohi. *Memory dependence prediction*. PhD thesis, Ph. D. thesis, University of Wisconsin-Madison, 1998.

[9] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 252–263. ACM, 1997.

[10] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, (12):7–21, 1978.

[11] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[12] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[13] CVE-2017-5754 (Rogue Data Cache Load). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754, 2017.

[14] CVE-2018-3640 (Rogue System Register Read). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3640, 2018.

[15] CVE-2018-3639 (Speculative Store Bypass). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639, 2018.

[16] CVE-2018-3665 (Lazy FP State Restore). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3665, 2018.

[17] CVE-2018-3693 (Bounds Check Bypass Store). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3693, 2018.

[18] 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 *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 991–1008, 2018.

[19] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[20] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, 2018.

[21] Intel. Intel analysis of speculative execution side channels. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html.

[22] Liam Tung. Linux meltdown patch: 'up to 800 percent CPU overhead', netflix tests show. https://www.zdnet.com/article/linux-meltdown-patch-up-to-800-percent-cpu-overhead-netflix-tests-show/, 2018.

[23] Greg Kroah-Hartman. Linux 4.14.11 meltdown mitigation using KPTI. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11, 2018.

[24] [patch] d41723: Introduce the "retpoline". http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html

[25] Spectre mitigations in msvc. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, Feb 2019.

[26] Speculation barrier for arm. https://github.com/ARM-software/speculation-barrier

[27] Lars Müller. Kpti a mitigation method against meltdown. *Advanced Microkernel Operating Systems*, page 41, 2018.

[28] Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[29] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[30] Gururaj Saileshwar and Moinuddin K Qureshi. Cleanupspec: An undo approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 73–86. ACM, 2019.

[31] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 723–735, New York, NY, USA, 2019. ACM.

[32] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-

Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. pages 60:1–60:6, 2019.

[33] 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 *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 264–276. IEEE, 2019.

[34] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, pages 61–1, 2019.

[35] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 572–586. ACM, 2019.

[36] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968. ACM, 2019.

[37] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–164. IEEE, 2019.

[38] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[39] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. *arXiv preprint arXiv:1812.09822*, 2018.

[40] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[41] Moinuddin K Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787. IEEE, 2018.

[42] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro. Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. *IEEE Computer Architecture Letters*, 19(1):9–12, Jan 2020.

[43] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 364–373. ACM, 1990.

[44] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[46] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[47] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[48] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[49] Colin Percival. Cache missing for fun and profit, 2005.

[50] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[51] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 62. ACM, 2019.

[52] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[53] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+ reload attack on aes. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 111–126. Springer, 2015.

[54] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[55] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA, pages 139–152, February 2020.

[56] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE S&P 2019*, 2019.

[57] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Home-Alone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[58] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 312–320, New York, NY, USA, 2007. ACM.

[59] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 693–707, New York, NY, USA, 2018. ACM.

[60] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[61] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 15–26. IEEE, 1990.

[62] SPEC CPU2017. https://www.spec.org/cpu2017.

[63] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*, pages 694–701. IEEE Press, 2011.

[64] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480. ACM, 2009.

[65] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. Simulating dram controllers for future system architecture exploration. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 201–210. IEEE, 2014.

[66] Mengjia Yan. Git hub repository of invisispec implementation. https://github.com/mjyan0720/InvisiSpec-1.0, 2019.

[67] Gururaj. Git hub repository of cleanupspec implementation. https://github.com/gururaj-s/cleanupspec, 2019.

[68] Moinuddin K Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 360–371. ACM, 2019.

[69] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[70] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. pages 249–266, August 2019.

[71] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 897–912, 2015.

APPENDIX A
SECURITY ANALYSIS OF CLEANUPSPEC

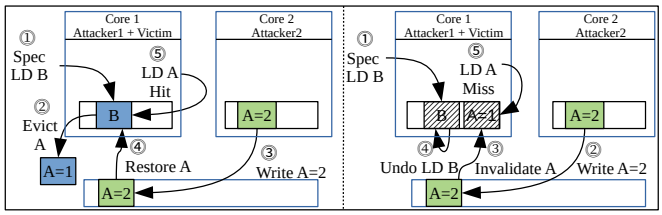
In this appendix, we describe several possible attack scenarios on CleanupSpec [30], to help identify the key requirements to designing a secure Undo-based solution.

A. Shared-Core Eviction Attacks in L1

Before undoing, CleanupSpec hides a speculative load by sending a dummy request to L2 and waiting for its response. However, if a line is speculatively evicted, CleanupSpec cannot speed up a subsequent load to hide the eviction; this allows a SMT or shared-thread attacker to potentially steal the secret within the speculation window. Such a scenario does not apply to shared-core attackers (but different threads) as context switches in CleanupSpec will clean up all speculatively-installed lines.

B. Shared Cache Lines

CleanupSpec defends against cross-core attacks using Cache Address Encryption (CEASER [41]) in L2. This approach stops attackers from identifying mutually evicted lines, but does not prevent attacks that target a single line shared between the attacker and the victim. Reload-based attacks (Flush+Reload) or invalidation-based attacks (e.g., Flush+Flush) generally rely on shared lines, which can occur with shared libraries or Kernel Same-page Merging (KSM), or with a same-thread attacker.



(a) Set(A) = Set(B) (b) Set(A) ≠ Set(B)

Figure 7: A cross-core invalidation attack on CleanupSpec [30]. (a) show a victim's line B evicts an attacker's line A, with core 2 invalidating A. A is eventually restored in core 1, causing the attacker to get a hit with A. In (b), the attacker gets a miss because A is invalidated without eviction.

C. Cross-Core Invalidation Attacks

We can show a case where invalidation from another core can cause different Undo results in CleanupSpec (see Figure 7). Assume the victim evicts a cache line (line A) of the attacker in L1. If another attacker writes to A on a different core, A will be updated in L2 without sending invalidation to the victim's core. Later, after undoing, the victim's L1 will have the latest A, giving the attacker a cache hit. If the victim never evicts A, and A is invalidated before undoing, the attacker will get a cache miss in L1.

D. Algorithmic Attacks on L2 Randomization

CEASER [41] in CleanupSpec uses Cache Address Encryption to prevent the attacker from determining a set of mutually evicting lines in L2. To disrupt brute-force attacks, CEASER rotates the keys every 100 loads / line and uses two keys for gradual remapping. A follow-up work [68] shows that such a remap rate is insufficient, and proposes CEASER-S, an additional skewed mapping scheme across cache ways as a new mitigation. Budduna et al. [42] examined the randomization scheme of CEASER and CEASER-S based on Cache Address Encryption. They discovered that both CEASER and CEASER-S use a low-latency block cipher (LLBC) which generates encrypted output as linear functions of the input and the key. As a result, the cipher does not invalidate eviction sets even after key remapping.

E. Oracle Attacks on L2 Randomization

Even if CEASER could choose an encryption scheme that invalidate eviction sets properly, we show that the key remapping windows of CEASER and CEASER-S are not large enough to be immune to oracle attacks. We discover that, if the attacker can feed an input X to the vulnerable code (a common scenario for Spectre), to cause out-of-bound access to $array[X] = Y$, she can create an oracle by recording which line is evicted in L2. Later, by observing the same line being evicted, the attacker can infer $secret = Y$. This attack is much faster than brute-force, since Prime+Probe for one value only needs 2 loads / line. This attack cannot be thwarted by CEASER-S since its skewed mapping scheme only adds another layer of encryption to the output which is still deterministic for the oracle attack.

APPENDIX B
DETAILED OPERATIONS OF REVICE

We show the detailed operations of ReViCe considering the TSO memory model [69]. Section III-D explains how other memory models can be supported.

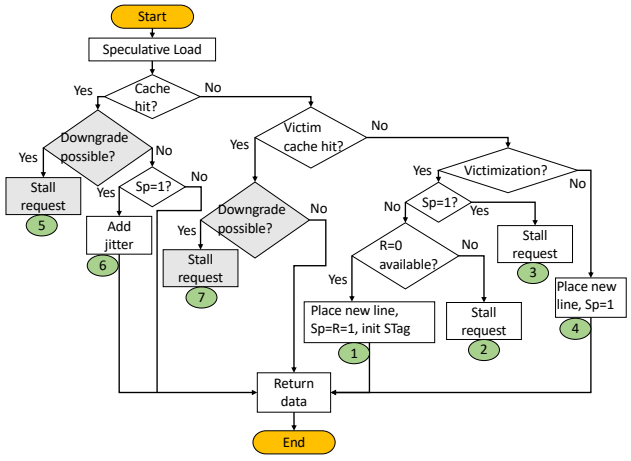


Figure 8: Flow of Speculative Load (SL). The shaded area applies to the shared L2 cache.

A. Speculative Load Memory Access

Figure 8 shows how a memory request from an SL is handled. Let us consider L1 cache first. If the request causes a cache hit on speculative line, ReViCe adds some jitter (details in IV-B) during the response to conceal the fact that cache

line has been already accessed speculatively (⑥). If it hits on non-speculative line in the cache or victim cache, the data is returned directly. In case of a miss, the cache controller allocates a line in the main cache before bringing the new line. If the allocation does not victimize any other cache line, then the allocation succeeds and the cache controller sends a miss request to the lower level. When the requested line arrives, the cache controller places the line in the allocated spot and marks it as speculative by setting Sp to 1 (④). However, if the allocation victimizes any non-speculative cache line (one with $Sp = 0$), ReViCe puts it in the victim cache as a restoration line only if it does not evict any existing restoration line, thereby maintaining P2 (①). In other cases, the allocation tries to violate P1 or P2 and thus, ReViCe stalls the SL (② & ③).

A request from an SL is handled by L2 (shared cache) in a similar fashion except for the following changes (highlighted in Figure 8). If the request causes a downgrade of the cache line in other caches, ReViCe stalls the SL maintaining P3 (⑤ & ⑦).

B. Non-Speculative Load Memory Access

Figure 9 shows how a request from an NSL is handled. The figure is applicable for L1 or L2. If the memory access causes a hit on a non-speculative line, cache returns the requested data. However, if the hit occurs on a speculative line, there are two options: (i) cache can return data immediately (as if it were a regular cache hit) or (ii) cache can add jitter before returning the data. In case of option (i) since a memory access which would have been a cache miss in the absence of any speculation becomes a cache hit, there will be a side channel. Therefore, we adopt option (ii) (marked by ① in Figure 9) to disguise a cache hit on a speculative line as a cache miss. Section IV-B explains how to choose jitter length and implement it in hardware.

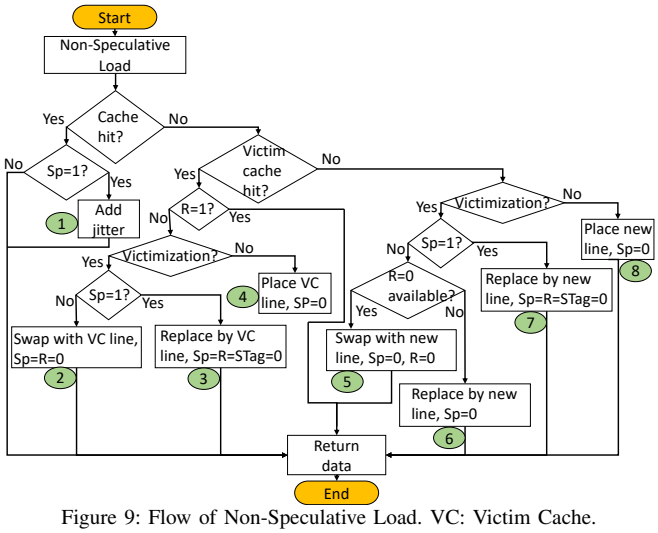


Figure 9: Flow of Non-Speculative Load. VC: Victim Cache.

If an NSL causes a cache miss, ReViCe checks in victim cache. If the access request hits on a cache line in victim cache, data is returned as usual. The cache line is then brought back to the main cache only if it is not marked as a restoration line (②, ③ & ④). If the access request does not cause a hit in victim cache, the requested cache line is brought from lower level and placed in the main cache. If the new cache line victimizes any other cache line, the other cache line is either completely evicted from cache (⑥ & ⑦) or put in the victim cache (⑤) depending on whether the other line is speculative and the victim cache cache is full of restoration lines, respectively.

C. Effect on Cache Coherence Protocol

The cache coherence protocol is modified in ReViCe to accommodate speculative loads. Although we assume MESI protocol here, other protocols can be similarly accommodated. The changes to coherence protocols are as follows:

Case 1: Whenever a speculative load causes a cache miss that requires a coherence transaction, the controller issues a $GetSp$ (Get Speculative) message (similar to InvisiSpec [29]). This new message distinguishes a speculative load miss from a non-speculative load miss.

Case 2: When the directory receives $GetSp$ for a cache line that is in (M)odified or (E)xclusive state in some other cache, the controller stalls the request. This is to preserve principle P3. There is no change in coherence protocol if the line is in (S)hared state.

Case 3: When the directory receives $GetSp$, $GetS$ or $GetX$ message (i.e., any load or store miss) for a line only in speculative state in any other cache, the directory adds some jitter before responding to the requester. This is to conceal the fact that the line has been already brought to the shared L2 cache by some SL. Section IV-B explains the details of the jitter mechanism.

Any other case is handled without any change in cache coherence protocol.

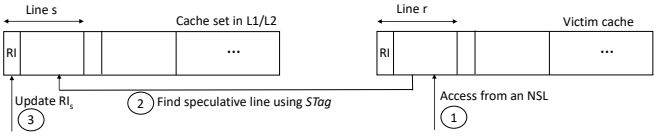


Figure 10: Updating replacement info (RI) for an NSL.

D. Maintaining Replacement Policy

Any alteration in replacement states by misspeculated loads can be exploited as a side channel [55]. Assume that a speculative line s victimizes another line r that is now moved to the victim cache as a restoration line. In order to maintain the correct replacement states, ReViCe delays any update to the replacement state of s (denoted by RI_s) until the speculation is confirmed. The load can then be considered safe and RI_s can be safely updated to reflect the load. Figure 10 explains how an NSL access to r is handled. ReViCe treats RI_s as belonging to the restoration line r. Therefore, it keeps updating RI_s for each NSL access to the restoration line. When the SL gets confirmed, RI_s starts to belong to s and reflect s's LRU state (i.e., as if the load has just occurred). If r is restored due to misspeculation of the SL, r replaces s and inherits RI_s as it is. Thus, both Restore and Confirm operations maintain correct replacement states.

E. Restoration and Confirmation

After a memory access request is sent from the pipeline, ReViCe decides whether and when to send a Confirm or Restore message. When any speculation source resolves, ReViCe checks if there is any later (with respect to the speculation source) load in the ROB. If the speculation source is misspeculated, ReViCe marks the load as requiring a restore operation. Eventually, when the load does not have any speculation source in front of it in the ROB, ReViCe checks if the load is marked as requiring a restore operation. If so, ReViCe sends a Restore message to the cache hierarchy. Otherwise, ReViCe sends a Confirm message. If ReViCe can send messsages for multiple loads at a particular cycle, the messages are sent in program order (staring from the oldest load). After a Confirm or Restore message is sent, the processor pipeline can consider the confirm or restore operation to be completed and does not need to stall for any response from the cache hierarchy. The cache hierarchy eventually completes the actual operation. This is reminiscent of a store operation.

L1 and L2 cache controllers perform the actual Confirm and Restore operation. If the speculative line corresponding to an SL is not found in L1, L1 controller forwards them to lower level cache. Otherwise, it proceeds to perform the operation. In order to confirm an SL, L1 controller finds the corresponding speculative and restoration line, and clears Sp and R bits, respectively. Replacement bits of the speculative line are updated as if the line is just accessed. In order to restore an SL, L1 controller replaces the speculative line with the corresponding restoration line. Sp bit is cleared and replacement bits of the speculative line are inherited by the restoration line. Meanwhile, L1 controller sends a Confirm or Restore message to L2 controller.

When L2 controller receives a Confirm message and the line is no longer speculative, it updates the replacement bits and discards the message. Otherwise, L2 controller performs the Confirm operation in the same way as the L1 controller. When L2 controller receives a Restore message, it might happen that the line has been already confirmed by a Confirm message from some other L1 cache. In that case, the controller just updates the sharer bitmap of the directory to indicate that the requesting L1 cache no longer holds the line. However, if the line is still speculative when L2 receives a Restore message, the controller first updates the sharer bitmap. If the line is no longer present in any L1 cache, the controller replaces the speculative line with the corresponding restoration line, clears Sp bit. Similar to L1 Restore operation, the replacement bits of the speculative line are inherited by the restoration line. Thus, L2 performs the Restore operation after it receives Restore messages from all sharer L1 caches.

APPENDIX C
SECURITY ANALYSIS AND EVALUATION ON REVICE

In this appendix, we conduct a security analysis of ReViCe and attempt various attack scenarios on ReViCe to show proof-of-concept defense.

A. Security Analysis

We examine the security principles listed in Section III-B2 to verify that ReViCe mitigates Spectre and Meltdown. Since such attacks rely on *misspeculation*, to fully mitigate them, **any execution that is eventually squashed must not expose any cache state changes (we define as Property A)**. We use Figure 8 and 9 to exhaustively examine the possible cases (15 cases in total) to show *ReViCe's actions initiated by SLs are not observable by attackers*. We target speculation based attack, so other cases that are only associated with **instructions that eventually retired are considered *trusted* (we define as Property B)** and are not inspected. The cases are categorized into 4 types of *final actions*: **Adding jitter:** There are two cases when jitter is used—⑥ in Figure 8 and ① in Figure 9. Both cases happen when a load hits on a speculative line and adding jitter makes the latencies identical to cache misses.

Stall: Stalls occur only for SL requests – ②, ③, ⑤, and ⑦ in Figure 8. In case of correct speculation (Confirm), all stalled loads are eventually retired (*Property B*). In case of misspeculation, the stalled SL is eventually squashed and nothing can be observed (*Property A*). If the stall condition is resolved, the SL is processed by other cases.

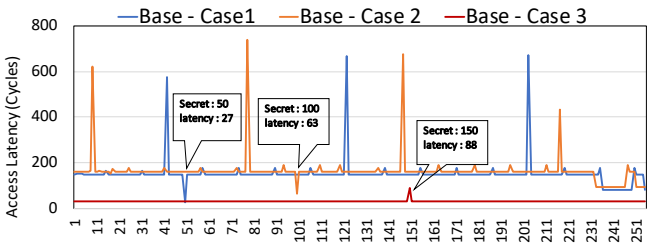
A question is whether stalling can create any new side channels for prior SLs. ⑤ and ⑦ require the possibility of downgrading a line brought by NSL which overshadows other SLs. ② and ③ only stall when it needs to victimize a speculative line or when the VC is full of restoration lines. It is theoretically possible to use ② or ③ to detect the existence of *some* prior SLs, the attacker cannot distinguish from other common cases where the pipeline is simply busy. Thus, this side channel is difficult to exploit.

Installing a new line: ① in Figure 8 and ⑤, ② in Figure 9 victimize one cache line and install a new line. In case ① (Figure 8), accessing the victimized line has the same latency as accessing from the main cache (*Property A*). Also, other NSLs to the same line as newly installed are jittered (*Property A*). Later, if the SL is squashed, the victimized line is restored as the SL never exists (*Property A*). Otherwise, the SL is eventually retired and considered safe (*Property B*). ② (Figure 9) is not related to SLs. ④ in Figure 8 and ④ in Figure 9 install a new line without victimization and also cannot be observed due to jittering and restoration.

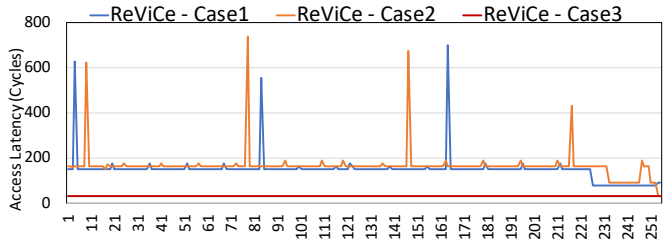
Replacement: ③, ⑥, ⑦ in Figure 9 replace a line in the main cache with a victimized line. In cases ③ and ⑦, the speculative line previously brought by an SL is replaced by the NSL, and the cache states are updated. In case the previous SL gets confirmed to become NSL, the cache changes are considered trusted (*Property B*) and cannot be used to distinguish other prior SL. If the previous SL is misspeculated (Restore), the replaced speculative line has been invalidated from the main cache (*Property A*). ⑥ is related only to NSLs, therefore it is safe (*Property B*).

In addition to examining the security principles, we also show the safety from the cross-core invalidation attack scenario that CleanupSpec cannot protect (See Appendix A).

The vulnerability comes from the different cache states for memory A - LD A hit (⑤) in Figure 7.a or LD A miss(⑤) in Figure 7.b. In stark contrast, ReViCe does not allow SL to evict any cache line, so LD A always causes misses in both Figure 7.a and Figure 7.b.



(a) Attack to *Base*



(b) Attack to ReViCe

Figure 11: Access latency for the three test cases. The secret values of 50, 100, and 150 can be deduced in Baseline

B. *Proof-of-Concept (PoC) Defense*

We evaluate the security of ReViCe using a Spectre test suite containing various attack proof-of-concepts. We assume the attacker will explore as many variants of Spectre as possible. We pick four Spectre variants, including PHT, BTB, RSB, and STL [70]. We combine the variants with three well-known cache side channels, Evict+Time [71], Flush+Reload [48], and Prime+Probe [52]. Then, we exercise the attacks in following three cases of concurrency between the attacker and the victim:

Case 1: Leakage from L1 (Defended by *Restore*): We implement this case by placing the attacker and the victim in the same core. The attacker first trains the branch predictors, runs the *reset* phase of the side-channel attacks, waits for the victim, and then tries to retrieve the secret. As shown in Figure 11a, the PHT attack succeeded to find the secret value 50 with Flush+Reload on the baseline CPU, but cannot leak the secret in ReViCe as shown in Figure 11b. This is because the secret loaded by misspeculation in L1 is invalidated by Restore. All 12 combinations of variants are successfully mitigated.

Case 2: Leakage from L2 (Defended by *Restore*) We implement a cross-core attacker that observes the cache state *after* the speculative execution in the victim. As shown in Figure 11a and Figure 11b, the combination of PHT and Flush+Reload on L2 cache succeeded in baseline CPU but failed in ReViCe. Similar to Case 1, the secret loaded by misspeculation in L2 is also invalidated by Restore. All 12 combinations of variants are successfully mitigated.

Case 3: Leakage from L2 (Defended by *Jitter*) We implement a cross-core attack that observes the cache state *concurrently* with the victim execution. For this attack, we use Prime+Probe instead of Flush+Reload for the PoC because `clflush` is too slow in a concurrent attack. As shown in Figure 11a, the attacker observes the secret with higher latency (miss in L2) on the baseline CPU, but cannot observe any difference in ReViCe as *jitter* increased the latency. All four Spectre variants are mitigated by ReViCe with Prime+Probe.