CleanupSpec: An "Undo" Approach to Safe Speculation

Gururaj Saileshwar
gururaj.s@gatech.edu
Georgia Institute of Technology

Moinuddin K. Qureshi
moin@gatech.edu
Georgia Institute of Technology

ABSTRACT

Speculation-based attacks affect hundreds of millions of computers. These attacks typically exploit caches to leak information, using speculative instructions to cause changes to the cache state. Hardware-based solutions that protect against such forms of attacks try to prevent any speculative changes to the cache sub-system by delaying them. For example, InvisiSpec, a recent work, splits the load into two operations: the first operation is speculative and obtains the value and the second operation is non-speculative and changes the state of the cache. Unfortunately, such a "Redo" based approach typically incurs slowdown due to the requirement of extra operations for correctly speculated loads, that form the large majority of loads.

In this work, we propose *CleanupSpec*, an "Undo"-based approach to safe speculation. CleanupSpec is a hardware-based solution that mitigates these attacks by undoing the changes to the cache sub-system caused by speculative instructions, in the event they are squashed on a mis-speculation. As a result, CleanupSpec prevents information leakage on the correct path of execution due to any mis-speculated load and is secure against speculation-based attacks exploiting caches (we demonstrate a proof-of-concept defense on Spectre Variant-1 PoC). Unlike a Redo-based approach which incurs overheads for correct-path loads, CleanupSpec incurs overheads only for the wrong-path loads that are less frequent. As a result, CleanupSpec only incurs an average slowdown of 5.1% compared to a non-secure baseline. Moreover, CleanupSpec incurs a modest storage overhead of less than 1 kilobyte per core, for tracking and undoing the speculative changes to the caches.

CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**; • **Computer systems organization** → **Architectures**.

KEYWORDS

Transient-Execution Attacks, Side-channel Attacks, Caches

ACM Reference Format:
Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358314

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358314

1 INTRODUCTION

Speculation-based attacks like Spectre [28], Meltdown [32], Foreshadow [46], etc., have caused considerable concern in the computing industry given that they affect practically every processor-manufacturer and allow a software-based adversary to arbitrarily bypass software and hardware-enforced isolation without detection. These attacks are hard to detect or mitigate in software as they exploit micro-architectural vulnerabilities that are invisible to software. Therefore, commercially deployed software and micro-code patches are limited in being able to mitigate only specific attack variants (e.g. KAISER [16], Retpoline [45], IBRS [23]). As new attack variants continue to be discovered [7, 29, 53], there is a pressing need for broader hardware solutions [33].

Unfortunately, mitigating these attacks in hardware alone with minimal overheads is challenging, as these attacks leverage processor speculation and its side-effects on caches to obtain and leak secrets. Both speculation and caching being the cornerstones of high performance processors, naively disabling them to mitigate the attacks would cause intolerable slowdown. While recent hardware designs [31, 43] have emerged that mitigate Spectre attacks with low overheads by identifying potentially "unsafe" load patterns in the attacks and delaying them, they may not protect against other attacks that do not exhibit such patterns. In this context, there is a need for low-overhead mitigation of current and future speculation-based attacks exploiting caches, in particular data-caches[1].

These attacks can be generalized as having three key components, as observed by recent works [26, 58]: (a) *getting access to a secret* during speculative execution, (b) *speculatively transmitting the secret*, by making a secret dependent modification to the cache state (preserved even after a mis-speculation), and (c) *inferring the secret on the correct path* of execution, using side-channel attacks like Flush+Reload [63]. Stopping any of these three components is sufficient to defeat these attacks. In this paper, we focus on preventing the third component, i.e. the secret inference on the correct path.

When a mis-speculation is detected on a modern processor, the processor invokes a pipeline flush to ensure that all of the data in the pipeline stages gets invalidated. However, a pipeline flush does not affect the content of the cache and any state change caused by the speculative instructions to the cache is retained. It is possible to close this cache side-channel by ensuring that either (a) no state change to the cache is caused by a speculative instruction or (b) the changes caused by the speculative instructions are *undone* when the mis-speculation is detected. We call the former approach a *Redo* approach to safe speculation (as the data may be read twice, once speculatively and a second time to change the cache state). The latter is an *Undo* approach (as the state change is performed and later undone on mis-speculation).

[1] I-Cache, TLB or Branch Predictor can also be used to leak information, but delaying [40] or buffering [25, 58] transient changes to these structures can prevent this. Port-contention [1] is out-of-scope due to its orthogonal nature, like prior works [26, 58].

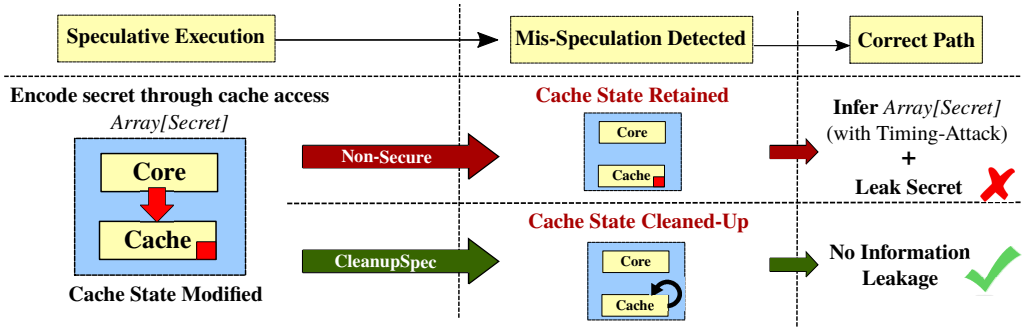


Figure 1: Speculation-based attacks leak secrets using transient instructions, by modifying state of a cache line whose address is based on the secret value. Currently, modifications are retained on the correct path and inferred using timing attacks, to leak the secret. CleanupSpec rolls back changes or ensures changes that remain are randomized, preventing information leakage on the correct path.

A recent proposal, InvisiSpec [58], represents a *Redo-based* approach to safe speculation. To prevent speculatively executed (transient) instructions from making cache changes, for each data load, InvisiSpec first performs a load that fetches the data without making any changes to the cache state and then a second load that changes cache state (once it is deemed safe). Given that most loads are correctly speculated, InvisiSpec incurs the cost of double accesses to the cache hierarchy for most loads. Additionally, InvisiSpec requires the second access to be performed on the critical path before load-retirement, to verify that no memory consistency violations occurred during the *invisible* phase of the load, when the directory did not know the core had the data. As a result, InvisiSpec causes considerable slowdown (initial estimates show 67% on average)[2].

In this paper, we observe that an *Undo-based* approach is better suited to mitigate these attacks with lower overheads. Just as the processor state is flushed when a mis-speculation is detected and any illegal instructions are removed, if the illegal changes to the cache made by transient instructions were also rolled-back at the same time, then any information leakage can be prevented. Such an approach would satisfy correctly speculated loads (which are the common case) with a single cache access. Additionally, overheads to roll-back cache state would only be incurred in the uncommon-case of a mis-speculation. With this insight, we propose *CleanupSpec*, a low-overhead hardware mechanism to prevent speculation-based attacks leaking information through the data caches.

As shown in Figure 1, these attacks use transient instructions to extract a secret value to the correct-path. By encoding the secret value as the address of an array access (*Array*[*secret*]), these attacks install a secret-dependent array line in the cache. This line is retained in the cache even after a mis-speculation is detected. As a result, the secret is accessible on the correct-path with cache timing attacks [63] that infer which line was installed by the transient instruction.

To roll back the cache changes made by transient instructions on a mis-speculation, the first step is to invalidate the cache line installs by the transient instructions. Unfortunately, invalidation alone is not sufficient [30] as information is leaked even through the cache lines evicted when these lines were installed. For example, an adversary could use an attack like Prime+Probe [34] to pre-load sets with its lines and observe which set experienced evictions during line installs by the transient instructions. Thus, the rollback of cache changes by transient instructions is feasible only if we prevent information leakage through evictions, and we need to do so without incurring overheads to buffer all the evictions.

Recent works on cache randomization (e.g. CEASER [38] or Skewed Caches [39, 54]) make evictions benign for larger caches (L2 or LLC) with low overheads. By randomizing the cache-indexing, such proposals remove any discernible relation between co-resident lines in a set. Thus, observing an eviction leaks no information about the installed line causing it. Leveraging this, CleanupSpec uses randomization for L2/LLC (and for the directory [61]), and consequently only needs to rollback evictions from L1 data caches (as existing proposals [50, 51] for randomizing the L1 cache have pitfalls due to its VIPT design - see Section 2.4.2).

To enable L1 data cache rollback, CleanupSpec executes transient loads while tracking their cache side-effects (L1/ L2/ LLC installs and L1 eviction-victim line address). On a mis-speculation, to roll-back the changes due to squashed loads, CleanupSpec invalidates the line in the L1 cache if it was installed speculatively and fetches the evicted line from L2 cache to restore the original line. For changes to the L2/LLC, CleanupSpec merely invalidates the copy there if it was installed speculatively (as L2/LLC evictions leak no information).

Rollbacks and invalidations are enabled with the help of side-effect tracking metadata in the load-store queue and L1/L2 MSHR (requiring <1KB of extra storage per core). These cleanup operations are only needed when a mis-speculation occurs and has squashed transient loads that missed in the L1 data cache and installed new lines. Given the high branch-prediction and L1-cache hit rates in typical applications, these operations are uncommon and cause limited slowdown. Additionally, the perceived cache state after these operations is as if the transient loads did not access the cache, so no information leaks on the correct path. Additionally, the transiently installed lines are also protected from access by other threads within the transient window, so no information leaks even in this window.

In addition to installs and evictions, CleanupSpec also prevents transient changes to replacement state and coherence state from leaking information. To protect the replacement state, it uses Random Replacement policy for the L1 data cache (any replacement policy can be safely used for a randomized L2 or LLC). For coherence state changes, CleanupSpec delays loads that cause such changes (M/E to S) until they are unsquashable on the correct path. As such transitions are infrequent compared to regular loads (<3% of loads cause such transitions – see Section 3.5), this adds negligible slowdown.

[2]While our estimates are close to InvisiSpec results in [58], its authors reached out to intimate us about an updated implementation with lower overheads (refer Section 6.5).

Overall, this paper makes the following contributions:

(1) We propose ***CleanupSpec, an Undo-based Approach*** to mitigate speculation-based cache attacks, with lower overheads compared to prior approaches.

(2) **Mechanisms for Safe Speculative Cache Accesses:**
   (a) **Rollback** changes to L1 efficiently.
   (b) **Invalidate** changes to the Randomized L2/LLC.
   (c) **Randomize** replacement policy for L1.
   (d) **Delay** changes to coherence state in remote cache.

We model CleanupSpec in Gem5 [5], under a threat model where any mis-speculated load could leak information (like prior work [58]). As a proof-of-concept, we demonstrate that it mitigates the Spectre Variant-1 PoC. We also evaluate the performance of CleanupSpec over 19 SPEC-CPU2006 workloads. Compared to a non-secure baseline, CleanupSpec incurs an average slowdown of 5.1%, that is much less than Redo-based approaches like InvisiSpec for a similar threat model. Moreover, CleanupSpec only requires <1KB of storage per core and simple logic, for tracking and restoring cache state changes by transient loads.

2 BACKGROUND AND MOTIVATION

2.1 Threat Model

Modern processors speculatively execute instructions out-of-order to avoid stalls due to control and data dependencies and achieve high performance. This can result in execution of *transient instructions*, i.e. speculatively executed instructions on wrong execution paths. We deem all transient instructions potentially "unsafe" until they retire, to protect against current and future attacks like InvisiSpec [58], unlike others [31, 43] that only focus on Spectre variants.

Adversarial Capability: We assume attacker-executed transient instructions have arbitrary access to secrets. The secret may be accessed from the memory or from a register, or computed, transiently. We assume the secret is transmitted to the correct path only using cache side-channels, as a majority of attacks [7, 28, 32, 46] use them given their high bandwidth. We only consider attacks exploiting the data-cache hierarchy, including private data caches (e.g. L1-D cache) and shared caches (L2/LLC), as other structures like instruction caches, TLB, etc. can be protected with a low overhead with prior works [40, 58]. The adversary may transiently access the cache and modify its state through installs [63], evictions [34], updates to replacement [26] and coherence [62] state and infer changes on the correct path through timing difference on cache accesses.

Out of Scope: We do not consider speculation-attacks using AVX side-channels [42], since they are easily mitigated by disabling the speculative power-up of AVX-units. We also assume speculative hardware prefetching is disabled for caches (similar to [58]), preventing information leakage through training of the prefetcher on transient loads. Similarly, we assume a close-page row-buffer policy for the memory controller, to prevent covert-channels like DRAMA [36]. We further consider out-of-scope side-channels due to SMT port-contention [1], network [49] or DRAM [56] contention, EM radiation or power, given their orthogonal nature.

2.2 Speculation-Based Attacks

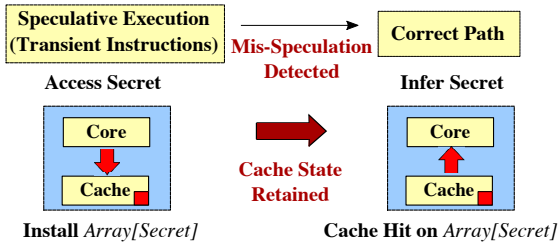


Figure 2: Recipe for speculation based attacks, using the cache as a transmission channel for leaking secrets.

Attacks like Spectre [28], Meltdown [32], etc., leverage transient instructions to access secrets, bypassing any software checks. When a mis-speculation is detected by the processor, these transient instructions are squashed. Hence, these attacks attempt to exfiltrate the secret to the instructions on the correct path of execution before mis-speculation is detected – most commonly using the Flush+Reload [63] attack on the cache. As shown in Figure 2, these attacks transiently access an array entry, whose index is computed using the secret value (the array is completely evicted from the cache previously, using `clflush`). This installs array entry *Array*[*Secret*] into the cache. As the cache state is preserved even after a mis-speculation, the attacker infers the secret on the correct path by accessing each array entry and observing which one has a cache hit.

Meltdown and Spectre (and other variants [7, 29, 46, 53]) use this recipe to leak secrets transiently accessed from memory illegally. Future attacks could use this recipe to leak secrets stored in registers, or results of malicious computations performed transiently. In fact, a recent malware called exSpectre [47] computes some of its malicious payload speculatively, to evade reverse engineering attempts, and extracts the results of these computations through the cache-channel.

While software [16, 45] and microcode patches [22] mitigate the original attacks, their adoption is hindered [64] due to high slowdown [37]. Recent hardware mitigations [31, 43] limit the slowdown by detecting and delaying only certain *unsafe* load patterns in Spectre, arising from transient access to secrets in memory; but they fail against attacks without such patterns (e.g. leaking secrets from registers or computations). Thus, there is a need to prevent *any* transient load from leaking information through cache state changes (preferably without OS/SW support unlike other proposals [6, 26]).

2.3 InvisiSpec: Redo-Based Mitigation

A recent design InvisiSpec [58] developed a way to tolerate all such attacks in hardware. As transient instructions modify the cache state to transmit secrets, InvisiSpec disallows any changes to the cache state during speculative execution. For speculatively issued loads, InvisiSpec executes an *Invisible* load that makes no changes to the cache hierarchy and only brings the data to the core, as shown in Figure 3. Once the speculation is determined as correct and the load is ready for commit at the head of the ROB, it *Redo*es the load – executing a second load to update the cache. Thus, a transient instruction squashed on a mis-speculation leaves no trace in the cache, preventing information leakage on the correct path.

While this approach prevents mis-speculated loads modifying the cache, most loads are issued speculatively and correctly speculated in

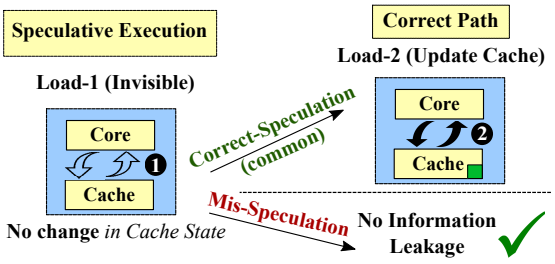


Figure 3: InvisiSpec adopts a redo-based approach to prevent mis-speculated loads from modifying the cache – speculative loads are invisible to the cache; On the correct path, the load is repeated to update the cache.

the common-case – requiring a second *Update* load for cache update. This increase in loads impacts performance, causing slowdown.

2.3.1 Performance Problem in InvisiSpec. For common-case correctly speculated loads, InvisiSpec requires the second load to be performed on the critical path at commit-time, before retiring the instruction. This is because of the need to ensure no memory consistency violations occurred in the period between the invisible load and commit, due to a modification of the data by a different core. As the invisible load does not update the ownership of the line in the directory, the core is unable to receive invalidation updates until the second load updates the cache and directory state.

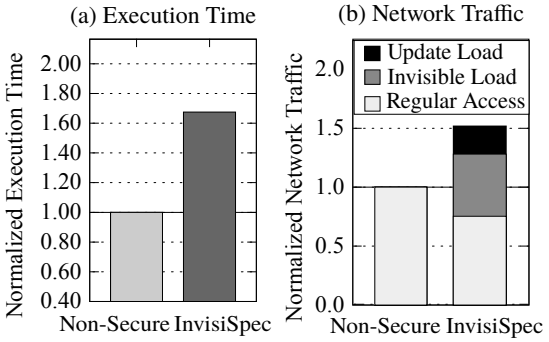


Figure 4: (a) Execution Time and (b) Network Traffic for InvisiSpec normalized to Non-Secure Baseline (initial estimates)

Figure 4 illustrates the performance and network traffic for InvisiSpec normalized to a Non-Secure baseline, using 19 SPEC-CPU2006 workloads. As per initial estimates[3], InvisiSpec suffers an average slowdown of 67.5% compared to Non-Secure, that is accompanied by a 51% increase in network traffic. Approximately half of the additional traffic is driven by the extra accesses to update the cache at commit-time (Update-Loads in Figure 4(b)) and check for potential consistency violations. As these accesses fall on the critical path, the resulting stalls cause slowdown. Additionally, close to half of the total network traffic is due to speculative loads, indicating a large majority of loads are issued speculatively.

[3]We report our results using the InvisiSpec public code-base (commit: 39cfb85) [57] as initial estimates, as the authors intimated us shortly before the camera-ready deadline that they have an updated implementation with lower overheads (refer Section 6.5).

2.4 Undo Approach: Benefits and Challenges

Given the high branch prediction accuracy in modern processors, a large fraction of the loads issued would typically be correctly speculated. Therefore, rather than adopting a *redo*-based approach that requires performing a load twice for correctly-speculated loads, an *undo*-based approach is preferable from a performance perspective.

An undo-based approach would allow all loads to modify the cache speculatively, as shown in Figure 5. In case a mis-speculation is detected, then corrective action could be activated that cleans-up the cache state changes made by the illegal transient loads to ensure that no information is leaked on the correct path. Such an undo-based approach would be beneficial for performance, as any overhead would be incurred only in the uncommon case of a mis-speculation, while the correctly speculated loads execute without any change. Moreover, some of the cleanup overhead would be hidden by the pipeline drain latency that is incurred in any case.

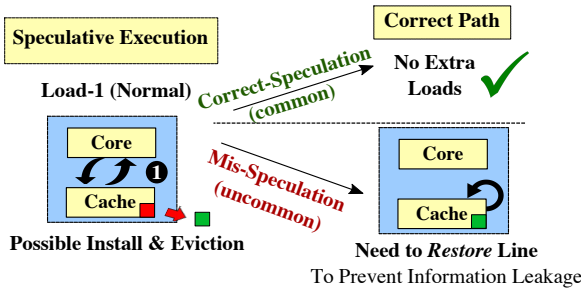


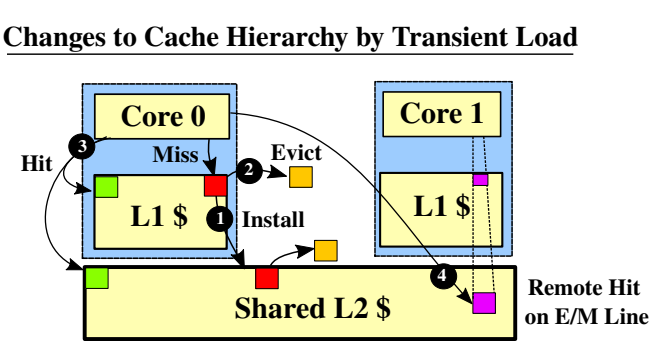
Figure 5: A low-overhead Undo-approach is viable as long as cache state cleanup on mis-speculation prevents information leakage on the correct-path.

2.4.1 Naive Invalidation - Vulnerable to Prime+Probe. A naive design to undo cache changes by transient instructions could track lines installed in each level of the cache hierarchy by such instructions and invalidate them on a mis-speculation. This prevents an adversary from using Flush+Reload attack to infer transiently installed lines.

Unfortunately, lines installed transiently can evict other lines from the same set. Leveraging this, an adversary can use Prime+Probe [34] attack – pre-load sets with its lines and observe which lines get evicted due to installs. Thus, an adversary can infer cache sets that had a transient line install. To prevent such attacks, we need to not just remove the installed line, but restore the evicted line in its place.

2.4.2 Mitigating Prime+Probe for L2/LLC. Recently, CEASER [38] and Skewed-CEASER [39] proposed randomizing last-level caches with address-encryption, to mitigate eviction-based attacks like Prime+Probe. With randomized cache-indexing, these solutions map random lines to the same set. As a cache install evicts an unrelated line, evictions stop leaking information about installed lines.

Large caches like L2/LLC or Directories [61] can adopt such randomization without any changes to OS/SW. However, randomization is challenging for performance-sensitive L1 caches. Address-encryption is not feasible as it would add 1-2 cycles of encryption latency to every access, doubling the L1-access latency, that would be detrimental to performance. Additionally, other proposals like NewCache [51] or RPCache [50] face challenges due to the preferred VIPT operation of L1 caches – they either need VIVT operation of the cache that has the problem of synonyms [9], or need PIPT



Change by Transient Load			CleanupSpec Mitigation
❶	Install on Miss	L1/L2	Remove on Squash
❷	Eviction on Miss	L1	Restore on Squash
		L2	Randomize L2 LineAddr
❸	Replacement State Update on Hit	L2	
		L1	Random Replacement
❹	Coherence Downgrade (E/M -> S)		Delay Until Correct Path

Figure 6: Overview of CleanupSpec Design. To ensure changes made by Transient Loads do not leak any information on the correct path, CleanupSpec *Removes, Restores, Randomizes* or *Delays* these changes, as appropriate.

operation that has slower access (TLB look-up is on the critical path). Given these pitfalls, L1 caches are hard to randomize, and remain vulnerable to eviction based attacks, making our Undo-approach for safe speculation challenging.

2.5 Goal of this Paper

Our goal is to enable a secure and low-overhead implementation of an undo-based approach to safe speculation, by reusing existing structures, avoiding overheads of buffering evicted lines, and without OS or software support. We develop our solution as a combination of different strategies: restoration of evictions from L1 caches, invalidation and randomization for L2/LLC, random-replacement for L1 cache to avoid state changes of replacement metadata, and delaying the state change due to coherence for speculative instructions.

3 DESIGN

Our design philosophy with CleanupSpec is to optimize the design for the common-case where loads are correctly speculated. To this end, CleanupSpec allows transient loads to speculatively access the cache and make changes as required. To enable security on a mis-speculation, we study the changes a transient load could make to the data-caches, and delay, reverse or randomize these changes.

We study the transient cache changes using a configuration shown in Figure 6, with two cores having private L1 data-caches and sharing a common L2 cache. A load can incur a miss in the L1 or L2 cache, that evicts a victim and installs a new cache line in its place, which is retained on the correct path. Furthermore, if a dirty line is evicted from L1 cache as result of the install, it would cause a write-back to L2, that could, in turn, cause an eviction from L2. A load hit can update the replacement state that can also leak information [26], as it affects victim selection and can be used to engineer evictions. Similarly, a load to line in Exclusive (E) or Modified (M) state owned by a different core would cause a coherence state downgrade for the line to Shared (S) – even this can leak information [62] on the correct path due to difference in access latencies to E/M and S lines. To prevent such changes by transient loads from leaking information, CleanupSpec needs to track, protect and reverse these changes on a mis-speculation, so that they are imperceptible to an adversary.

3.1 Overview of CleanupSpec Design

There are six main components to CleanupSpec that make the transient changes to the cache hierarchy benign.

(1) **Address Randomization for L2 Cache – Prevent leaks from L2 Evictions and Replacement Policy**: Buffering evictions from L2 cache can be expensive in terms of both storage and complexity. Moreover, tracking recursive evictions in the cache hierarchy due to writebacks causes further complications. Address randomization (e.g. CEASER [38]) randomizes the sets that spatially contiguous lines map to, making co-residents of a line in a set unpredictable. As a result, an eviction leaks no information about the address of the Install or L1-Writeback that caused it. Similarly, the replacement state of L2 becomes benign, as it can only be exploited to induce evictions of random-lines. We analyze the overheads of L2-Randomization in Section 3.2.

(2) **Removing L1 or L2 Installs from the Cache:** To prevent a transiently installed line from causing hits on the correct path after a mis-speculation, we remove the line from the levels of the cache it got installed in by issuing an invalidation to only those cache levels. We enable this by tracking which levels a load caused an install, propagating this information with the load-data through its lifetime in the L1/L2-MSHR and the Load-Queue, till it is retired (more details in Section 3.3).

(3) **Restoring L1-Evictions:** Without randomizing the L1-cache, we need to prevent evictions from leaking information. So, on a mis-speculation, while removing the installed line, we also restore the original line that was evicted. We achieve this by tracking the line address of the evicted line in the L1-MSHR on an install and propagating it with the load-data to the Load-Queue. After restoring the evicted lines, we achieve a L1-cache state such that the unsafe L1-installs and evictions never occurred (more details in Section 3.4).

(4) **Random Replacement Policy for L1:** To prevent replacement state updates on L1-hits from leaking information, we use random replacement policy for the L1 cache. We find that this has a negligible impact on the overall system performance (see Section 3.2).

(5) **Delaying Coherence Downgrades from M/E to S, till correct path:** Changes in coherence state are not only perceptible from the same core on the correct path, but also have a non-reversible impact on other cores. Thankfully, only transitions from M/E to S are perceptible due to difference in access-latency (and relatively infrequent), hence we delay them till the correct path. We describe the changes required to the coherence protocol to achieve this in Section 3.5.

(6) **Protecting lines in the window of speculation:** In the small window of time between a speculative line install and a load-squash, an access from another thread/core that has a hit on this speculatively installed line can leak information. Our design can detect such hits and service them with cache-miss latency (using dummy requests) to prevent information leakage. We describe this mechanism in Section 3.6.

3.2 Randomizing L2 Lines & L1 Replacement

We use address randomization for L2 cache, using an encrypted line address to index the cache (like CEASER [38]). As the encryptor only adds 2-cycles to access latency, this incurs minimal overhead. While this prevents L2-evictions from leaking information, we observe it also prevents the L2 replacement state from being exploited. Prior work [26] exploited transient replacement state updates to influence victim selection and engineer evictions. However, as evictions are benign with randomized caches, even replacement state updates cannot be exploited and intelligent replacement policies can be freely used for the L2 cache. As randomizing L1 data caches is challenging, we use a random-replacement policy for L1.[4] Thus, replacement state updates for both L1 and L2 caches leak no information.

Table 1 quantifies the performance impact of randomization, comparing L2-Randomization and Random Replacement for L1-D cache versus a baseline non-secure design, using 19 SPEC CPU2006 benchmarks. While random replacement for L1 causes a minor increase in L1 miss-rate, the extra misses are serviced from the L2 with minimal overhead. While L2-randomization further incurs a minor increase in L2 misses, together they still have negligible slowdown ($<1\%$).

Table 1: Impact of Randomization for L2 (2MB) and Random Replacement for L1-DCache (64KB) vs LRU-Baseline.

Configuration	Slowdown
L1-Rand Replacement	0.1%
L2-Randomization	0.4%
Both Together	0.8%

3.3 Removing L1 and L2 Installs

On a transient cache miss, data is speculatively installed in the cache. On a mis-speculation, to remove any trace of such installs, the line is removed by issuing an invalidation for it in those levels. To identify which levels of the cache had an install, we track the side-effects of every load on L1 and L2 cache in a Side-Effect Entry (SEFE, pronounced *safe*), by augmenting entries in the L1/L2-MSHR and Load Queue (LQ). As shown in Figure 7, the SEFE includes 1-bit (*isSpec*) indicating a speculative load and 1-bit per cache level *(L1-Fill*, *L2-Fill*) to indicate that the load caused an install at this cache level. The fields *L1-Fill* and *L2-Fill* are updated in L1/L2-MSHR entry when the load causes cache changes during miss-handling, and the *LoadID* tracks the order in which these changes occur. Whereas, the shaded SEFE fields – *EpochID* (that uniquely identifies the phase of execution between 2 cleanups) and *isSpec*, are updated by the Load/Store unit when the load is issued. The SEFE is returned with the load-data to the core and retained in LQ, until the load is retired.

On a load squash, the LQ-Entry SEFE is referred to decide whether to send invalidation message to L1/L2 based on L1-Fill/L2-Fill values, or if it can be skipped – if the load was not issued or if there was a L1-Hit (L1-Fill=0 and L2-Fill=0).

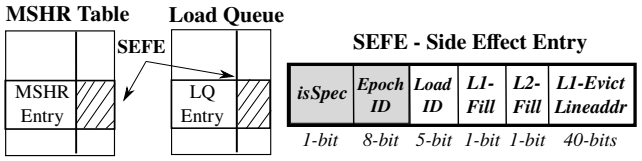


Figure 7: Side-Effect Entry (SEFE) tracks side effects of a load, in Load Queue and L1/L2-MSHR. Shaded SEFE fields are filled by Load/Store unit, unshaded by L1/L2.

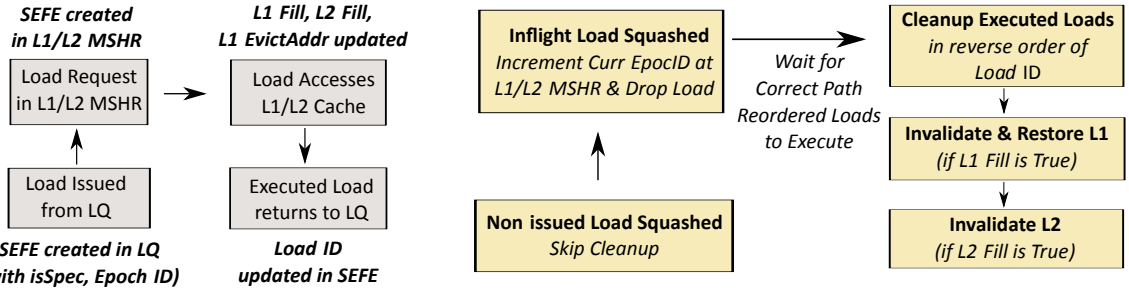
If a load is yet to return to the LQ, and is in flight, then a cleanup request is sent to L1/L2-MSHR to squash pending loads and increment the current active EpochID. The core is stalled only till an acknowledgment is received from L1/L2-MSHR. Subsequently, loads are safely issued with an incremented EpochID, which receive a new MHSR entry and issue a fresh memory request safely. Squashed MSHR entries (with an outdated EpochID) waiting for in-flight memory requests are preserved until the data is returned from memory, upon which both the data and the MSHR entry are safely dropped (any cache changes like install and victim replacement are made only when a load returns and is for the current EpochID). EpochID has sufficient bits so that the incremented value does not match that of an inflight load.

3.4 Restoring Lines Evicted due to L1 Installs

Squashed loads that install a line in the L1 cache also leak information through evictions. To prevent this, in addition to invalidating the line, we restore the original line evicted from the L1 cache. We enable this by recording the line address of the evicted victim in the L1-MSHR SEFE (*L1-Evict Lineaddr* in Figure 7). When the load is squashed, the evicted line is restored from the L2 by issuing a normal demand access for this address to L2, and installing it in place of the invalidated line. To ensure no illegal changes are introduced due to the restoration, we need to consider the following cases:

Avoiding Recursive Squash During Cleanup: Before operations like invalidate and restore begin, we wait for the retirement of all in-flight loads before the squashed loads in program order (potentially correct-path loads), reordered due to out-of-order execution. Thus, any squash at an earlier point in program order is detected before a cleanup begins. Moreover, no new loads are issued while the cleanup requests are being issued and an acknowledgement is yet to be received, avoiding restoration of an incorrect transient state.

[4] Using a less intelligent Random-Replacement Policy with some performance cost, is better than paying the cost of security vulnerability.



(a) SEFE population for Transient Loads **(b) Using SEFE to execute Cleanup operations**

Figure 8: Flowchart for Two Phases of Execution. (a) Regular Operation where Side-Effect Entries (SEFE) are updated. (b) Cleanup on Squash – state in SEFE is used for determining and executing Cleanup operations.

Squashing Re-ordered Loads: We ensure cleanup operations for loads are issued in the reverse of the order in which they executed, with cleanup of independent sets proceeding in parallel. Moreover, we squash in flight loads before the invalidation and restoration of executed loads. Ensuring that cleanup operations are correctly ordered is possible using the *Load-ID* field in the SEFE that tracks the order of loads returning from the L1 cache. As a result, removal/restoration effectively reverses time for the cache and no illegal state is retained.

Squashing Loads Re-ordered with Correct-Path Loads: Changes made to the cache by older non-squashed loads that execute after squashed loads due to re-ordering, need to be preserved and not reversed, as they would have occurred irrespective of the squashed load. Such execution patterns can be detected by the LQ-SEFE, and corresponding invalidations / restorations are skipped.

3.5 Delaying Coherence State Downgrades

Prior work [62] showed that the coherence state of a line can be used to leak information. This is due to access latency difference between E/M and S lines: accesses to S lines in L2 can be satisfied directly from the L2, whereas E/M lines typically require servicing the line from a remote-L1. As allowing the change from E/M -> S transiently can be a vulnerability, CleanupSpec delays such changes until the correct path. Note that transient changes like I -> E or adding S-sharers (even in remote directories), that do not affect the state of a line cached by a remote core, are allowed and reversed.

Table 2: Coherence state transitions in a remote core, caused due to actions initiated by transient instruction.

Old State	New State	Transient Instruction	Mitigation
M,E	S	Load Shared Data	Retry on correct-path
M,E,S	I	`clflush`	Delay till correct-path

A transient load to shared data can cause change from E/M state to S in a remote cache. To selectively delay such transition-causing loads till they are unsquashable, we add a new transaction called GetS-Safe that is used by default instead of GetS. GetS-Safe only succeeds in getting the data if a E/M->S transition is not required. In case it fails, the core delays the load till it is unsquashable, and repeats the load with GetS only once it is safe to force the transition.

We expect these transitions to be infrequent as they usually occur on transfers of lock ownership in multi-threaded workloads. Such transfers are relatively uncommon, compared to accesses for actual work. We characterized this in 23 multi-threaded PARSEC [4] and SPLASH-2 [55] benchmarks using *simlarge* data-set on a 4-core system with Snipersim [8]. As shown in Figure 9, loads to Remote-E/M lines make up just 2.4% of total loads on average. Thus, delaying such loads would have negligible slowdown, with 96.8% of loads to local lines in M/E/S state or Remote-S lines proceeding as is.

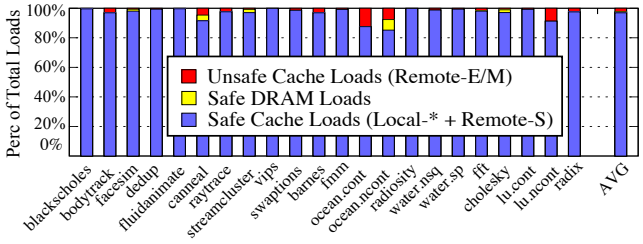


Figure 9: Breakup of Loads based on State of Line for multi-threaded PARSEC and SPLASH2 benchmarks.

A transiently executed `clflush` can also be used to leak information, as it can invalidate all copies of a cacheline, even in adversary-controlled remote caches. We delay such instructions, until they are unsquashable on the correct path. As such instructions are ordered with respect to stores, and normally used after stores, they typically execute on the correct path anyway and see no added delays.

3.6 Protecting Installs in Speculation Window

In the small window of time between the speculative install of a line into L1/L2 cache, and removal of the load from the pipeline through a squash, a cross-core or cross-thread SMT adversary might access the line, hoping to infer the speculative install with a cache hit. To prevent such information leakage within this speculation window, we ensure the first access to a line within this window from a different thread/core than the one installing it, incurs a dummy cache-miss. In this case, the cache issues an explicit request to the backing store (L2/main-memory), waits till it completes and only then returns the data, thus leaking no information.

We characterized the speculation window for SPEC workloads and observed that >98% loads commit/squash within 200 cycles

of being issued, with >99.5% finishing within 600 cycles. Within this speculation window, detecting cross-thread L1-cache accesses for a line is possible with SEFE metadata in LQ/L1-MSHR. To detect cross-core accesses to L2-cache within this window, we keep the SEFE in L2-MSHR active for 200 cycles after a line install (MSHR entries are accordingly provisioned). For loads that continue to be speculated beyond 200 cycles, the core sends a message to the cache to extend the window for such a load every 200 cycles. These messages constitute <2% cache traffic.

Once detected, these accesses from a different thread to speculatively installed data can be serviced as cache misses. This protection is only activated in uncommon scenarios in benign programs and incurs negligible slowdown – repeated cross-program accesses to recently installed lines within the small window are unlikely. In multi-threaded programs, such behavior triggers coherence-downgrades that we showed to be uncommon (<3% loads in Figure 9).

An adversary may also attempt to infer speculative evictions within this window. For cross-core adversary, our L2-cache randomization prevents L2-evictions from leaking information (Section 3.2). For SMT adversary, a mitigation like way-partitioned L1 cache (e.g. NoMo [14]), that is anyway required for preventing non-speculative L1 cache side-channel attacks, also prevents speculative L1 evictions being observed. Our evaluations show that partitioning L1-ways between 2 threads incurs <2% slowdown (concurring with [14]).

4 SECURITY ANALYSIS

We study three adversary models: *SameThread* - where a transient cache access is initiated and observed from the same thread, *CrossCore* - where the initiator and observer are on different cores and *SMT* - where they are in different threads running simultaneously on the same core. In each scenario, we assume the transient change is initiated with a speculative read and observed by the adversary using: speculative reads, or non-speculative reads or writes. We assume the transient change (or lack thereof) is inferred based on latency difference of a cache hit or miss, or a latency difference on a coherence upgrade or downgrade operation on cached lines.

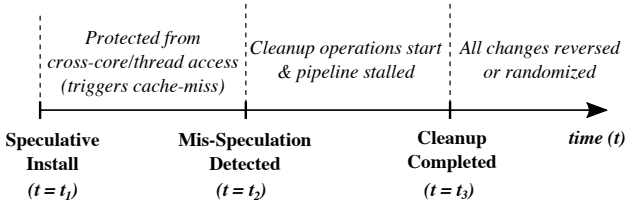


Figure 10: High level idea of security with CleanupSpec.

For security, we argue that CleanupSpec satisfies three properties:

(a) **Before detection of mis-speculation** *(t1 to t2)*, transient cache changes are protected and not observable.

(b) **During cleanup operations** *(t2 to t3)*, the process of restoration of cache state does not practically leak information.

(c) **After cleanup** *(t3 onwards)*, all transient changes are restored or made imperceptible, so no changes are observable later.

(a) Security until detection of mis-speculation: CrossCore and SMT adversaries are prevented from observing a hit on a line transiently installed in L1 or L2 cache by the SEFE entries in MSHR, which trigger a dummy cache miss in such scenarios (refer Section 3.6). Similarly, a CrossCore adversary cannot infer any information from L2 evictions due to L2-randomization. SMT adversaries are prevented from observing transient L1-evictions, with L1-cache way partitioning across threads (required anyway for preventing non-speculative L1 cache side-channel attacks under SMT). These hit/miss attacks are impractical with SameThread adversary, as observing a hit or miss on a transiently installed/evicted L1-cacheline requires hoisting wrong-path loads over older correct-path loads. But, measuring their timing with any fidelity requires using timers with serializing instructions (like `cpuid` or memory fences) [35] that would prevent any such hoisting, precluding this attack.

To leak information, CrossCore adversary may also initiate coherence upgrades (using writes) or downgrades (using non-speculative reads) on transiently installed/evicted addresses. To prevent latency differences in such scenarios, CleanupSpec imposes two constraints: (1) writes need to be constant-time and equal to worst-case of downgrading S lines in every core. (2) reads that trigger downgrades on a transiently installed line in a remote L1-cache, need to be serviced directly from the L2-cache if the transient install did not cause a L2-Fill (otherwise from the main-memory) and not from the remote-L1 cache. Note that this is possible because the L2 copy never becomes stale transiently as RFOs are issued non-speculatively to prevent Spectre-Prime[44] attacks. If the adversary attempts downgrade on a line transiently evicted from remote L1 cache, a dummy delay can be added to the adversary's read, to emulate service from the remote cache. These constraints can prevent latency differences for coherence operations while incurring negligible performance loss – as write-latency is not on the critical path of program execution, and read-caused-downgrades are infrequent in typical programs (as shown in Figure 9). Note that CleanupSpec prevents speculative initiation of a coherence downgrade (E->S), as it is naively observable by a CrossCore adversary (as described in Section 3.5).

(b) Security during cleanup operations: On detection of mis-speculation, all non-squashed loads that are inflight are completed before cleanup operations begin and no new instructions are fetched until cleanup finishes. This prevents SameThread adversary from observing any contention during cleanup operations. While this increases the stall-time after a mis-speculation, this causes the actual time taken for cleanup operations to make up only a small portion of the stall (as shown in Figure 14) and hence hard to exploit for the SameThread adversary. This is also because the corresponding restoration cache accesses are pipelined and serviced from the inclusive L2 (which is the common configuration in modern processors like Intel Skylake-X or AMD Ryzen-2). Moreover, the adversary cannot transiently evict the line to be restored from L2-cache to increase the time for cleanup operations, as that would need thousands of accesses due to the randomized-mapping (that is impractical to perform transiently). Future work can explore making the cleanup-operations incur a constant-time stall to make this theoretically impossible to exploit. For CrossCore or SMT adversary, ensuring security during cleanup operations is along the same lines as Section 4(a).

(c) Security after cleanup on mis-speculation: After cleanup, all lines are invalidated from the cache levels where they were transiently installed into, and the evicted lines are restored for L1 and randomized for L2. So no new cache hits or misses are observable due to the transient changes, in any adversary model. Moreover, attempting to even infer if a single L2 eviction occurred is impractical, as filling the randomized L2 deterministically (required to initialize this attack) is not feasible – the randomized and changing mapping would result in self-evictions during the attempt to fill the cache. Finally, the coherence state of the restored/invalidated lines is updated based on presence in other private caches, such that it is independent of the transient access by the victim, so that it leaks no information.

5 EXPERIMENTAL METHODOLOGY

5.1 Simulation Framework

We evaluate our design using Gem5 [5], a cycle accurate simulator that models the out-of-order processor including the wrong-path execution effects in the core and the caches. We simulate a single-core system in System-call Emulation (SE) mode in Gem5. For the multi-core characterization in Section 3.5, we used Sniper [8] that simulates multi-threaded binaries using a Pin front-end (due to errors with multi-threaded workloads on InvisiSpec's infrastructure). For InvisiSpec evaluation, we use their public code-base (commit: 39cfb85 [57]) and evaluate the *InvisiSpec-Future* mode that treats all speculative loads as "unsafe" (same as our threat model).

5.2 Workloads

We use 19 workloads from SPEC-CPU2006 [20] with the *reference* data-set. We forward the execution by 10 billion instructions and simulate 500 million instructions. Table 3 shows the important characteristics of these workloads. For the multi-threaded workload characterization in Section 3.5, we used 23 workloads from PARSEC [4] and SPLASH2 [55] benchmarks using the *sim-large* data-set, and collect statistics for the entire region of interest.

Table 3: Workload Characteristics

Workload	Branch Mis-Prediction Rate	L1-Data Cache Miss Rate
astar	12.4%	1.8%
gobmk	11.9%	1.0%
sjeng	11.3%	0.2%
bzip2	9.7%	2.0%
perl	7.7%	0.5%
povray	7.5%	0.2%
gromacs	6.8%	1.1%
h264	5.4%	0.5%
namd	4.2%	0.3%
sphinx3	4.1%	4.0%
wrf	2.2%	0.5%
hmmer	1.9%	0.2%
mcf	1.6%	2.5%
soplex	1.5%	5.9%
gcc	1.3%	0.1%
lbm	0.3%	11.0%
cactus	0.1%	0.9%
milc	0.0%	4.6%
libq	0.0%	10.4%

5.3 Configuration

We evaluate a system configured as in Table 4. A minor difference compared to prior work InvisiSpec, is that we use a Close-Page policy in the memory controller to prevent information leakage through DRAM row-buffer hits and misses. Additionally, we increase the L2 cache access latency by 2 cycles (from 8 cycles to 10 cycles) to incorporate the overheads of address randomization [38]. We evaluate a 2-level inclusive cache-hierarchy for simplicity, but our design philosophy is equally applicable to other configurations.

Table 4: System configuration (similar to InvisiSpec [58])

Architecture	1-core OOO, no SMT, 2GHz
Core	ROB-192 Entry, LQ / SQ-32 Entry Tournament Branch-Pred, BTB-4096 entry, RAS-16 entry
L1 I-Cache	32KB, 4-Way, 64B line 1-cycle RT Latency
L1 D-Cache	64KB, 8-Way, 64B line 1-cycle RT Latency
Shared L2-Cache (inclusive)	2MB/core, 16-Way, 64B line, 10-cycle RT (incl. 2-cycle addr-encryption latency)
Coherence	Directory-based MESI
DRAM	50ns RT after L2

6 RESULTS

6.1 Proof-of-concept Defense

We test our Gem5-based design with Spectre Variant-1 PoC [2], that exploits branch prediction in victim code – `if(x < array1_bound) {array2[array1[x] * 512]}`. The attack attempts to bypass the array bounds-check by using benign values of `x` to prime the branch-predictor. Subsequently, using a `malicious_x` causes a speculative access to a secret memory location `array1[malicious_x]`, which is used to generate the array-index for an access to `array2`. On the correct path, the secret is inferred by testing which line of the `array2` gets a low-latency cache hit. Figure 11 shows the access latency for different `array2` indices during this secret-inference phase, averaged across 100 attack iterations.

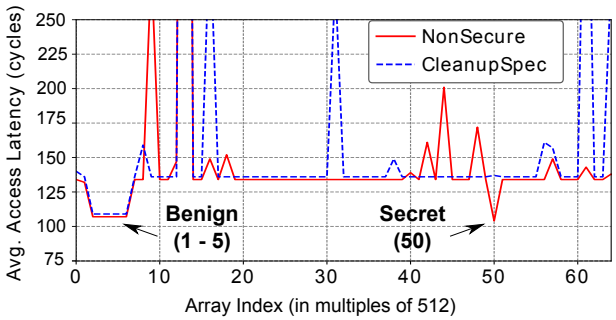


Figure 11: Average array access time for secret-inference phase of Spectre Variant-1. CleanupSpec has no latency difference for the secret index (50) installed on the wrong-path, while having identical behavior as non-secure for (benign) lines installed on the correct-path.

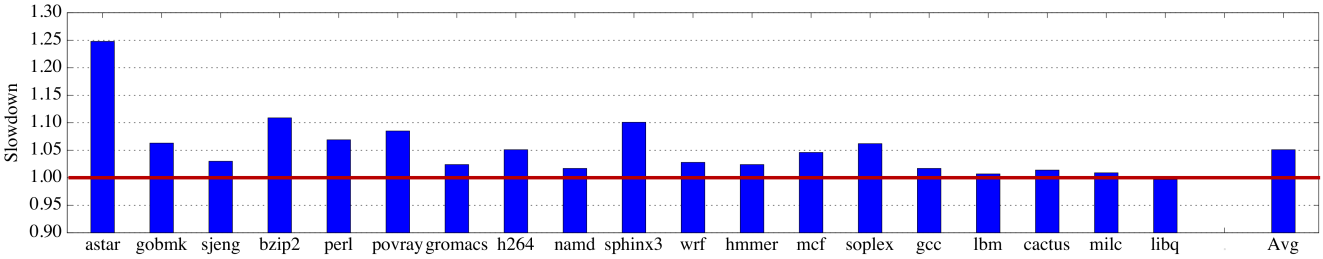


Figure 12: Execution time of CleanupSpec normalized to Non-Secure baseline. On average, CleanupSpec causes a slowdown of 5.1%.

In the baseline (NonSecure), the attacker perceives lower latency for `array2` entries installed during the benign training phase (corresponding to `array1[x]` values 1–5). For these locations, the observed latency average lies between the raw cache and memory access latency, because the attack fails to detect cache hits in some attack iterations. A similar low latency is also observed for `array2[50 * 512]`, that leaks the secret value of 50 stored in `array1[malicious_x]`. On the other hand, CleanupSpec only has low-latency for the benign accesses (installed on the correct path), and has no latency difference for the secret index (accessed on the wrong-path). This is because, when a mis-speculation is detected after the wrong-path access, the cleanup operations restore the cache state and make the wrong-path accesses imperceptible. As a result, CleanupSpec prevents any leakage of information.

6.2 Performance

Figure 12 shows the execution time of CleanupSpec, normalized to that of the Non-Secure baseline. The bar labeled *Avg* denotes the geometric mean over all the 19 workloads. On average, CleanupSpec incurs a slowdown of 5.1%. This is because CleanupSpec allows the loads to speculatively modify the cache, and incurs no additional overheads for correctly speculated loads. As most workloads in Table 3 have branch mis-prediction rates of <10% and a majority of the loads are correctly speculated, CleanupSpec only incurs minimal slowdown. In Figure 12, we order the workloads in terms of branch mis-prediction rates (from highest to lowest) and observe that the workloads on the left-hand side with higher mis-prediction rates have the highest slowdowns (e.g. *astar (24%), bzip2 (11%)*). Whereas, workloads on the right side with the lowest mis-predictions have negligible slowdown (e.g *lbm, milc, libq*).

Additionally, CleanupSpec incurs low overheads for workloads with high L1 data-cache hit rates. This is because CleanupSpec issues Restore or Invalidate requests only for squashed loads that install a *new* line on incurring a cache miss. The randomization-based approach (L1 rand replacement and L2 cache randomization) of CleanupSpec makes the hits benign, so no overhead is incurred for them. As the L1 data cache hit-rate is typically high (>95% for our workloads as shown in Table 3), many of the workloads on the left side of Figure 12 incur limited slowdowns (e.g. *gobmk*, *sjeng*), and the only workloads on the right of Figure 12 with slowdown (e.g. *sphinx3 (10%), soplex (7%))* have higher data cache miss-rates.

Thus, CleanupSpec provides low overheads in the common-case operation (high cache hit rate and accurate branch prediction) for most workloads. In the next section, we analyze the time required for the cleanup operations, to better understand the root cause of the overheads in the workloads with the maximum slowdown.

6.3 Main Cause of Slowdown - Cleanup Stalls

The slowdown in CleanupSpec is due to a core stall (no new instructions fetched), while a cleanup is in progress on a mis-prediction. This stall time depends on the frequency of squashes (Figure 13) and the stall-time per squash (Figure 14). Note that it is necessary to first wait for inflight correct-path loads to complete, before starting cleanup operations (invalidation and restoration). This prevents interference of cleanup operations with inflight correct-path loads, that could leak information about locations undergoing cleanup, and even prevents nested mis-predictions. On average, workloads have 20 squashes per 1000 instructions, with most of the stall-time per squash (20 out of 25 cycles on avg.) spent waiting for inflight correct-path loads to retire. Only a small fraction (5 cycles) is needed for invalidation and restoration operations, on average.

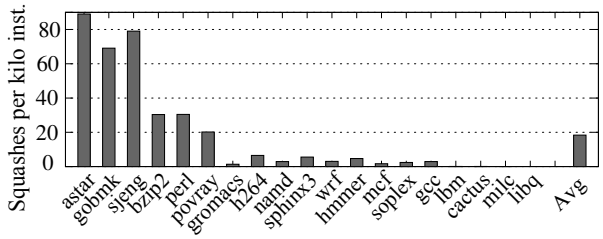


Figure 13: Squash frequency (per 1000 instructions). As squash frequency decreases (from left to right), the slowdown due to CleanupSpec also typically decreases.

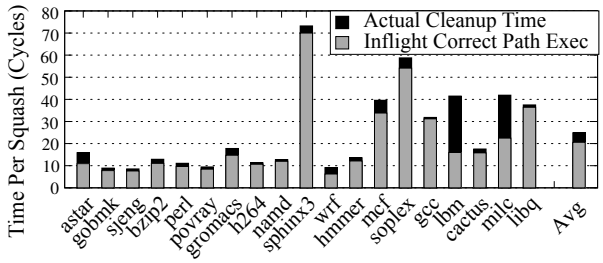


Figure 14: Stall time due to cleanup, per squash. Large fraction of the time is spent waiting for inflight correct-path instructions to complete, on every squash.

Workloads on the left (e.g. *astar*) have a short stall time per squash, but a high frequency of squashes due to high branch mis-prediction rates, and incur higher overheads. Workloads on the right have infrequent squashes, but those with high L1 data cache miss-rates (e.g. *mcf*, *sphinx3*, *soplex*) still have high stall time per squash while waiting for the correct path loads to execute, and consequently incur modest slowdowns (5%–10%). Outliers like *lbm* and *milc* need 20-25 cycles per squash to execute cleanup operations as they have a larger number of loads per squash that need cleanup (see Table 5), but they have no slowdown as squashes are uncommon.

Table 5: Cleanup statistics – Squash per kilo instruction (PKI), Loads/Squash, State of the load when squashed – Not issued (NI), L1-Hit (L1H), L2-Hit (L2H) or L2-Miss (L2M). Cleanup is needed only for Squashed Loads that are L2H or L2M.

Workload	Squash PKI	Loads/ Squash	Squashed Loads (%)			
			NI	L1H	L2H	L2M
astar	89.02	11.20	40	57	1.72	0.36
gobmk	69.10	4.27	52	45	0.57	0.43
sjeng	79.06	4.09	49	49	0.33	0.23
bzip2	30.37	8.01	48	50	1.00	0.11
perl	30.44	4.09	51	46	0.97	0.52
povray	20.20	5.71	50	48	0.13	0.07
gromacs	1.38	8.36	38	59	1.30	0.23
h264	6.53	6.46	57	41	0.36	0.18
namd	2.92	9.77	28	71	0.29	0.05
sphinx3	5.56	8.33	45	51	0.86	0.43
wrf	3.07	4.64	30	59	0.31	0.71
hmmer	4.68	15.09	41	58	0.33	0.07
mcf	1.68	17.51	68	28	0.22	0.87
soplex	2.42	11.49	29	67	0.57	0.75
gcc	2.90	4.59	60	38	0.16	0.12
lbm	0.08	24.51	52	39	0.36	3.63
cactus	0.10	13.26	37	60	0.36	0.42
milc	0.01	29.88	12	78	0.26	0.30
libq	0.00	1.37	70	23	0.00	0.36

6.4 Analysis of Loads Requiring Cleanup

Cleanup operations are only required for loads that were issued and had an L1-Miss. However, close to 50% of these loads on average are still in flight when a cleanup request is issued to the cache hierarchy, as shown in Figure 15. For such loads, it is sufficient to drop the pending request without any invalidation or restoration, as any changes to the cache like install or eviction of victim are only done when the request returns. We observe such low-overhead cleanups commonly for L2-misses, where branch mis-prediction is often detected before a memory request issues or before it completes.

6.5 Comparison with InvisiSpec

Table 6 compares the performance of CleanupSpec with InvisiSpec, all normalized to a non-secure baseline. Our initial estimates using the public InvisiSpec code-base showed that it incurs an average slowdown of 67.5% across 19 SPEC benchmarks we used (close

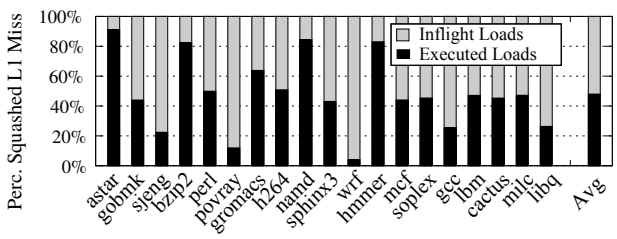


Figure 15: Breakup of Loads Cleaned-up (L1-Misses). For squashing Inflight Loads (50% L1-Misses), it is sufficient to drop these pending requests without invalidation or restoration.

to the slowdown reported in [58]). Subsequently, the authors of InvisiSpec intimated us through an email (M. Yan, Personal Communication, August 19, 2019) that they had an updated implementation incurring only 15% slowdown. According to the authors, the difference was due to "a simulation bug that incorrectly delayed the propagation of speculatively accessed data to dependent instructions". The initial estimate incorrectly delayed the propagation till the visibility point of the speculative load, whereas the revised results propagate the data to dependents as soon it is received by the speculative load. Nevertheless, both InvisiSpec implementations incur extra accesses to update the cache for correctly speculated loads.

In comparison, CleanupSpec only incurs 5.1% slowdown, because it incurs no extra accesses for correctly speculated loads, which make up the common-case. Even on a mis-speculation, no extra accesses are required for loads that were not issued, or loads that were issued and had L1-Hit – these make up >98% of squashed loads, as shown in Table 5. Only on squashed L1-misses, the overhead of extra cleanup accesses is incurred. Additionally, messages to extend the preservation of SEFE entries in the speculation window are only required for loads that do not commit/squash within 200 cycles (which is uncommon). This allows CleanupSpec to incur <2% extra accesses to the cache hierarchy and incur lower overheads.

Table 6: Overheads for CleanupSpec vs InvisiSpec, all normalized to a Non-Secure baseline.

Configuration	Slowdown
InvisiSpec (initial estimates close to [58])	67.5%
InvisiSpec (revised results as per its authors)	15%
CleanupSpec	5.1%

6.6 Storage Overhead

We require extra storage for the Side-Effect Entries (SEFE), associated with each LQ entry at the core-side and with each L1/L2-MSHR entry at the cache-side, that track the side-effects of the loads as they execute. Each SEFE in L2-MSHR occupies 2-bytes (3 status bits, a 5-bit Load-ID, a 8-bit Epoch-ID), while each SEFE in L1-MSHR/LQ occupies 7-bytes (with an extra 40-bit L1 evicted line address), as shown in Figure 7. For a configuration with 32 LQ and 64 L1/L2-MSHR entries per core, our design incurs a overhead of <1KB per core, scaling linearly with LQ and L1/L2-MSHR entries.

7 RELATED WORK

7.1 Types of Speculation-Based Attacks

Numerous speculation-attacks have been demonstrated that break intra-process, inter-process and trust-domain isolation in software. These differ based on the speculation-mechanism exploited: exception-based attacks (e.g. Meltdown [32], Foreshadow [46]) exploit race-condition between illegal data access and permission-check, whereas control/data speculation-based attacks (e.g. Spectre [28], SSB [21]) exploit speculation to bypass permission checks. Many variants of these attacks [7, 10, 12, 13, 27, 29, 44] have emerged over the past year. Recently, exSpectre [47] showed that even malware can exploit speculation to evade reverse engineering, by decrypting malicious payloads during transient execution. CleanupSpec mitigates these attacks by breaking the cache channel used to transmit information. We invalidate transient installs after mis-speculation, preventing flush-based attacks [18, 63] and randomize the L2/Directory and restore L1 state to prevent eviction-based attacks [17, 19, 34, 61].

7.2 Software and Microcode Based Defenses

Software mitigations prevent speculative access to secrets by unmapping them (e.g. KAISER [16]) or by disabling speculation in unsafe contexts (e.g. Retpoline [45], Memory Fences [11], Intel's microcode patches [23]). Unfortunately, many of these mitigations require rewriting SW or OS-changes [15] and are incompatible with legacy code. Recent studies [37] also show that commercially deployed SW mitigations have up to 50% slowdown. In contrast, our hardware defense has low overheads and requires no SW changes.

7.3 Hardware Based Defenses

7.3.1 Redo-based Mitigations. **InvisiSpec** [58] represents a *Redo-based* approach to safe speculation, whereby the load instruction is done twice: first time to get the value and second time to change the cache state. **Safespec** [25] is a similar mitigation for single-core systems that buffers transient changes to caches, until speculation is resolved. In contrast, we propose a *Undo-based* approach to safe speculation that incurs minimal overheads and does not require buffering data, making it more practical for adoption.

7.3.2 Delay-based Mitigations. **Context Sensitive Fencing** [43] (CSF) and **Conditional Speculation** [31] (CS) mitigate Spectre variants in hardware by delaying potentially *unsafe* loads, until speculation is resolved and they are deemed *safe*. However, these mitigations are limited to attacks exploiting control/data speculation that leak secrets stored in memory. For example, CSF uses taint-tracking to identify and delay kernel loads dependent on user-space data, that might potentially access secrets. CS uses heuristics to filter a subset of L1-cache misses that could potentially leak data, and delays them. In comparison, CleanupSpec prevents information leakage through *any* speculative load, while incurring two-third of the slowdown of CS and CSF. A contemporary work **Context** [41] leverages taint-tracking to determine the secret-dependent instructions in a program and modifies the processor to avoid speculation on such instructions, delaying their execution until they are non-speculative. CleanupSpec can be combined with CS, CSF, or Context, to selectively perform clean-up operations only when *unsafe* or *secret-dependent* loads execute, to achieve even lower slowdown.

Another contemporary work leverages **Value Prediction** [40] to continue performing computations, despite delayed cache accesses. However, such a proposal only benefits workloads with significant value locality and still incurs close to 10% slowdown on average. Other contemporary works like **NDA** [52] and **SpecShield** [3] delay any usage of speculatively accessed data until speculation resolves, resulting in slowdown upwards of 20%. In comparison, CleanupSpec incurs lesser slowdown since it allows speculative usage of data and only penalizes mis-speculated loads that are uncommon.

7.3.3 Partitioning-based Mitigations. **DAWG** [26] proposed a hardware software co-design to prevent information leakage through cache hits or misses on lines shared between a victim and spy, by way-partitioning the cache and duplicating such shared cache lines in each partition. However, this requires all software to be rewritten utilizing protection-domains, so that they may be mapped to separate cache partitions. Additionally, it hinders inter-process communication. In contrast, we provide a software transparent solution that also protects legacy applications. **MI6** [6] proposed a hardware-software co-design providing strong isolation guarantees for code inside enclaves. It leverages mechanisms like spatial isolation (set partitioned L2/LLC) and temporal isolation (flushing L1 on enclave entry/exit) to prevent speculation-based attacks using caches. However, it is limited by its requirement for a HW/SW infrastructure supporting enclaves, and faces scalability challenges due to its requirement of static LLC-set partitioning between enclaves. In comparison, our design is a scalable mitigation that allows efficient sharing of LLC (no partitioning), requires no software support, and has lesser slowdown.

7.3.4 Defenses against non-speculative side-channel attacks. Other proposals mitigate cache side channels [14, 24, 38, 39, 48, 50, 51, 54, 59, 60] in a non-speculative setting – we build on some of these to prevent speculation-attacks exploiting caches.

8 CONCLUSION

In this paper, we investigate a hardware solution for mitigating speculation-based attacks. We propose *CleanupSpec*, an undo approach to safe speculation by mitigating the cache side channel used in these attacks to transmit secret information from the transient path to the correct path. CleanupSpec allows transient loads to change cache state, but when a mis-speculation is detected, these changes are made imperceptible. As CleanupSpec incurs overheads only for wrong-path loads and only those that miss in the L1 cache, it only incurs a minimal slowdown of 5.1%. While this undo-based approach requires careful analysis of the implementation to ensure security, we believe it is worth the effort given that the gains in performance come at a minimal cost of <1 kilobyte per core storage overhead and simple logic for tracking and undoing speculative cache changes.

9 ACKNOWLEDGMENT

We thank the anonymous reviewers of ISCA-2019 and MICRO-2019 for their valuable feedback. We also extend our gratitude to Mengjia Yan and the other authors of InvisiSpec for help and guidance with their Gem5-based simulation infrastructure. We would also like to thank Sriseshan Srikanth, Vinson Young, Swamit Tannu, Sanjay Kariyappa and Poulami Das for proof-reading initial drafts of the paper and providing helpful feedback.

REFERENCES

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[2] Erik August. 2018. Spectre example code on ErikAugust Github Repository. https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6. (Accessed: March 19, 2019).

[3] Kristin Barber, Li Zhou, Anys Bacha, Yinqian Zhang, and Radu Teodorescu. 2019. Isolating Speculative Data to Prevent Transient Execution Attacks. *IEEE Computer Architecture Letters* (2019).

[4] Christian Bienia. 2011. Benchmarking Modern Multiprocessors. In *Ph.D. Thesis, Princeton University*.

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

[6] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. 2018. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. *arXiv preprint arXiv:1812.09822* (2018).

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv preprint arXiv:1811.05441* (2018).

[8] T.E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. 1–12.

[9] Michel Cekleov and Michel Dubois. 1997. Virtual-address caches. Part 1: problems and solutions in uniprocessors. *IEEE Micro* 17, 5 (1997).

[10] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution.(2018). *arXiv preprint arXiv:1802.09085* (2018).

[11] Intel Corporation. 2018. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. (Accessed: December 1, 2018).

[12] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2018. Cache timing side-channel vulnerability checking with computation tree logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM.

[13] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Analysis of Secure Caches and Timing-Based Side-Channel Attacks. *IACR Cryptology ePrint Archive* 2019 (2019), 167.

[14] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Trans. Archit. Code Optim.* 8, 4, Article 35 (Jan. 2012), 21 pages. https://doi.org/10.1145/2086696.2086714

[15] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. 2018. Shielding software from privileged side-channel attacks. In *({USENIX} Security)*.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. Springer, 161–176.

[17] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 368–379.

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

[19] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 38–55.

[20] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[21] Jann Horn. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. (Accessed: December 1, 2018).

[22] Intel. 2018. Intel Analysis of Speculative Execution Side Channels . https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf. (Accessed: December 1, 2018).

[23] Intel. 2018. Speculative Execution Side Channel Mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf. (Accessed: December 1, 2018).

[24] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. 2017. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017), 1–6.

[25] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the Design Automation Conference (DAC)*.

[26] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*.

[27] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[28] 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. 2019. Spectre attacks: exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

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

[30] Ruby Lee. 2018. Security Aware Microarchitecture Design. Keynote at the 51st 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Fukuoka, Japan.

[31] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *High Performance Computer Architecture (HPCA), 2019 IEEE International Symposium on*. IEEE.

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

[33] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).

[34] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[35] Gabriele Paoloni. 2010. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[36] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks.. In *USENIX Security Symposium*. 565–581.

[37] Phoronix. 2018. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower. https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect&num=1. (Accessed: December 1, 2018).

[38] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Dynamically Encrypted Address. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*.

[39] Moinuddin K. Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*.

[40] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 723–735.

[41] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. 2019. ConTExT: Leakage-Free Transient Execution. arXiv:arXiv:1905.09100

[42] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. Netspectre: Read arbitrary memory over network. *arXiv preprint arXiv:1807.10535* (2018).

[43] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[44] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *Proceedings of the 51st International Symposium on Microarchitecture*.

[45] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. (Accessed: December 1, 2018).

[46] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium USENIX Security 18)*. USENIX Association.

[47] Jack Wampler, Ian Martiny, and Eric Wustrow. 2019. ExSpectre: Hiding Malware in Speculative Execution. In *26th Annual Network and Distributed System Security Symposium (NDSS 2019)*.

[48] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *Design Automation Conference (DAC)*.

[49] Yao Wang and G Edward Suh. 2012. Efficient timing channel protection for on-chip networks. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*. IEEE, 142–151.

[50] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[51] Zhenghong Wang and Ruby B. Lee. 2008. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[52] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas Wenisch, and Baris Kasikc. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *International Symposium on Microarchitecture (MICRO)*.

[53] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Technical Report. Technical report.

[54] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 675–692.

[55] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH computer architecture news*, Vol. 23. ACM, 24–36.

[56] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.. In *USENIX Security symposium*. 159–173.

[57] Mengjia Yan. 2018. Invisispec 1.0. https://github.com/mjyan0720/InvisiSpec-1.0/tree/39cfb858d4b2e404282b54094f0220b8098053f6. (Accessed: December 1, 2018).

[58] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st International Symposium on Microarchitecture*.

[59] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360.

[60] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. 2016. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *International Symposium on Microarchitecture (MICRO)*.

[61] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[62] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are Coherence Protocol States Vulnerable to Information Leakage?. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 168–179.

[63] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*, Vol. 1. 22–25.

[64] Zdnet.com. 2018. After big Linux performance hit, Spectre v2 patch needs curbs. https://www.zdnet.com/article/linus-torvalds-after-big-linux-performance-hit-spectre-v2-patch-needs-curbs/. (Accessed: December 1, 2018).