SpecASan: Mitigating Transient Execution Attacks Using Speculative Address Sanitization

Saber Ganjisaffar
Computer Science and Engineering Department
University of California, Riverside
Riverside, CA, USA
sganj003@ucr.edu

Esmaeil Mohmmadian Koruyeh
Samsung Research America
Mountain View, California, USA
esmaeil.mk@samsung.com

Jason Zellmer
University of California, Riverside
Riverside, CA, USA
jzell001@ucr.edu

Hodjat Asghari Esfeden
University of California, Riverside
Riverside, California, USA
hodjat.asghari@email.ucr.edu

Chengyu Song
University of California, Riverside
Riverside, California, USA
csong@cs.ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
Riverside, California, USA
naelag@ucr.edu

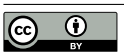
Abstract

Transient execution attacks (TEAs), such as Spectre and Meltdown, exploit speculative execution to leak sensitive data through residual microarchitectural state. Traditional defenses often incur high performance and hardware costs by delaying speculative execution or requiring additional shadow structures and dynamic information flow tracking. In contrast, our approach models these attacks as violations of software-defined security contracts and enforces these contracts in hardware using existing features. We introduce *Speculative Address Sanitization (SpecASan)*, which leverages ARM's Memory Tagging Extension (MTE) to extend memory safety protection from the committed path to the speculative path. When a speculative access does not pass the MTE tag comparison, this access is delayed until speculation resolves. This ensures that only validated accesses affect the microarchitectural state while preserving the performance benefits of speculation. When combined with Control-Flow Integrity (CFI) enforcement mechanisms, already supported by some hardware implementations, our evaluation shows that SpecASan effectively mitigates a broad class of transient execution attacks, including Spectre and Microarchitectural Data Sampling (MDS). Furthermore, SpecASan achieves this with low performance overhead and minimal hardware complexity, highlighting its practicality and efficiency.

Keywords

Speculative Execution Attacks, Memory Safety, Address Sanitizer.

ACM Reference Format:
Saber Ganjisaffar, Esmaeil Mohmmadian Koruyeh, Jason Zellmer, Hodjat Asghari Esfeden, Chengyu Song, and Nael Abu-Ghazaleh. 2025. SpecASan: Mitigating Transient Execution Attacks Using Speculative Address Sanitization. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731119



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731119

1 Introduction

Speculative execution has been an essential technique for improving the performance of processors, used in most modern high-performance processor architectures. In recent years, speculative execution has been demonstrated to be vulnerable to *transient execution attacks (TEAs)* [22]. These attacks exploit misspeculation to access sensitive data across permission boundaries. While the incorrectly speculated execution paths are ultimately squashed, preventing the leakage of sensitive data through architectural states, residual traces often persist in hardware components such as caches and microarchitectural buffers. These residual states can then be exploited via side-channel techniques to extract the sensitive data accessed during speculation. *Spectre* [18, 24, 35, 42–44, 52, 69], *Meltdown* [49, 74, 79, 82] and Microarchitectural Data Sampling (MDS) [21, 68, 80] are well-known examples of TEAs.

To address the threat posed by TEAs, researchers have proposed various mitigation strategies to protect sensitive data from leakage through microarchitectural side-channels. Many of these approaches are effective, preventing attackers from retrieving sensitive data from observable changes to the microarchitecture states during speculation. However, the mitigations often come with significant hardware complexity and incur substantial performance overhead. They typically rely on expensive techniques such as isolating microarchitectural components [20, 41, 58, 75], replicating them with shadow structures [11, 12, 38, 65, 85], rolling back microarchitectural state changes [61, 64], or employing dynamic information-flow tracking [25, 50, 81, 89]. These overheads are further compounded by the need for software stack modifications [30, 67] or changes to microcode and microoperations [50, 75, 89]. These factors contribute to high deployment costs and pose challenges to achieving widespread adoption [63].

In this work, we propose a different approach to mitigating TEAs. Our key observation is that TEAs are inherently more powerful than traditional side-channel attacks due to their ability to bypass permission boundaries during speculative execution. This allows attackers to access sensitive data that would otherwise remain inaccessible along the commit paths. Therefore, by enforcing permission boundaries during speculative execution, we can prevent these attacks from accessing sensitive data in the first place. Realizing this

mitigation strategy faces two main challenges: (1) what kind of permission boundaries should be enforced, and (2) how to enforce the boundaries efficiently.

We make the observation that the majority of TEAs violate memory safety properties as a core part of their attack procedure to access sensitive data. For instance, the *Spectre-v1* attack (discussed in Section §2.1) manipulates branch predictors to bypass array bounds checks [43]. Similarly, MDS attacks [21, 68, 80] inadvertently forward sensitive data from microarchitectural buffers to unauthorized load instructions across memory boundaries. This observation allows us to classify these exploits as "**speculative memory safety violations**," where speculative execution bypasses traditional memory safety mechanisms such as bounds checks (against out-of-bound memory access), and reference counting (against use-after-free). Therefore, by enforcing these memory safety properties at the hardware level during speculative execution, we can mitigate the sensitive data access phase of many TEAs.

Based on the aforementioned key observations, we present ***Speculative Address Sanitization (SpecASan)***, a novel mitigation mechanism against TEAs. SpecASan overcomes two key challenges to supporting speculative memory safety: (1) translating software-level memory safety rules into enforceable hardware mechanisms, and (2) effectively applying these mechanisms during speculative execution, where conventional memory safety mechanisms are often insufficient. To address the first challenge, SpecASan leverages recent advancements in hardware-accelerated memory safety technologies. In this paper, we present a design based on *ARM Memory Tagging Extension (MTE)*, to retrieve and enforce software-level memory safety requirements through a lightweight memory coloring mechanism [71] (detailed in Section §2.3). ARM MTE is selected due to its well-established yet continuously evolving software toolchain support [6, 7, 9, 10, 28] and its integration into widely used consumer devices, such as the recent Google Pixel and Samsung Galaxy smartphones [53, 62]. However, the underlying principle of extending memory safety protections to speculative execution is adaptable and can be applied in conjunction with alternative memory safety enforcement techniques.

To achieve a balance between performance and security when enforcing memory safety requirements during speculative execution, SpecASan incorporates tag checking information into various microarchitectural buffers and caches, which demands careful design and integration. Moreover, it employs a selective delay mechanism that *only delays unsafe speculative tagged memory accesses* (i.e., when tag mismatches), which are infrequent, until speculation is resolved; while allowing safe, untagged, or independent memory accesses to proceed without delay. This approach ensures that data is propagated to microarchitectural buffers and the cache hierarchy only after the validity of access is confirmed, thereby mitigating a wide range of speculative execution attacks that exploit residual data from unauthorized speculative memory accesses.

This approach offers three main advantages. First, we benefit from extending strong safety guarantees available on the committed execution path, to those instructions executed speculatively, effectively closing a critical vulnerability gap. Second, SpecASan introduces negligible performance overhead (i.e., pipeline stalls) during normal operation when there is no memory safety violation;

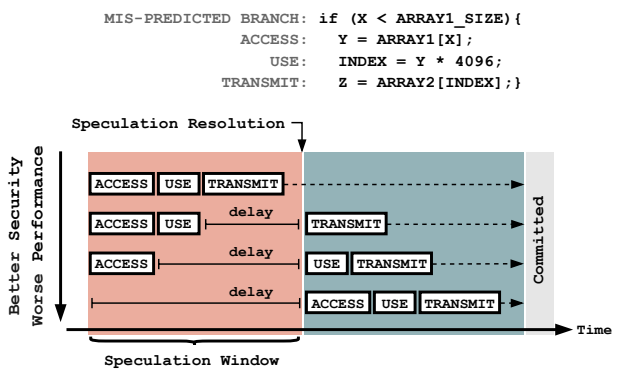


Figure 1: Comparison of various TEA defense classes, using a Spectre-v1 gadget as the reference code.

such violations are rare during normal execution. Finally, the solution achieves low hardware complexity by leveraging and minimally extending existing hardware support for memory safety, ensuring its practicality, ease of integration, and feasibility for adoption in real-world hardware implementations.

Security analysis (Section §4) demonstrates that SpecASan is effective in mitigating a wide range of TEAs, and can be extended to protect against other TEA variants, such as MDS [21, 68, 80] and Speculative Interference [17] attacks. Our performance evaluation (Section §5) utilizing detailed gem5 simulations across standard benchmarks such as SPEC CPU2017 and PARSEC, reveals that SpecASan incurs a minimal performance overhead of 1.8% and 2.5% on single-threaded and multi-threaded applications, respectively. Notably, this overhead remains consistently lower compared to two well-known hardware mitigation mechanisms, GhostMinion [11] and Speculative Taint Tracking (STT) [89], which leverage shadow structures and dynamic information flow tracking, respectively.

In summary, the key contributions of this paper are:

- We introduce SpecASan, a mitigation technique for TEAs that leverages software-level memory safety techniques to prevent illegal data accesses during speculative execution.
- We present a potential realization of SpecASan based on ARM MTE. Integrating with existing hardware support for memory safety results in getting this protection at a minimal overhead, significantly lower than other existing solutions.
- We show that SpecASan effectively mitigates a wide range of TEAs, addressing the majority of known variants, and, when combined with a *Control-Flow Integrity (CFI)* enforcement mechanism, also prevents control-flow attacks.
- Performance evaluation shows that SpecASan incurs minimal overhead (even when combined with CFI mechanisms), providing a performant, secure, and low-overhead solution to TEAs.

2 Background

In this section, we review important background related to TEAs as well as proposed mitigations. We also introduce ARM MTE-based memory safety enforcement, which we extend to implement our solution.

2.1 Transient Execution Attacks (TEAs)

Speculative execution is one of the most important techniques used by modern processors to achieve high performance. Rather than stalling the processor pipeline when unresolved dependencies (e.g., a branch target) are encountered, speculative execution keeps processors' pipeline busy by predicting these dependencies and speculatively executing the dependent instructions. Once the dependency is resolved, execution seamlessly proceeds if the predictions are correct. If the predictions prove incorrect, the speculative instructions are squashed, restoring the correct state of execution prior to the start of the speculation.

Since the program architectural state is restored once misspeculation is detected, common wisdom held that allowing arbitrary speculation posed no significant risk and could be leveraged to maximize performance. For example, speculative execution could bypass or delay operations like permission checks. However, the *Spectre* [43], *Meltdown* [49], and *Microarchitectural Data Sampling (MDS)* [21, 68, 80] attacks demonstrated that data accessed during speculative execution can be disclosed through side channels, as it leaves observable traces in the microarchitectural state.

An attacker can cause misspeculation to a carefully selected code gadget by manipulating prediction mechanisms. The gadget exploits speculative execution to access sensitive data across permission boundaries, encoding the data as observable traces within microarchitectural components (e.g., caches). Subsequently, the attacker extracts the sensitive information via side-channel techniques, such as cache timing analysis [34, 59, 87], exploiting the residual microarchitectural state left by speculative execution.

Transient execution attacks, shown in the top row of Figure 1, typically follow three stages. First, in the ACCESS stage, a secret that is inaccessible in the normal execution path is speculatively accessed. Next, in the USE stage, the secret may be optionally processed. Finally, in the TRANSMIT stage, the secret is transmitted by modifying a microarchitectural structure, such as the cache, in a way dependent on the value of the secret.

Figure 1 provides an example of a Spectre-v1 attack code. In this scenario, the adversary manipulates the processor's branch predictor to mispredict the branch as taken, even when the index X is greater than the ARRAY1_SIZE. As a result, the subsequent ACCESS instruction speculatively loads a secret value using an out-of-bound index ARRAY1[X]. The secret is then processed in the USE stage, where it is encoded into an INDEX value. Finally, the TRANSMIT instruction accesses ARRAY2 with this INDEX, leaving observable traces in the cache. Once speculation is over, the attackers can extract the secret through a cache timing side-channel.

Existing transient execution mitigations differ in their choice of delaying the ACCESS [16, 23, 57, 75, 77, 90], USE [13, 30, 67, 81, 89], or TRANSMIT [11, 12, 20, 38, 41, 50, 58, 64–66, 85] stages of the TEA until speculation is resolved [36], as depicted in the bottom three rows of Figure 1. Delaying the TRANSMIT stage offers better performance but provides limited security. In contrast, delaying the ACCESS stage ensures the strongest security guarantee, but incurs high performance overheads which sometimes even translates to disabling the speculative execution entirely. Regardless of the stage they target, these defenses often require complex hardware and software mechanisms to implement effectively [36, 63].

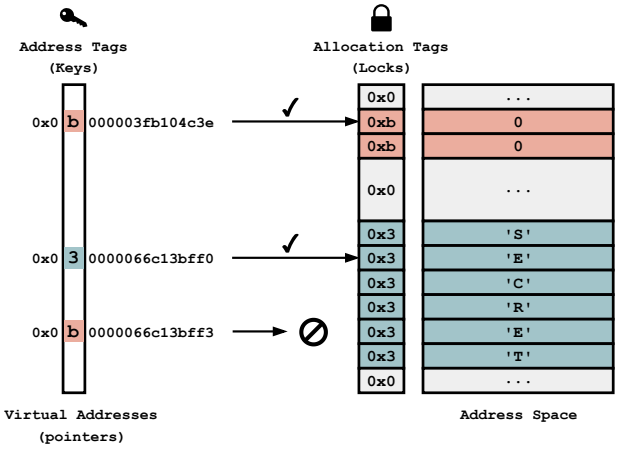


Figure 2: Overview of MTE memory tagging

2.2 Memory Safety Vulnerabilities

Memory safety vulnerabilities occur when programs improperly access memory, violating intended permission boundaries. These issues often result from programming errors such as buffer overflows, use-after-free, use-before-initialization, or failing to validate pointers. Various software and hardware techniques have been proposed to enforce memory safety, aiming to prevent unauthorized accesses while balancing performance and compatibility.

Traditionally, software-based approaches such as *Address Sanitizer (ASan)* [72], *SafeCode* [27], and *SoftBound* [54] have been employed to detect and mitigate memory safety violations. While *ASan* is a powerful tool for development, its significant runtime overhead makes it impractical for deployment in production environments. To address this limitation, hardware-assisted solutions such as *HWASAN* [73] have emerged. *HWASAN* utilizes hardware acceleration to streamline *ASan*'s memory checks, offering a significant improvement over software-based solutions alone. However, *HWASAN* still introduces a considerable performance penalty. To further improve memory safety without sacrificing performance, hardware-based mechanisms such as memory tagging have been developed. The success of these mechanisms has prompted manufacturers to integrate them into their production hardware. ARM, for example, introduced the *Memory Tagging Extension (MTE)* [71] as part of the ARMv8.5 architecture.

2.3 Memory Tagging

Memory tagging has been proposed as a tool to associate metadata with memory, at a finer granularity than the page size, to enable a range of security related capabilities, including protection against memory safety vulnerabilities. For instance, the ARM MTE [71] employs a 4-bit *allocation tag* referred to as a lock with each 16-byte of memory called a *tag granule*. Pointers that access these memory locations also carry a 4-bit *address tag*, known as a key, stored in the top byte of the pointer. ARM's Top-Byte Ignore (TBI) feature repurposes this typically unused top byte for address tag storage without affecting address translation. During memory accesses, the hardware compares the pointer's key with the memory location's lock.

Any mismatch triggers a fault, preventing unauthorized access, as shown in Figure 2.

Heap memory allocation serves as an example of ARM MTE's functionality. The `malloc()` call assigns a tag to both the allocated memory block (in 16-byte chunks) and the returned pointer. Subsequent accesses using this pointer are validated against the memory block's tag. By assigning unique tags to different memory regions, MTE can detect out-of-bounds accesses, and by updating the tag of a memory region after it is freed, MTE can detect use-after-free errors. However, while memory tagging mechanisms such as ARM MTE are effective in detecting traditional, non-speculative memory accesses, they are not used to limit accesses during speculative execution.

3 SpecASan Design and Implementation

To address the challenge of preventing unauthorized speculative memory accesses exploited by various TEAs, we propose *Speculative Address Sanitization (SpecASan)*. SpecASan builds on memory tagging hardware support such as ARM's MTE, and the software stack supporting it, to extend memory safety to speculative execution with minimal overhead. As a result, it provides robust protection without compromising performance when the speculation does not break memory safety, but delays speculative memory accesses when they appear to break memory safety. This section overviews the design and implementation of SpecASan.

3.1 Threat Model and Protection Scope

We assume a powerful adversary capable of exploiting speculative execution vulnerabilities to read arbitrary memory locations, and of using any microarchitectural covert or side channels to exfiltrate the sensitive data accessd during speculation. Consistent with typical TEAs assumptions such as Spectre-type attacks [18, 24, 35, 42–44, 52, 69], the attacker is assumed to have the ability to execute unprivileged code. The attacker's objective is to bypass privilege boundaries and address space limitations to access and exfiltrate information residing anywhere within the system's memory; these include the boundaries of a sandbox, a user process, a virtual machine, or secure enclave.

Scope of SpecASan Mitigation: SpecASan's primary goal is to prevent *unauthorized speculative memory accesses* in the first place from forwarding data to caches and internal microarchitectural buffers during speculative execution. SpecASan deliberately excludes committed (or bound-to-commit) memory accesses from protection, as these are no longer speculative operations. These accesses can be analyzed by programmers and compilers, and existing memory safety mechanisms like ARM MTE offer protection for such scenarios. It also allows safe speculative memory operations where the instruction tag matches the memory tag. Additionally, physical attacks employing methods such as electromagnetic or power analysis [26, 31] fall outside the scope of SpecASan. TEAs that aim to leak the MTE tag [40] are also out-of-scope, as software is in charge of defining protection boundaries with tags. More importantly, software can define the boundaries in flexible ways, including using deterministic tag assignment [33], where leaking the tag would not allow attackers to bypass protection boundaries.

3.2 Overview and Design Goals

The high-level design principle of SpecASan is to enforce memory tagging-based protection boundaries during speculative execution. In particular, we aim to achieve the following design goals:

- **G1**: We aim to prevent any memory load operation executed speculatively from receiving data with a mismatched tag.
- **G2**: We aim to prevent any memory store operation executed speculatively from altering any in-transient memory state when tags mismatch.
- **G3**: We aim to prevent any unsafe load/store operation executed speculatively from altering microarchitectural states (i.e., leaving traces in microarchitectural components such as caches).

These requirements prevent *unsafe memory accesses* from execution (G1 and G2), including any alteration of microarchitecture state (G3). As a result, an attacker cannot read a secret speculatively, preventing the essential first step of TEAs. Specifically, SpecASan ensures that speculative access to all potential memory sources and destinations, including microarchitectural components is permitted only if the accessed memory has a matching tag with the corresponding memory access instruction. If there is a tag mismatch, the speculative access is delayed. In most cases, mismatched accesses are an indicator of misspeculation, and delaying them does not affect performance. By ensuring that even speculative accesses satisfy memory safety requirements, systems can be protected against transient execution attacks.

3.3 Microarchitectural Changes

This section overviews microarchitectural changes and assumptions made to achieve our design goals. It describes the various locations in the memory access path and explains how they are protected, including key microarchitectural resources such as the Load/Store Queue (LSQ) and the Line-Fill Buffer (LFB).

3.3.1 L1 and Lower Level Caches. Each 16-byte data granule is assigned a 4-bit allocation tag (i.e. `lock`). Consequently, a 64-byte cache line would hold four allocation tags, as illustrated in Figure 3. Assuming this cache model, the two highest address offset bits can be used to concurrently look up the allocation tag for each cache line, alongside the regular cache tag lookup. A mismatch between the retrieved allocation tag associated with the cache line and the address tag (i.e. `key`) embedded in the memory request address' top byte signifies a tag mismatch/memory safety violation. Additionally, SpecASan modifies the cache to propagate the tag check operation to the earliest point that tag checking is possible and forwards the tag check outcome back to the core. The L1 cache utilizes a dedicated signal for this purpose, while lower-level caches incorporate a single-bit flag within Miss Status Handling Register (MSHR) entries, which is also included in the memory access response to indicate the tag check outcome (safe or unsafe). Dedicated cache maintenance operations, such as clean and invalidate operations, ensure the coherence of the stored allocation tags in the cache with the tags stored for the same address in other caches within the system.

3.3.2 Load/Store Queue (LSQ). The LSQ is a core microarchitectural component in the memory access path of high-performance, out-of-order processors. It is responsible for ensuring the correct ordering of memory access instructions in a way that satisfies the

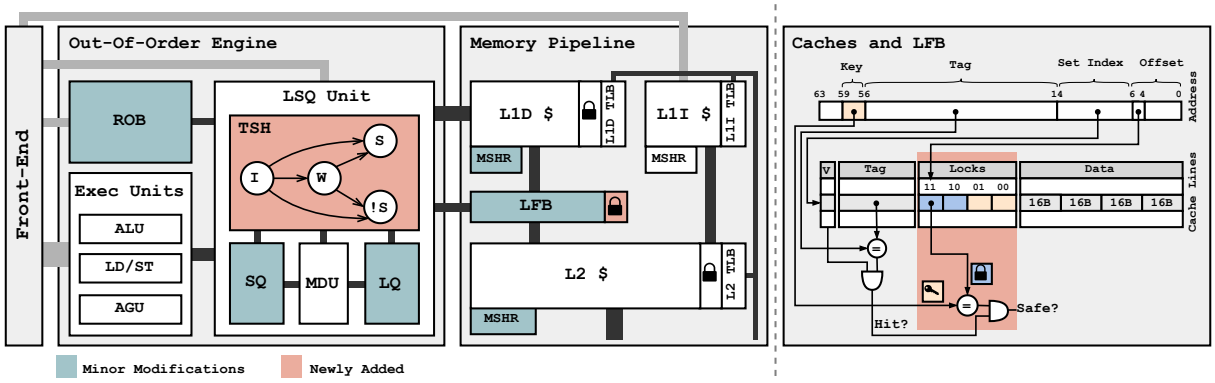


Figure 3: Overview of the SpecASan architectural modifications (left) and the conceptual cache and line fill buffer design to support allocation tag storage and tag checking (right).

memory consistency model. The LSQ consists of two dedicated buffers: the Load Queue (LQ) and the Store Queue (SQ), which track in-flight load and store operations respectively. SpecASan extends memory tagging into these microarchitectural buffers to enforce memory safety during speculative execution. We introduce a waiting mechanism within the LSQ to address potential speculative memory safety vulnerabilities. Specifically, SpecASan augments LSQ entries with a two-bit ***tag check status (tcs)*** field, which tracks different states of a tag check operation and it can be in one of four states: "init" (00), "safe" (01), "unsafe" (10), or "wait" (11). A dedicated *Tag-Check Status Handler (TSH)* is introduced within the LSQ to manage tag verification outcomes in coordination with the Reorder Buffer (ROB). This mechanism is used to ensure the desired actions, such as evaluating tag checking outcome, delaying memory access instructions until their validity is confirmed or generating tag check faults as necessary.

3.3.3 Line Fill Buffer (LFB). The LFB is another component in out-of-order processors designed to enhance performance by supporting non-blocking writes and cache line-fill operations. The role of the LFB is to hold cache values in transit to allow the cache to continue handling other operations while it is waiting for events such as a cache line fill following a cache miss. The LFB is also used when a store instruction requests ownership to change the state of a cache line from shared to exclusive. During this process, any shared instances of the cache line in other caches are invalidated, but the cache is not stalled. An LFB entry is allocated to track the status of the pending cache line fill or ownership change. Once the requested data is returned from lower-level caches or when exclusive ownership is granted, the cache line is written back into the cache. In the meantime, if another memory request targets the same cache line, and the cache line is not yet valid in the cache, it can be fetched directly from the LFB.

The LFB can be exploited in certain types of TEAs called Microarchitectural Data Sampling (MDS) attacks, where speculative execution accesses data in the LFB before it is written back to the cache. Attackers can use speculative instructions to trigger cache misses or line fill operations, then use side-channel techniques to infer the contents of the LFB, potentially exposing sensitive data [21, 68, 80]. To mitigate these risks, it is crucial to make the LFB safer by enhancing its access controls and validation mechanism.

To ensure safe access to data residing in the LFB, SpecASan extends LFB entries to include allocation tags associated with the cache lines. A lightweight tag-checking mechanism, similar to the one used in the cache, is integrated into the LFB. This allows the system to verify memory access requests by checking the requested address tag against the allocation tag stored in the LFB. The cache maintenance and coherency mechanisms are extended to manage the allocation tags not just in the cache but also in the LFB. For instance, allocation tag store operations, such as the ARM MTE's STG instruction, must now also check the LFB to update allocation tag values associated with specific tag granules within cache lines. This ensures that allocation tag integrity is maintained throughout both cache and LFB operations, enabling memory safety during access to the LFB.

3.3.4 Main Memory and Memory Controller. SpecASan uses a similar approach to the ARM MTE extension for handling memory tag allocation, storage, and check in the main memory and memory controller. In the main memory, tags are stored in a separate address space called tag storage with a specific base address. The memory controller handles the tag check operation by creating two separate memory access requests to the data memory and the tag storage simultaneously. The fetched allocation tag, which is stored in the tag storage, is checked against the address tag of the memory access operation to validate its safety. SpecASan modifies the memory controller to communicate the tag check outcome (safe or unsafe access) to the upper levels of the memory subsystem. In the event of a tag mismatch, the data is not returned to the upper memory levels or the core along with the memory response. Instead, the empty data field in the response could be utilized to integrate or encode the correct tag, enabling faster validation of subsequent requests to the same address at higher levels. However, this is a design choice and is not incorporated into the current implementation of SpecASan mechanism.

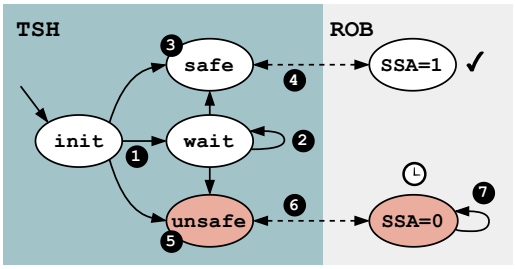


Figure 4: the state machine representing SpecASan's high-level mechanism.

3.4 SpecASan Operation

Figure 4 illustrates a high-level state machine of the operations implemented by SpecASan. This state machine outlines the sequence of stages involved in evaluating tag-check outcomes for load and store operations, as well as the coordination between the Reorder Buffer (ROB) and the Tag-Check Status Handler (TSH).

Upon issuing a load/store instruction, corresponding entries are allocated in the LQ/SQ, and ROB. The ***tcs*** field associated with these instructions is initialized to the "init" state, indicating the start of the tag validation process.

No Store-to-Load Forwarding: Before a load in the LQ is issued to the memory subsystem, it checks for matching store instructions in the SQ with the same address. If all younger store addresses are resolved and there is no match, the LSQ sends a memory access request to the L1D Cache (or LFB) and the TSH updates the load instruction's ***tcs*** to the "wait" state ❶, where it remains until the tag-checking operation completes and its outcome is returned ❷. During this access, a tag check operation compares the allocation tag with the address tag of the request. If there's a tag match, the access is deemed safe, and the requested data is returned to the LSQ. The TSH transitions the ***tcs*** to the "safe" state ❸, and notifies the ROB that this is a *safe speculative access* (SSA = 1) ❹. If there's a tag mismatch, the request is considered a potential *unsafe* access, and it must wait until speculation is resolved. To prevent leaving microarchitectural traces, the load does not return data; only a response containing the tag-checking outcome returns to the LSQ. The TSH changes the ***tcs*** to the "unsafe" state ❺ and signals the ROB of an *unsafe speculative access* (SSA = 0) ❻, preventing the ROB and dependent instructions from proceeding until speculation is resolved ❼. SpecASan allows any independent instruction or any instruction under the speculation of another independent branch to proceed without waiting for speculation resolution. Furthermore, the ROB sends a signal to the TSH to mark any dependent memory load/store instructions within the LQ/SQ as "unsafe" ❻. In a small-scale ROB with efficient broadcasting, marking dependent instructions could approximate a single-cycle operation. However, in a larger ROB with complex dependency tracking, it is more likely to require multiple cycles due to architectural constraints.

If the branch was speculated correctly and an unsafe speculative access occurred during the speculation window, the ROB raises a tag-check fault, which is immediately addressed. However, if the branch was speculated incorrectly, all misspeculated instructions are flushed without leaving any microarchitectural trace.

Store-to-Load Forwarding: If all preceding store instructions are resolved and a load-store match exists, store-to-load forwarding occurs *only if address tags match*. In this case, the TSH transitions the ***tcs*** to "safe" state ❸ and notifies the ROB of a safe speculative access (SSA=1) ❹. If address tags mismatch, store-to-load forwarding is prevented ❺, and the TSH notifies the ROB of a speculative unsafe access (SSA=0) ❻, either waiting for branch speculation resolution or raising a tag-check fault if the instructions are not under another branch speculation ❼. If unresolved stores precede the load, the Memory Disambiguation Unit (MDU) may speculate an address mismatch with all older stores, opening a memory dependency speculation window. In this case, a memory access request is sent to the cache to fetch data. During this window, two scenarios may occur: (1) The response returns before speculation is resolved. If the tag-check outcome indicates a safe access (***tcs***="safe", SSA=1) ❹, the fetched data is allowed to be used by dependent instructions. However, if there's a tag mismatch, the data is not forwarded, the TSH transitions ***tcs*** to "unsafe," and the ROB is notified of a speculative unsafe access (SSA=0) ❻. (2) One or more stores are resolved before the response returns. To maintain correct memory ordering, the response is discarded. Store-to-load forwarding occurs if address tags match; otherwise, it is prevented, and a tag-check fault is raised if the instructions are not under another speculation ❻.

With this support in place, all speculative accesses to data are regulated using the memory safety property enjoyed by committed path instructions. From a performance perspective, SpecASan enables all accesses that are safe to speculatively execute, incurring little to no overhead for these instructions. It is worthwhile to note that unsafe accesses are likely to be either misspeculated instructions or memory safety violations. Stopping these instructions should have little to no impact on performance, since neither productively advances the computation. By guiding the memory safety decisions through the compiler/program analysis, SpecASan provides this protection without requiring complex overheads to identify which instructions to delay.

4 Security Evaluation

This section delves into the security evaluation of SpecASan. We will systematically assess how SpecASan mitigates various TEAs, highlighting its strengths and potential limitations compared to existing solutions.

4.1 Attacks Exploiting Memory Safety

SpecASan robustly mitigates transient execution attacks that exploit unsafe or unauthorized memory accesses. By delaying *unsafe speculative loads and stores* until speculation is resolved, SpecASan ensures that speculative memory accesses cannot leak sensitive data through microarchitectural side channels.

Spectre-v1 (Bounds-Check Bypass) [43], is a powerful TEA where the attacker mispredicts a branch to bypass software-based bounds checks, allowing speculative execution to access out-of-bounds memory and leak sensitive data. SpecASan detects the speculative out-of-bounds access via tag mismatch, and delays the memory load until the branch condition is resolved. This ensures that speculative

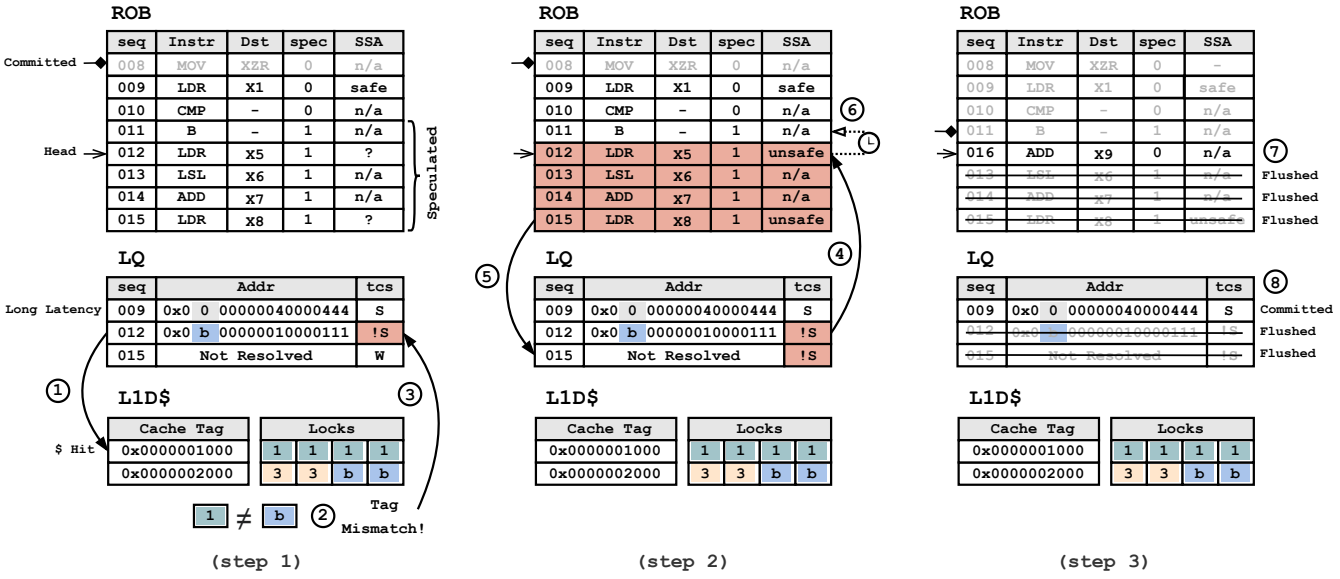


Figure 5: An example of SpecASan's mitigation mechanism, demonstrating its effectiveness in blocking a Spectre-v1 attack.

paths cannot access unauthorized memory, fully mitigating *Spectre-v1*. To illustrate the SpecASan mitigation process against *Spectre-v1* attack, Figure 5 presents a step-by-step breakdown. The code snippet in Listing 1 highlights the instructions exploited by *Spectre-v1* to speculatively access sensitive data during a mispredicted execution path and subsequently leak this information through an attacker controlled side channel.

The attack begins with a load instruction (`seq=009`) that accesses `ARRAY1_SIZE`, resulting in a cache miss and initiating a long latency memory fetch operation. During this latency, the subsequent compare instruction (`seq=010`) is delayed. Exploiting this delay, the mistrained branch predictor directs execution along an attacker-crafted speculative path (`spec_v1_path`). All instructions issued from this point forward are speculative until the branch is resolved. When the speculative load instruction (`seq=012`) is issued, the LQ allocates an entry and marks its ***tcs*** as "wait" (`W`). A memory read request is then sent to the L1 data cache ①. Although the address hits in the cache, a tag mismatch occurs, indicating an unsafe memory access ②. The L1 data cache responds with a tag mismatch signal, preventing data forwarding to the core and LSQ ③.

Upon receiving the signal, the TSH transitions the load instruction's ***tcs*** field to "unsafe" (`!S`), and notifies the ROB of the unsafe speculative access. The ROB updates the SSA of the load instruction to 1, marking it as *unsafe* ④. Consequently, the ROB also marks all dependent younger memory instructions (`seq=015`) as *unsafe* and signals the TSH to transition their corresponding ***tcs*** fields in both the LQ and SQ to "unsafe" ⑤.

The ROB then stalls executing the unsafe memory accesses and their dependent instructions until the branch prediction is resolved ⑥, while allowing the independent instructions to proceed. Since the branch was mispredicted, all instructions executed speculatively are flushed from the ROB, including the *unsafe* load and its dependents ⑦. The ROB subsequently initiates a new entry for the correct path's next instruction (`seq=016`). It also signals the LQ and SQ to flush all speculative entries, including the unsafe accesses ⑧.

By delaying unsafe memory accesses until branch resolution, SpecASan prevents unauthorized speculative memory accesses from propagating data to microarchitectural components or dependent instructions. This comprehensive mechanism eliminates information leakage, effectively mitigating *Spectre-v1* attacks.

Spectre-STL (Spectre-V4) [35], another TEA, targets data-flow dependencies in programs by exploiting the relationship between loads and preceding stores to the same memory address. Modern processors utilize memory disambiguators to predict dependencies between loads and stores, allowing loads to execute speculatively before the exact addresses of preceding stores are resolved. *Spectre-STL* leverages this speculative execution window by manipulating these predictions, enabling speculative bypass of store instructions.

```
1  LDR X1, [ARRAY1_SIZE]
2  mistrained_branch:
3      CMP X0, X1       // X < ARRAY1_SIZE
4      B.LO spec_v1_path
5  spec_v1_path:
6      LDR X5, [X2]     // ACCESS: load ARRAY1[X]
7      LSL X6, X5, #12 // USE: Y * 4096
8      ADD X7, X3, X6
9      LDR X8, [X7]     // TRANSMIT: load ARRAY2[Y*4096]
10 safe_path:
11     ADD X9, X9, #1
```

Listing 1: ARM Aarch64 assembly PoC code demonstrating the out-of-bound access used by the Spectre-v1 attack.

This creates an opportunity for attackers to access stale or unauthorized data via speculative loads, potentially leading to information leakage through microarchitectural side channels.

SpecASan effectively mitigates this attack by delaying *tagged speculative memory load instructions* until the dependency speculation is resolved by the SQ. Meanwhile, a memory access request is issued to verify the address tag of the instruction against the allocation tag stored in the memory subsystem. If the prediction is incorrect, and the tag check was also a mismatch, the load instruction is squashed without propagating any sensitive data to caches or other microarchitectural buffers, thus preventing potential TEAs. For correct predictions, the load instruction is replayed based on the memory subsystem's tag check outcome. An unsafe access, indicated by a tag mismatch, triggers a tag check fault, alerting the system to a potential memory vulnerability. For a safe access, the data is already brought into caches, allowing the replayed load to access the requested data with minimal overhead. This mechanism effectively mitigates *Spectre-STL* type attacks by blocking the malicious speculative memory load instructions from propagating any data and modifying the microarchitectural state of the processor such as caches.

MDS attacks such as *Fallout* [21], *RIDL* [80], and *ZombieLoad* [68] rely on the fact that speculative execution can inadvertently or maliciously access stale, uninitialized, or in-flight data present in microarchitectural buffers. For example, *Fallout* [21] targets stale entries in the SQ, while *RIDL* [80] and *ZombieLoad* [68] exploit in-flight data residing in components such as the LFB. By extending its memory safety enforcement to these microarchitectural buffers, SpecASan ensures that speculative accesses to such transient data are delayed until proper validation is completed, preventing unauthorized or unsafe data usage.

SpecASan achieves this by introducing memory tagging metadata to microarchitectural buffers, enabling robust validation of speculative memory accesses. Each buffer entry is associated with memory tagging metadata, as illustrated in Figure 3. Before speculative execution can consume data from these buffers, SpecASan enforces strict tag checks to verify that the access is both authorized and safe. In the event of a validation failure, such as a tag mismatch, the speculative operation is delayed, and all dependent speculative instructions are similarly stalled until the speculation is resolved. This mechanism prevents speculative execution paths from leveraging microarchitectural buffers to leak sensitive data. Unlike traditional mitigations that rely on flushing buffers or disabling speculative execution optimizations—often at a significant performance cost—SpecASan maintains system performance by permitting validated speculative execution while blocking only unsafe operations. This comprehensive approach ensures that speculative execution adheres to robust memory safety principles, effectively neutralizing not only MDS attacks but also other speculative execution vulnerabilities that exploit lapses in memory safety.

Speculative Contention Channel (SCC) attacks, such as *Speculative Interference* [17], *SMoTHERSpectre* [18], and *SpectreRewind* [29], exploit timing variations in microarchitectural components like ALUs, MSHRs, and Reservation Stations. These attacks do not rely on cache-based leaks but instead manipulate execution timing and contention to infer secret data. *Speculative Interference* [17] and *SpectreRewind* [29] introduce timing delays that influence following instructions, while *SmotherSpectre* [18] exploits execution unit port contention to leak information through resource conflicts; this is also true for any potential side channel transmitter [55]. SpecASan mitigates these attacks since it blocks the unauthorized speculative accesses of the secret, preventing it from transmission by influencing microarchitectural timing behavior. This class of attacks highlights the advantage of stopping speculative execution at the "***access***" stage rather than mitigating side effects afterward. By enforcing strict speculative memory safety, SpecASan eliminates the root cause of speculative timing/contention leaks.

Table 1: Comparison of different mitigation mechanisms for various classes of TEAs. Symbols indicate full (●), partial (◐), or no mitigation (○). (-) denotes unimplemented mitigations.

Attack	Variant	STT	GhostMinion	SpecCFI	SpecASan	SpecASan+CFI
Spectre	PHT (aka Spectre v1) [43]	●	●	-	●	●
	BTB (aka Spectre v2) [43]	●	●	●	◐	●
	RSB (aka Spectre v5) [44, 52]	●	●	●	◐	●
	STL (aka Spectre v4) [35]	●	-	○	●	●
	BHB (BHI) [14, 15]	●	●	●	◐	●
MDS	Fallout [21]	○	○	○	●	●
	RIDL [80]	○	○	○	●	●
	ZombieLoad [68]	○	○	○	●	●
SCC	SMoTHERSpectre [18]	●	◐	●	◐	●
	Spec. Interference [17]	○	◐	○	●	●
	SpectreRewind [29]	○	◐	○	●	●

4.2 Attacks Exploiting Control-Flow

Control-flow integrity (CFI) attacks are a class of transient execution vulnerabilities that exploit speculative execution to redirect a program's control flow to malicious paths or disclosure gadgets. These attacks manipulate various microarchitectural components, such as the Pattern History Table (PHT), Branch Target Buffer (BTB), Return Stack Buffer (RSB), and Branch History Buffer (BHB), to mispredict branch outcomes or return addresses. Examples include early Spectre variants like *Spectre-PHT*, *Spectre-BTB* [43], and *Spectre-RSB (ret2spec)* [44, 52], as well as more recent attacks such as *Spectre-BHB* [14, 15], *Inception* [76], *RETBLEED* [84], and *InSpectre Gadgets* [83]. Despite targeting different microarchitectural predictors, the core mechanism of these attacks remains consistent: they divert speculative execution to attacker-controlled gadgets, which access unauthorized memory or process sensitive data, and leak this information through side channels.

SpecASan partially mitigates this class of attacks by enforcing memory safety within speculative paths. It ensures that speculative memory accesses in diverted paths are validated against memory safety rules, before being executed. This prevents unauthorized memory accesses within disclosure gadgets and blocks sensitive data from propagating to side channels. However, SpecASan does not stop speculative control flow redirection, allowing execution to follow mispredicted branches to attacker-controlled gadgets. As a result, the system remains vulnerable if disclosure gadgets access memory with valid tags or operate on already-loaded sensitive data.

To achieve comprehensive protection against control-flow integrity attacks, SpecASan must be integrated with complementary mechanisms such as *SpecCFI* [45]. *SpecCFI* follows the same philosophy as SpecASan of bringing CFI, a program analysis for ensuring control flow safety, to speculatively executed code. *SpecCFI* validates speculative control flow and delays execution of branches that do not match the legal control flow until speculation is resolved. Together, these mechanisms address both speculative control-flow integrity and memory safety, ensuring robust mitigation against TEAs.

4.3 Empirical Security Evaluation

Evaluating mitigation mechanisms on a microarchitectural simulator has always been a challenge [86], primarily due to the complexity of accurately modeling timing-dependent behaviors. Many TEAs rely on precise timing variations across microarchitectural components, such as caches, to leak secrets, making an end-to-end attack implementation infeasible in simulation environments. However, since SpecASan enforces software-level safety constraints to validate speculative memory accesses, its effectiveness can be evaluated by whether it successfully blocks unsafe speculative accesses to sensitive data. This approach is analogous to memory sanitizers, where security violations are flagged. For existing proof-of-concept implementations, we assess SpecASan's effectiveness by monitoring detection logs for malicious speculative accesses, rather than measuring secret leakage. For other TEAs without available implementations, we reconstructed attack patterns based on prior research and verified whether the simulator (detailed in §5.1) correctly identified and reported unauthorized speculative accesses, ensuring SpecASan's enforcement of memory safety.

Table 1 summarizes the mitigation capabilities of SpecASan alone and in combination with *SpecCFI*, demonstrating its ability to address a broader spectrum of TEAs. *Full mitigation* signifies that the attack is entirely prevented, whereas *partial mitigation* means the defense reduces but does not completely eliminate the attack surface. This arises for control flow speculation attacks such as *Spectre-BTB* and *RSB*. When control flow is diverted, memory safety is still enforced preventing a speculative load that violates MTE. However, the attacker may redirect control flow to a gadget with a load that has a matching tag to the secret data, enabling the secret to be speculatively read. Thus, partial mitigation refers to reducing the available attack gadgets rather than eliminating the threat completely. Preventing the exploitation of malicious control flow misspeculation is the primary reason we use *SpecCFI*.

To compare SpecASan with *GhostMinion* [11] and *STT* [89], we reference their security evaluations and claims from their original papers. Both *STT* and *GhostMinion* provide protection against all known Spectre variants; however, they remain vulnerable to MDS attacks [21, 68, 80] and offer only limited mitigation against SCC attacks, including the newer variants of *Speculative Interference* attacks [86]. In contrast, SpecASan is the only mitigation mechanism that effectively defends against MDS attacks and, when integrated with CFI enforcement mechanisms such as *SpecCFI*, provides comprehensive protection against SCC attacks.

5 Performance and Complexity Evaluation

In this section, we first present our experimental methodology and the ARM MTE model. We also present the baselines we compare against. We then present an experimental evaluation of SpecASan.

5.1 Experimental Methodology

We conducted our evaluations using the gem5 cycle-level architectural simulator [51], configured to model ARM's Memory Tagging Extension (MTE) and the microarchitectural modifications introduced by SpecASan. Detailed configuration parameters for the simulated system are provided in Table 2.

Modeling certain TEAs, such as MDS variants that exploit secret leakage from the LFB [68, 80], required modifying and extending the baseline ARM architecture to incorporate an LFB-like structure. Since the ARM architecture natively lacks an LFB, we implemented a simplified LFB model, inspired by the Intel processor's design, to accurately simulate speculative data forwarding behavior and evaluate potential vulnerabilities.

To evaluate the performance of SpecASan, we compared it against two other mitigation techniques: *GhostMinion* [11] and *STT* [89]. These represent two distinct approaches to mitigating speculative execution attacks: *GhostMinion* employs shadow structures to minimize the visibility of speculative side effects, while *STT* tracks speculatively accessed data and prevents their transmission. In our performance evaluations (Section §6), we use the default *STT* variant (i.e., *STT-Default*). The more stringent *STT-Future* variant, which extends taint tracking to include registers, is excluded from our evaluations due to the lack of memory tagging support for registers.

To implement *SpecCFI* on our baseline ARM architecture, we leveraged binaries instrumented with ARM's *Branch Target Identification (BTI)* instructions [8] as a substitute for the Intel's *Control-Flow Enforcement (CET)* [5] instructions originally utilized by *SpecCFI*. Furthermore, we extended the CPU model to incorporate the required logic for performing control-flow enforcement checks, aligning with the assumptions and design principles outlined in *SpecCFI* [45].

With respect to benchmarks, we use the SPEC CPU2017 [2] and PARSEC [19] benchmark suites, representing both single-threaded and multi-threaded workloads, respectively. Following the methodology described in *GhostMinion* [11] and *STT* [89], the SPEC CPU2017 benchmarks were executed using the *ref* input size in syscall emulation mode. Each benchmark was fast-forwarded for 10 billion instructions, followed by a detailed simulation of 1 billion instructions. For the PARSEC benchmarks, simulations were conducted with the *simsmall* input size in full-system mode, configured with 4 cores. We could not compile and therefore excluded a number of the benchmarks (8 out of 23 for SPEC CPU2017 and 6 out of 13 for PARSEC). For most of these benchmarks, tools in the required toolchain did not support memory tagging: for example, certain benchmarks required a Fortran compiler, which currently does not provide memory tagging support.

5.2 Modeling ARM MTE

To simulate ARM MTE and memory tagging functionality, we extended the Out-of-Order (O3) CPU model in gem5, configured to

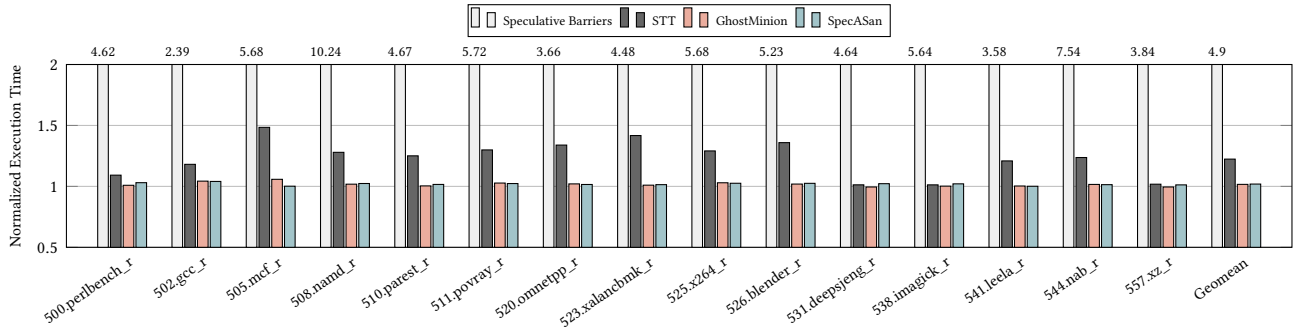


Figure 6: Performance on SPEC CPU2017 benchmarks normalized to unsafe baseline.

utilize the ARM Instruction Set Architecture (ISA). Additionally, significant modifications were made to key components of the memory subsystem, including caches and the memory controller, to incorporate memory tagging capabilities as detailed in Section §3. We also extended the cache coherence protocol and cache maintenance operations in gem5 to support tag management and ensure coherence within the memory subsystem. The current implementation of the ARM MTE supports all the main instructions added by this extension. These instructions facilitate random allocation, tag generation, tag insertion into addresses, and tag store/load to/from memory [1, 3]. Having a complete model of MTE allows us to take advantage of existing software toolchains to automatically inject MTE instructions for stack and heap memory safety [6, 7].

5.3 Performance Evaluation

Figure 6 presents the relative performance overhead (normalized execution time) of SpecASan in comparison to alternative mitigation techniques across selected SPEC CPU 2017 benchmarks. The results show that SpecASan and GhostMinion achieve similar performance with minimal overhead. However, SpecASan offers the advantage of lower hardware complexity. In contrast, STT imposes a substantial overhead compared to GhostMinion and SpecASan, making it significantly less practical for real-world deployment. The primary source of SpecASan's overhead stems from the latency associated with handling unsafe speculative accesses; however, due to their infrequent occurrence in benign applications, the impact remains negligible.

Similarly, Figure 7 illustrates the performance overhead of SpecASan in multi-threaded workloads from the PARSEC benchmark suite. Most of the observed overhead originates from the baseline ARM MTE mechanism rather than the SpecASan framework itself. Consequently, the additional overhead introduced by SpecASan is minimal, reinforcing its efficiency in securing speculative memory accesses while maintaining high performance.

Table 2: Configuration of the simulated CPU

Parameter	Configuration
CPU	ARM Cortex A76
Issue/Commit	8-way issue, 8 micro-ops/cycle commit
IQ/ROB	32-entry Issue Queue, 40-entry Reorder Buffer
Load/Store Queues (LDQ/STQ)	16-entry each
TLBs (iTLB/dTLB)	32-entry each
L1 I-Cache	32 KB, 2-way, 64B line, 1 cycle hit
L1 D-Cache	32 KB, 2-way, 64B line, 2 cycle hit, tagged
L2 Cache	1 MB, 16-way, 64B line, 12 cycle hit, tagged
Line Fill Buffer (LFB)	16-entry (cache line), 2 cycle hit, tagged

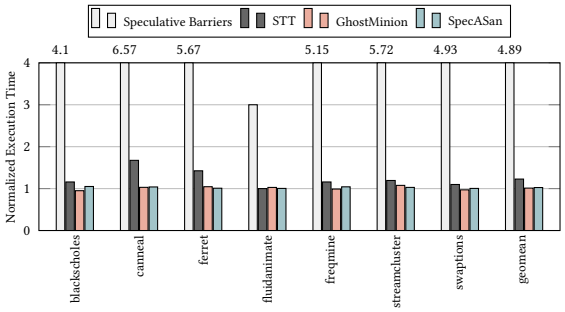


Figure 7: PARSEC Performance normalized to unsafe baseline.

Figure 8 illustrates the proportion of delayed instructions across the mitigation techniques, highlighting SpecASan's ability to effectively mitigate TEAs while minimizing unnecessary instruction restrictions and delays. This results in fewer pipeline stalls and improved overall performance. The upper section of Figure 8 presents data for the SPEC CPU 2017 benchmarks. SpecASan restricts only 0.76% of the total instructions on average, a substantial improvement compared to the 39.12% average restriction imposed by speculative barrier or fence-based methods and the 17.59% restriction caused by STT. This demonstrates SpecASan's efficiency in reducing instruction delays while maintaining security. The lower section of Figure 8 extends the analysis to the PARSEC benchmarks, where SpecASan similarly outperforms alternative approaches. On average, SpecASan restricts just 0.81% of total instructions, significantly lower than the 51.75% and 21.07% restrictions imposed by speculative barrier or fence-based methods and STT, respectively. These results emphasize SpecASan's capability to deliver robust protection against transient execution vulnerabilities with minimal impact on performance.

As detailed in Section §4.3, the integration of SpecASan, which enforces memory safety requirements during speculative execution, with a mitigation mechanism like SpecCFI, which ensures

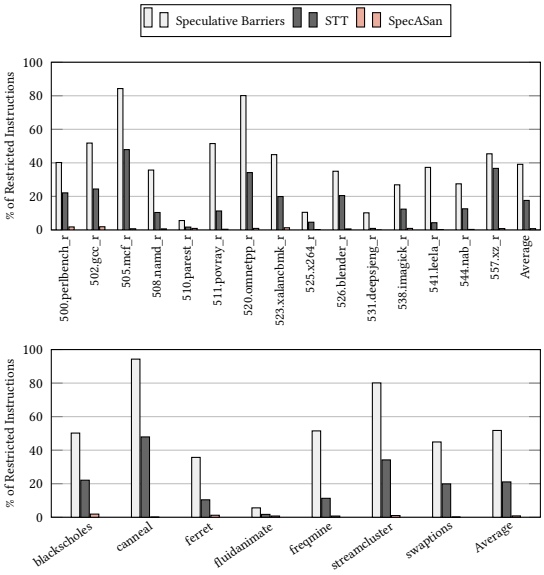


Figure 8: Comparison of different mitigation mechanisms on SPEC CPU2017 (top) and PARSEC (bottom) benchmarks based on the percentage of restricted speculative instructions.

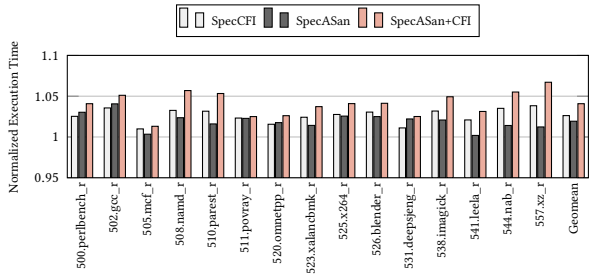


Figure 9: Performance comparison of SpecASan, SpecCFI, and the integration of both mitigation mechanisms normalized to unsafe baseline on SPEC CPU 2017.

control-flow integrity during speculation, significantly strengthens defenses against a broad spectrum of TEAs. The performance impact of these measures is summarized in Figure 9, showing geometric mean overheads of 2.6% for SpecCFI, 1.9% for SpecASan, and 4% for their combined implementation. These results prove the effectiveness of *speculative execution regulation* by enforcing software safety constraints on speculative execution, leveraging well-defined ISA extensions and microarchitectural support to achieve robust security with minimal performance costs.

5.4 Hardware Implementation Overhead

Table 3 presents the hardware cost and integration complexity of ARM MTE, SpecASan, and SpecASan+CFI across various CPU microarchitectural components. Our analysis focuses on the affected core-level structures, deliberately excluding higher-level caches,

Table 3: Comparison of hardware cost and complexity for ARM MTE, SpecASan, and SpecASan+CFI across various affected CPU microarchitectural components (percentage increase from baseline). The values in parentheses indicate the increase in the respective metric compared to the baseline ARM MTE.

Components	Metric	ARM MTE	SpecASan	SpecASan+CFI
L1 D-Cache	Area Overhead (%)	3.84	0.0	0.0
	Static Power (%)	3.31	0.0	0.0
	Dynamic Energy (%)	0.74	0.0	0.0
LFB	Area Overhead (%)	0.0	3.72	3.72
	Static Power (%)	0.0	3.11	3.11
	Dynamic Energy (%)	0.0	0.68	0.68
ROB/LSQ/MSHR	Area Overhead (%)	0.0	0.92	0.92
	Static Power (%)	0.0	0.88	0.88
	Dynamic Energy (%)	0.0	0.81	0.81
CFI Extensions	Area Overhead (%)	0.0	0.0	0.10
	Static Power (%)	0.0	0.0	0.34
	Dynamic Energy (%)	0.0	0.0	0.41
	Total Core Area Overhead (%)	0.17	0.28 (+0.11)	0.38 (+0.21)
	Total Core Static Power (%)	0.22	0.31 (+0.09)	0.65 (+0.43)

DRAM tag storage, the memory controller, and coherence mechanisms to provide a precise evaluation of in-core overheads. To quantify these costs, we utilized CACTI [47] at a 22nm technology node for SRAM-based structures. Additionally, we implemented the necessary logic components—such as tag-check logic and the TSH within the LSQ—in Verilog, followed by synthesis and cost estimation using Synopsys Design Compiler 2017.09 at the same technology node. The reported overheads in Table 3 reflect the increase relative to baseline components. While the LFB is not a native feature of ARM architectures, it must be modeled and extended for memory tagging. This overhead applies only to processors that incorporate an LFB. To evaluate the impact on CPU core area, power, and energy, we modeled an ARM CPU core with configuration parameters similar to our Gem5 ARM O3 model using McPAT [46]. The overhead of SpecCFI extensions is assessed based on their relative increase over the baseline CPU core. Additionally, the last two rows of Table 3 indicate the overall increase in total core area and static power consumption resulting from the integration of different mechanisms into the core microarchitecture.

6 Discussion

Our security and performance evaluations (Sections §4 and §6) demonstrate that SpecASan offers a balanced approach to mitigating TEAs, effectively trading off performance for security. While SpecASan's primary goal is to showcase the feasibility of hardware-enforced software memory safety contracts as a robust and efficient mitigation strategy, we acknowledge its limitations, especially our ARM MTE-based implementation.

Limitation of ARM MTE: Recall that the key insight behind SpecASan is that most, if not all, TEAs violate software-level permission boundaries to access sensitive data. Therefore, if we can utilize ISA extension to allow software to communicate those software-level permission boundaries to the hardware, and enforce the boundaries during speculative execution; then we can prevent a wide range of TEAs. Apparently, the effectiveness of SpecASan depends on how good the ISA extension is. Specifically, if an ISA extension cannot allow software to reliably specify fine-grained permission boundaries (e.g., ARM MTE), then SpecASan will also be limited

on preventing TEAs aiming to bypass such boundaries. On the contrary, if an ISA extension allows software to clearly specify fine-grained permission boundaries, then SpecASan can also provide more reliable defense against TEAs.

In this work, we choose to use ARM MTE to demonstrate SpecASan because of its wide adoption (real hardware and toolchains). However, ARM MTE also limits SpecASan's ability to prevent attacks. In particular, ARM MTE only supports 16 different tags, and its protection granularity is 16-byte. As a result, any tag collision will allow attackers to bypass the protection; and any out-of-bound access within the 16-byte cannot be detected.

For example, recent research has shown that it is possible to leak ARM MTE tag through brute-force attacks and timing analysis [4, 32, 33, 40]. So, if the software relies on using random tag to define isolation boundaries, then such boundaries may be bypassed, both in committed paths and during speculative execution. However, if the software uses deterministic tagging, and only apply it to protect security-critical data (e.g., cryptographic keys, passwords, authentication- or authorization-related metadata), as done in [39, 67], then the protection, including SpecASan's protection, will not be vulnerable to tag leaking attacks. Ongoing research is also exploring deterministic memory tagging and improved mechanisms to address this limitation [33, 48, 60, 70, 78].

Limitation of Memory Safety: SpecASan is designed to prevent TEAs that violate memory safety properties to access sensitive data. However, not all TEAs violates memory safety properties, or aim to perform unauthorized memory access. For instance, LVI attacks inject malicious values into microarchitectural buffers, which are then consumed speculatively by victim instructions. This makes *Load Value Injection (LVI)* fundamentally different, as it targets the integrity of the data being processed during speculation, rather than simply exploiting unauthorized memory accesses. SpecASan enforces strict memory tagging and validation for all speculative accesses to microarchitectural buffers, ensuring that only validated and authorized data can propagate through speculative paths. If injected or unauthorized data is accessed, SpecASan's tag validation mechanism detects the mismatch, stalling or squashing the speculative operation and its dependents. By ensuring that speculative execution operates only on safe and validated data, SpecASan effectively neutralizes the primary mechanism behind many LVI attacks. However, some LVI attacks target untagged resources, such as registers, bypassing SpecASan's protections. For example, an attacker may inject malicious values into registers or exploit speculative data paths unrelated to memory accesses, such as influencing speculative arithmetic or branch conditions. Such attacks cannot be mitigated by SpecASan. Another avenue for strengthening the enforcement of memory safety is extending it to hardware prefetchers [56], which can speculatively fetch unauthorized memory into microarchitectural buffers, such as caches. Integrating security mechanisms into prefetchers could address these risks while maintaining performance. We leave this direction for future work.

7 Related Work

Since the disclosure of the first variants of TEAs, a battle has emerged between proposing robust mitigation techniques and uncovering new attack variants that circumvent existing defenses. The primary objective of all mitigation techniques is to ensure that the microarchitectural side effects of speculatively executed instructions cannot be exploited by adversaries through side channels. To achieve this, mitigation strategies are broadly categorized into three general directions [36].

The first category focuses on proactively stopping potentially malicious "***access***" to sensitive data. Software-based techniques, such as inserting fences, as well as *Retpoline*-style mitigations [16, 77] and *Speculative Load Hardening (SLH)* [23, 90], have been employed to stop speculative execution until the validity of the access is confirmed. Hardware based approaches in this class include automatic insertion of fences before speculative load instructions [75]. SpecASan similarly prevents the "***access***" phase of attacks. Unlike other mechanisms in this category, it leverages software-defined memory safety requirements, enforcing them during speculative execution. By selectively delaying *unsafe* memory accesses, which are infrequent, rather than all speculative accesses, it effectively minimizes performance overhead.

The second category of mitigations focuses on preventing the "***use***" of sensitive data accessed during speculative execution. Mitigations in this class often employ costly mechanisms, such as *Dynamic Information-Flow Tracking (DIFT)*, to track speculatively accessed data and its "***use***" in dependent instructions until the access's validity is confirmed [13, 25, 30, 81, 88, 89]. An example of this approach is ConTExT [67], a hardware-software co-design that enables software to specify a subset of safety-critical memory objects, with hardware modifications delaying the use of these objects until speculation is resolved. While effective against TEAs, these methods introduce significant hardware complexity and incur substantial performance overhead.

The third approach to mitigating TEAs focuses on preventing speculative "***transmit***" instructions from modifying microarchitectural states until the corresponding memory access is committed. Techniques in this category encompass various strategies, including *isolation* [20, 41, 58], hiding via *shadow* structures [11, 12, 38, 65, 85], *rollback* or *cleanup* mechanisms [64], or simply *delaying* execution until speculation is resolved [37, 50]. These methods typically involve considerable performance and hardware overhead, posing challenges to their practical adoption in real-world systems. Moreover, many of these mitigations have been demonstrated to remain vulnerable to certain variants of TEAs [63].

8 Concluding Remarks

Program analysis techniques such as address sanitization, or control flow integrity, have increasingly been leveraged to improve the security of software by enforcing invariants during run-time. In this paper, we introduce a new mitigation against transient execution attacks, *SpecASan*, that extends these program analysis techniques to regulate speculative accesses to data, extending the benefits of memory safety to speculative execution. *SpecASan* extends the implementation of ARM Memory Tagging Extension (MTE) to provide this protection at low additional hardware cost, and with little impact on performance. When combined with control-flow integrity

enforcement mechanisms, SpecASan provides comprehensive protection against almost all known transient execution attacks, including those with limited practical defenses, such as Microarchitectural Data Sampling (MDS) attacks.

Acknowledgments

This research was partially supported by the U.S. National Science Foundation under grants CNS-1955650 and CCF-2212426.

References

[1] 2013. ARM® Architecture Reference Manual, ARMv8.
[2] 2017. SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. spec:org/cpu2017/
[3] 2019. Arm. Limited. Armv8.5-A Memory Tagging Extension White Paper. https://developer.arm.com/documentation/102925/latest/
[4] 2019. Arm. Limited. Speculative oracles on memory tagging. https://developer.arm.com/documentation/109544/0100
[5] 2019. Intel Corporation. Control-flow enforcement technology preview. https://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf.
[6] 2019. MemTagSanitizer — LLVM 19.0.0. https://llvm.org/docs/MemTagSanitizer.html
[7] 2020. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html
[8] 2021. ARM Limited. Arm® a64 instruction set architecture. https://developer.arm.com/documentation/ddi0596/2021-12.
[9] 2024. Chromium Partition Allocator. https://source.chromium.org/chromium/chromium/src/+/main:base/allocator/partition_allocator/src/partition_alloc/partition_bucket.cc?q=TagMemoryRangeRandomly&start=21
[10] 2024. Linux Tag-based Kernel Address Sanitizer (KASAN). https://docs.kernel.org/dev-tools/kasan.html
[11] Sam Ainsworth. 2021. Ghostminion: A strictness-ordered cache system for spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 592–606.
[12] Sam Ainsworth and Timothy M Jones. 2020. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 132–144.
[13] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 151–164.
[14] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against {Cross-Privilege} Spectre-v2 Attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 971–988.
[15] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 971–988.
[16] Markus Bauer, Lorenz Hetterich, Christian Rossow, and Michael Schwarz. 2024. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *Proceedings of the 2024 ACM Asia Conference on Computer and Communications Security*.
[17] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1046–1060.
[18] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.
[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.
[20] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.
[21] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 769–784.
[22] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.
[23] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.
[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.
[25] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 607–622.
[26] Jean-Sébastien Coron. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*. Springer, 292–302.
[27] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing alias analysis for weakly typed languages. *ACM SIGPLAN Notices* 41, 6 (2006), 144–157.
[28] Vincenzo Frascino and Catalin Marinas. 2020. Memory Tagging Extension (MTE) in AArch64 Linux. https://docs.kernel.org/arch/arm64/memory-tagging-extension.html
[29] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*. 117–126.
[30] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
[31] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: physical side-channel key-extraction attacks on pcs: Extended version. *Journal of Cryptographic Engineering* 5 (2015), 95–112.
[32] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative probing: Hacking blind in the Spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1871–1885.
[33] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. 2024. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 217–217.
[34] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 279–299.
[35] John Horn. 2018. speculative execution, variant 4: speculative store bypass.
[36] Guangyuan Hu, Zecheng He, and Ruby B Lee. 2021. Sok: Hardware defenses against speculative execution attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 108–120.
[37] Hai Jin, Zhuo He, and Weizhong Qiang. 2023. SpecTerminator: Blocking speculative side channels based on instruction classes on RISC-V. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2023), 1–26.
[38] 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 *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[39] Juhee Kim, Jinbum Park, Yoochan Lee, Chengyu Song, Taesoo Kim, and Byoungyoung Lee. 2024. PeTAL: Ensuring Access Control Integrity against Data-only Attacks on Linux. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS), Salt Lake City, UT*.
[40] Juhee Kim, Jinbum Park, Sihyeon Roh, Jaeyoung Chung, Youngjoo Lee, Taesoo Kim, and Byoungyoung Lee. 2024. TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution. *arXiv preprint arXiv:2406.08719* (2024).
[41] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.
[42] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).
[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.
[44] 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)*.

[45] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–53.

[46] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. 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*. 469–480.

[47] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 694–701.

[48] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N Asokan. 2022. Color My World: Deterministic Tagging for Memory Safety. *arXiv preprint arXiv:2204.03781* (2022).

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[50] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. {DOLMA}: Securing speculation with the principle of transient {Non-Observability}. In *30th USENIX Security Symposium (USENIX Security 21)*. 1397–1414.

[51] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[52] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2109–2122.

[53] Brand Mark. 2023. First handset with MTE on the market. https://googleprojectzero.blogspot.com/2023/11/first-handsetwith-mte-on-market.html

[54] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[55] Sumon Nath, Agustin Navarro-Torres, Alberto Ros, and Biswabandan Panda. 2024. Secure Prefetching for Secure Cache Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 92–104. doi:10.1109/MICRO61859.2024.00017

[56] Sumon Nath, Agustin Navarro-Torres, Alberto Ros, and Biswabandan Panda. 2024. Secure Prefetching for Secure Cache Systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 92–104.

[57] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506* (2018).

[58] Hamza Omar and Omer Khan. 2020. Ironhide: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 111–122.

[59] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 1–20.

[60] Aditi Partap and Dan Boneh. 2022. Memory Tagging: A Memory Efficient Design. *arXiv preprint arXiv:2209.00307* (2022).

[61] Arash Pashrashid, Ali Hajiabadi, and Trevor E Carlson. 2023. HidFix: Efficient mitigation of cache-based spectre attacks through hidden rollbacks. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[62] Andy Qin, Irene Ang, Kostya Serebryany, and Evgenii Stepanov. 2023. MTE - The promising path forward for memory safety. https://security.googleblog.com/2023/11/mte-promising-path-forward-for-memory.html

[63] Allison Randal. 2023. This is how you lose the transient execution war. *arXiv preprint arXiv:2309.03376* (2023).

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

[65] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost loads: What is the cost of invisible speculation?. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 153–163.

[66] 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*. 723–735.

[67] Michael Schwarz, Moritz Lipp, Claudio Alberto Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. Context: A generic approach for mitigating spectre. In *Network and Distributed System Security Symposium 2020*.

[68] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[69] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 279–299.

[70] Jiwon Seo, Junseung You, Donghyun Kwon, Yeongpil Cho, and Yunheung Paek. 2023. ZOMETAG: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on ARM. *IEEE Transactions on Information Forensics and Security* (2023).

[71] Kostya Serebryany. 2019. ARM memory tagging extension and how it improves C/C++ memory safety. *The Usenix Magazine* 44, 2 (2019), 12–16.

[72] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[73] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv preprint arXiv:1802.09517* (2018).

[74] Julian Stecklina and Thomas Prescher. 2018. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).

[75] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 395–410.

[76] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7303–7320.

[77] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection.

[78] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-tag: A hardware-software co-design for memory safety based on multi-granular memory tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 177–189.

[79] 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)*. 991–1008.

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

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

[82] 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. (2018).

[83] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. 2024. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. In *USENIX Security*.

[84] Johannes Wikner and Kaveh Razavi. 2022. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*. 3825–3842.

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

[86] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. 2023. Pensieve: Microarchitectural modeling for security evaluation. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[87] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*. 719–732.

[88] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 707–720.

[89] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. 2019. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 954–968.

[90] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate {SLH}: Taking Speculative Load Hardening to the Next Level. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7125–7142.