DOLMA: Securing Speculation with the Principle of Transient Non-Observability

Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai,
Ofir Weisse, Satish Narayanasamy, Baris Kasikci
University of Michigan

Abstract

Modern processors allow attackers to leak data during transient (i.e., mis-speculated) execution through microarchitectural covert timing channels. While initial defenses were channel-specific, recent solutions employ speculative information flow control in an attempt to automatically mitigate attacks via any channel. However, we demonstrate that the current state-of-the-art defense fails to mitigate attacks using speculative stores, still allowing arbitrary data leakage during transient execution. Furthermore, we show that the state of the art does not scale to protect data in registers, incurring 30.8–63.4% overhead on SPEC 2017, depending on the threat model.

We then present DOLMA, the first defense to automatically provide comprehensive protection against all known transient execution attacks. DOLMA combines a lightweight speculative information flow control scheme with a set of secure performance optimizations. By enforcing a novel principle of *transient non-observability*, DOLMA ensures that a time slice on a core provides a unit of isolation in the context of existing attacks. Accordingly, DOLMA can allow speculative TLB/L1 cache accesses and variable-time arithmetic without loss of security. On SPEC 2017, DOLMA achieves comprehensive protection of data in memory at 10.2–29.7% overhead, adding protection for data in registers at 22.6–42.2% overhead (8.2–21.2% less than the state of the art, with greater security).

1 Introduction

Speculative execution is a crucial performance optimization for modern processors. Unfortunately, the ongoing deluge of transient execution attacks [5, 10, 13, 31, 32, 35, 38, 40–42, 47, 48, 50, 53, 57, 59–61, 64, 66, 68, 69, 71–73, 77, 81] demonstrates that the implementation of speculative execution in commodity processors allows attackers to leak data during transient (i.e., mis-speculated or wrong-path) execution. Specifically, attackers exploit transient micro-ops whose operands are leaked via *covert timing channels*—e.g., hardware structures like the data cache (D-cache), which exhibit operand-dependent timing.

Transient execution attacks can be classified into two primary categories [9]. The first class of attacks rely on delayed handling of microarchitectural exception-like conditions—henceforth referred to as exceptions—to leak data (e.g., Meltdown [38] and similar attacks [10, 48, 50, 53, 59, 61, 64, 68, 69, 71, 72, 77]). In certain commodity processors, speculative reads can access data in spite of—or because of—exceptions. The exception is not handled until the associated micro-op reaches commit, offering attackers a window in which data can be transmitted through covert timing channels. Thankfully, all known Meltdown-type attacks can be thwarted by handling potential exceptions earlier in the pipeline, such that transient reads do not propagate data to dependent micro-ops [38, 76].

The second class of attacks do *not* rely on delayed exception handling, and instead solely exploit hardware mispredictions to leak data (e.g., Spectre [32] and similar attacks [5, 13, 31, 32, 35, 40–42, 47, 60]). For instance, Spectre v1 [32] shows that an attacker in one security domain can mistrain the branch predictor to transiently bypass a bounds check in a victim domain, thereby allowing micro-ops following a branch to leak victim data. Contrary to Meltdown-type attacks, there is no known comprehensive solution for Spectre-type attacks, apart from disabling speculation.

Because the majority of transient execution attacks use the D-cache as the covert channel [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 61, 64, 66, 68, 69, 71–73, 77, 81], initial defenses such as InvisiSpec [83] and others [1, 29, 30, 36, 54–56] have focused on protecting the D-cache. However, these solutions do not prevent numerous other covert channels [5, 9, 42, 53, 60, 70, 82] from leaking data during transient execution.

Recent solutions [3, 17, 34, 58, 76, 86, 88] acknowledge the shortcomings of cache-centric mitigations, and instead employ *speculative information flow control* to prevent secrets from entering *any* covert timing channel until speculation resolves. Unfortunately, current defenses are not comprehensive. For example, manual defenses [17, 58, 86] require error-prone annotations of secrets to limit performance overhead.

On the other hand, existing automatic defenses [3, 76, 88] suffer from high overhead. As such, they focus on the protection of speculatively-accessed data (e.g., data in memory at the beginning of the speculation window) and fail to comprehensively protect non-speculatively-accessed data (to a first approximation, data in registers at the beginning of the speculation window). For example, NDA [76] conservatively prohibits speculative micro-ops from propagating their results to *any* of their dependent micro-ops until speculation resolves. Thus, NDA eschews knowledge of the microarchitecture to achieve channel-agnostic protection, resulting in high overheads. NDA incurs 22.3% overhead to protect data in memory against Spectre-type attacks, and 100% overhead to supplementally protect against Meltdown-type attacks on SPEC 2017. To provide even partial protection for data in registers, NDA's performance overheads rise to 45–125%, respectively.

The current state-of-the-art defense, STT [86], uses speculative taint tracking to only delay dependent micro-ops

that affect processor backend timing (e.g., during execution) or frontend timing (e.g., during fetch) as a function of their operands. Thus, STT is able to significantly improve upon the overheads of channel-agnostic solutions such as NDA and variants of SpecShield [3]. Nonetheless, according to our evaluation, the overhead of protecting data in memory with STT is still 8.7–44.5%, with those figures rising to 30.8–63.4% if one extends STT to protect data in registers.

More importantly, we demonstrate that STT still allows arbitrary data leakages during transient execution. Despite documented transient execution attacks exploiting speculative stores [10, 49, 66, 73], STT assumes stores in isolation are safe unless the processor permits speculative cache line invalidations [66, 88]. However, even without speculative invalidations, stores can still leak information. In §3.3, we demonstrate a novel variant of Spectre [32] that uses a speculative store to transmit data through the TLB, despite STT's protections being enabled. Thus, STT does not yield the comprehensive protection it claims to offer; an attacker can still leak arbitrary data under both its Spectre-type threat model and Meltdown-type threat model.

In this paper, we present DOLMA, the first defense to automatically provide comprehensive protection against all existing transient execution attacks. DOLMA combines a speculative information flow control scheme with a set of secure performance optimizations, allowing it to protect data in both memory and—optionally—registers at tenable overhead. At a high level, DOLMA extends the microarchitecture to track speculative control and data dependencies, restricting execution as needed to prevent transient operand values from affecting processor timing.

DOLMA's key innovation is ensuring that a time slice on a core provides a unit of isolation in the context of known transient execution attacks. By enforcing a novel principle of *transient non-observability*, DOLMA can allow secure speculative access to select core-local resources (e.g., the TLB, L1 cache, and variable-time functional units) without loss of security.

In line with prior defenses [29, 36, 56, 76, 83, 88], DOLMA's default protection policy assumes a processor immune to Meltdown-type attacks, and therefore only provides mechanisms to mitigate Spectre-type attacks. However, as faulty data propagation is still possible in recent Intel processors [50, 59, 69, 71, 73], DOLMA additionally provides a conservative policy that extends its protections to Meltdown-type attacks.

We evaluate DOLMA on SPEC 2017 [8] in gem5 [6] and McPAT [37], using the same baseline processor as recent solutions [76, 88]. We show that DOLMA incurs negligible (<1%) area overhead and improves both security and performance over the state of the art [88]. DOLMA offers protection for data in memory at 10.2–29.7% performance overhead (energy: 10.8–29.2%), with protection for data in memory and registers incurring 22.6–42.2% performance overhead (energy: 22.4–40.9%).

In summary, this paper makes the following contributions:

- We present a novel variant of Spectre [32] that uses a speculative store to transmit data through the TLB, demonstrating that the state-of-the-art defense (STT [88]) is still vulnerable to arbitrary data leakages.
- We define and enforce the principle of transient non-observability, enabling secure speculative access to select core-local resources.
- We introduce DOLMA, the first defense to provide automatic comprehensive protection against existing transient execution attacks for data in both memory *and* registers.
- We improve both state-of-the-art security and performance, mitigating all existing transient execution attacks on data in memory at 10.2–29.7% overhead, as well as those on data in registers at 22.6–42.2% on SPEC 2017 [63].

Our implementation and evaluation infrastructure is open-source [39], including our gem5-compatible transient execution attack suite used for penetration testing.

2 Background

We first give background on speculative execution in modern out-of-order processors. We then describe how transient (i.e., mis-speculated) execution can be exploited to leak secrets.

2.1 Speculative, Out-of-Order Processors

A modern out-of-order (OoO) processor fetches instructions in program order and decodes them into micro-ops. OoO processors keep track of program order via a circular queue called the re-order buffer (ROB). Micro-ops enter at the tail of the ROB in-order upon dispatch, and exit from the head of the ROB in-order upon commit. However, rather than waiting for all elder micro-ops to retire, micro-ops in the ROB issue (i.e., begin executing) as soon as their operands become ready—potentially out of program order. Thus, OoO processors avoid idle execution units, exploiting instruction-level parallelism to improve efficiency over in-order processors.

To further improve efficiency, processors implement control-flow and data-flow speculation to avoid pipeline stalls. For example, the branch prediction unit (BPU) avoids stalls at fetch via control-flow speculation on a branch's target address (i.e., the next program counter) prior to branch resolution. The memory dependency unit (MDU) helps avoid stalls at issue via data-flow speculation on when a load with ready operands can bypass an elder store with unresolved operands.

Additionally, numerous modern processors do not handle exception-like conditions until the associated micro-op reaches commit, thereby implementing *exception speculation*. Specifically, these processors allow read micro-ops (e.g., loads) to broadcast their results to their dependants regardless of potential exceptions (e.g., permission faults).

In the event of mis-speculation, the processor must be able to revert to non-speculative state in order to maintain program correctness. Thus, when the processor detects mis-speculation for a given micro-op, younger entries in the ROB are *squashed*, meaning their effects will never become

```
1  // assume probe_array is flushed from cache
2  // speculatively access secret (will fault)
3  secret_byte = *kernel_addr;
4  // transmit by caching dependent element
5  tmp = probe_array[secret_byte * 512];
6  ...
7  // later in code, after recovering from fault
8  // infer secret via min time index (cached)
9  for (guess = 0; guess < 256; guess++) {
10   start_time = rdtscp();
11   tmp = probe_array[guess * 512];
12   times[guess] = rdtscp() - start_time;
13 }
14 secret = get_min_index(times);
```

Listing 1: Pseudocode for Meltdown [38]. The attacker exploits delayed fault handling to speculatively transmit kernel data via the D-cache timing side channel.

architecturally-visible. If necessary, the mis-speculated micro-op is re-issued according to non-speculative state, and execution resumes on the correct path.

2.2 Transient Execution Attacks

Squashing ensures that transient execution does not become architecturally-visible. However, the *microarchitectural* effects of transient execution may still be visible, depending on the processor implementation. Thus, under certain conditions, attackers can exploit covert timing channels to leak data.

Meltdown-type attacks. Meltdown [38] and similar exploits [10,48,50,59,61,64,68,69,71,72,77] exploit exception speculation to leak data. By allowing data propagation to proceed until the exception is handled at commit, processors present a transient attack window during which hardware protections can be bypassed. Attackers ensure that the sensitive data can be later inferred—in spite of squashing—by transmitting the value through microarchitectural state that is not reverted during squashing (e.g., D-cache lines).

A simplified version of Meltdown is shown in Listing 1. Key to the attack is the probe array, which the userspace attacker flushes from the D-cache prior to the attack. During the transient execution window (starting at line 3), the attacker is able to load a kernel value due to delayed exception handling. The attacker then uses that kernel value as an index into the probe array, loading the corresponding element into the cache (line 5). Since the cache update is not reverted during squashing, the attacker can later infer the secret value by timing access to each element in the probe array (lines 9–13). The element that is accessed most quickly corresponds to a cache hit, revealing the secret value (line 14).

The recent MDS attacks [10, 59, 71–73] similarly exploit exception speculation to leak data. However, unlike Meltdown, the address of the data leaked during transient execution does not necessarily correspond to the faulty load's address. Rather, the processor transiently forwards in-flight data: either arbitrary data, or data whose address matches a subset of the faulty load's address bits. CrossTalk [53] builds upon MDS primitives to leak data through the so-called *staging buffer* on Intel CPUs (shared amongst all cores).

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by caching dependent element
6   tmp = probe_array[secret_byte * 512];
7 }
```

Listing 2: Pseudocode for Spectre [32]. The attacker exploits a misprediction in victim code to speculatively transmit victim data via the D-cache timing side channel.

Prior work [76,82,83] has additionally theorized that various hardware events (e.g., interrupts, microcode assists, Intel TSX transaction aborts, etc.) could produce dangerous transient behavior in a similar way to microarchitectural exceptions. Indeed, during the revision of this paper, the TAA [59, 71] variants of MDS attacks exploited TSX transaction aborts. We consider these events to be special types of microarchitectural exceptions, where all micro-ops succeeding the event should be considered faulty until the processor pipeline is flushed.

Spectre-type attacks. Spectre [32] and similar exploits [5, 13,31,32,35,40,41,47,60] do not rely on exception speculation, but rather solely exploit control-flow or data-flow speculation arising from hardware prediction units to leak data. Prior to prediction resolution, a *Spectre gadget* transiently executes, transmitting data through a covert channel. The attacker later recovers the value using techniques similar to those in Meltdown.

A simplified version of Spectre is shown in Listing 2, also using the D-cache as the transmission channel. As in Meltdown, the attacker relies on a probe array to help leak the secret value. For simplicity, the attacker and victim share access to the probe array in our example. However, we note that the attacker and victim arrays can be at different physical (and virtual) memory locations; the arrays must merely compete for the same cache lines.

The attacker trains the victim code to transiently jump from a branch (line 2) to a vulnerable gadget (lines 3–6). The branch condition does not have to be related to the secret, and the gadget can be anywhere in the program; for simplicity, we show the gadget in the body of the mispredicted branch. Inside the gadget, vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and loads the corresponding element into the D-cache (line 6). The attacker later times access to each probe array element to retrieve the secret value.

Notably, recent exploits [61, 69] demonstrate that transient execution attacks may combine delayed exception handling and explicit hardware mispredictions to leak data. Because these exploits still rely on exception-like conditions, we consider them to be Meltdown-type, not Spectre-type.

3 Problem

Providing secure speculative execution requires that a processor does not leak transient operand values. In this section, we show that no existing defense satisfies this requirement, due to design flaws and security-performance trade-offs.

```
1 // victim code, mispredicted branch
2 if (some_condition) {
3   // speculatively access secret
4   secret_byte = *secret_addr;
5   // transmit by updating TLB via store
6   probe_array[secret_byte * 4096] = tmp;
7 }
```

Listing 3: Pseudocode for the access and transmit phases of a new Spectre [32] variant that leaks data through the D-TLB using a store micro-op.

3.1 Cache-Centric Defenses

Since the majority of transient execution attacks leak data through the D-cache, early defenses have focused on the D-cache transmission channel [1, 29, 36, 54–56, 83]. Though effective in protecting this channel, these works do not mitigate numerous other covert channels [5, 9, 42, 53, 60, 70, 82].

3.2 Memory-Centric Defenses

Recent solutions [3, 76, 88] acknowledge the shortcomings of cache-centric defenses, and instead focus on automatically preventing the speculative transmission of secrets via *any* covert channel. However, these solutions only protect data that is speculatively-accessed (e.g., loaded from memory during speculation); they fail to provide comprehensive protection for data in registers at the beginning of the speculation window.

In a transient execution attack on memory, prior work [30, 60, 76, 88] notes that the attacker relies on a two-step Spectre gadget; the gadget first *accesses* the secret by loading it into a register, and then *transmits* the secret via a dependent micro-op whose execution yields operand-dependent timing variations. Thus, attackers seeking to exploit victim programs rely on the presence of such two-step gadgets in the victim binary.

However, in the case of an attack on an unprivileged (e.g., general-purpose) register-based secret, the *access* step can be performed non-speculatively (e.g., the victim loads the secret into the register file prior to the beginning of the speculation window). Thus, if the attacker wishes to leak this register-based secret, they only need to execute the transmit portion of the classic Spectre gadget (line 6 of Listing 2). A "register" Spectre gadget is therefore embedded within every "memory" Spectre gadget, meaning *there are at least as many register Spectre gadgets as there are memory Spectre gadgets*.

Despite the risk of register leakages, automatic defenses [3, 88] are often only evaluated on protecting memory-based secrets, as a security-performance trade-off. An exception to this—NDA [76]—demonstrates that adding just *partial* protection for data in registers raises overhead from 22.3–100% to 45–125% on SPEC 2017 (depending on the threat model).

3.3 Attacking the State of the Art

The current state-of-the-art defense, STT [88], introduces the concept of speculative taint tracking to protect speculatively-accessed data during transient execution. In this section, we show that arbitrary data can still be leaked in spite of STT.

Despite existing transient attacks exploiting speculative stores [10, 49, 66, 73], STT incorrectly assumes that prohibiting store-triggered speculative cache coherency invalidations is sufficient to prevent transmission via stores in isolation [66, 88]. However, while stores might not speculatively modify cache state on many processors, stores can still leak information via the TLB—including on the processor used in STT's evaluation—among other channels [6, 10, 12, 88].

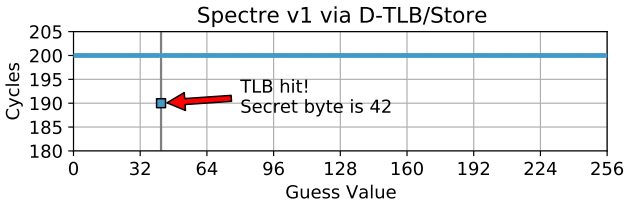

Figure 1: Leaking a speculatively-accessed secret through the D-TLB—despite enabling STT [88] protection—via a speculative store in the gem5 simulator [6].

As a result of this erroneous assumption, STT does not comprehensively prevent transient execution attacks that use stores to transmit a secret-dependent address, whether Spectre-type or Meltdown-type. Here, we demonstrate the most straightforward store-based exploit for brevity. We defer discussion of an additional, more subtle vulnerability in STT to DOLMA's design (§5.4).

Listing 3 displays the pseudocode for a novel Spectre variant that uses a transient store to leak data through the D-TLB, building on prior work [19] exploiting the TLB side channel. Inside the Spectre gadget (lines 3–6), vulnerable victim code accesses a secret byte (lines 4), uses the secret as an index into the probe array (line 6), and *speculatively stores the corresponding address in the TLB* (line 6). The attacker later recovers the secret using aforementioned techniques.

The result of running this attack with STT's protections enabled atop our baseline version of the gem5 simulator [6] is shown in Fig. 1. As pictured, the Spectre variant clearly leaks the secret byte (42). Thus, *arbitrary data can be leaked during transient execution on STT-protected processors*.

4 Scope of Protection

DOLMA considers an attacker exploiting transient execution to leak secrets (i.e., data) through any covert timing channel. DOLMA does not consider non-speculative side channels [15, 16, 20, 52, 84, 85], nor side channels that require physical access to the machine during the attack (e.g., power [33] and EM [44]). While physical side channels are viable sources of leakage, timing channels currently expose a larger threat surface, as they are remotely-exploitable.

DOLMA offers two protection policies, based on the processor's implementation of speculative execution. Technically-speaking, all micro-ops are speculative until they reach the head of the re-order buffer (ROB), at which point they are guaranteed to not be squashed. However, depending on the microarchitecture, not all speculation can

leak secrets. For simplicity, in the rest of this text, we assume that "speculation" refers to the subset of speculation that poses a security threat. We precisely define the speculative scenarios under consideration in each protection policy.

DOLMA's protection policies can additionally be tuned based on the data that the user wishes to protect. For instance, if the user only wishes to protect speculatively-accessed data (e.g., data in memory at the beginning of the speculation window, as opposed to data already loaded into registers), they may disable a subset of DOLMA's protections accordingly.

4.1 DOLMA-*Default*

DOLMA-*Default* assumes that the processor inherently mitigates all Meltdown-type attacks by preventing potentially faulty micro-ops from broadcasting (i.e., propagating) their results to dependent micro-ops. Therefore, DOLMA-*Default* only addresses Spectre-type attacks.

DOLMA-*Default* considers all hardware prediction units (e.g., units that speculate on control dependencies or data dependencies) to be sources of speculation. Thus, DOLMA-*Default* considers any micro-op fetched (control dependency) or issued (data dependency) as a result of a hardware prediction unit to be a potential source of leakage. While the exact units are implementation-specific, we detail generalizable considerations for both a typical control-flow prediction unit (the branch prediction unit) and a typical data-flow prediction unit (the memory dependency unit).

Branch Prediction Unit (BPU). The BPU can induce transient execution in three scenarios. First, the BPU can mispredict whether a branch is taken, as shown in Fig. 2a. Second, the BPU can mispredict the target of the branch. Thus, DOLMA-*Default* must prevent information leakages stemming from micro-ops following a branch in the ROB, until the prediction resolves as correct or the processor squashes.

In the third scenario, the BPU can mispredict a non-branch to be a branch (i.e., before decoding the non-branch's opcode, the BPU mispredicts that the instruction is a branch and fetches from the wrong address). However, because the misprediction is realized at decode (an in-order stage), the younger (transient) micro-ops can be squashed prior to operand resolution. Thus, operand-dependent timing variations are not possible.

Memory Dependency Unit (MDU). The MDU can induce transient execution for one or two reasons, depending on the memory consistency model: speculative store bypass (SSB) and speculative load bypass (SLB).

Speculative Store Bypass (SSB): The MDU may induce transient execution by allowing a load to bypass an earlier, unresolved store [47, 83], as shown in Fig. 2b. If the store resolves to an address used by the load, the load and its dependants must be squashed. Accordingly, DOLMA-*Default* must prevent leakages stemming from any load-dependent micro-ops, until all prior stores resolve.

Notably, DOLMA-*Default* need not prevent leakages stemming from the load itself in bypass scenarios, unless the load is already under consideration (e.g., due to following an unresolved branch). To understand this intuition, we consider the two possible scenarios for a speculative store bypass attack. First, the load can be used to access a secret in memory. In this case, the load relies on a dependent micro-op to transmit the secret, meaning the load itself need not be considered.

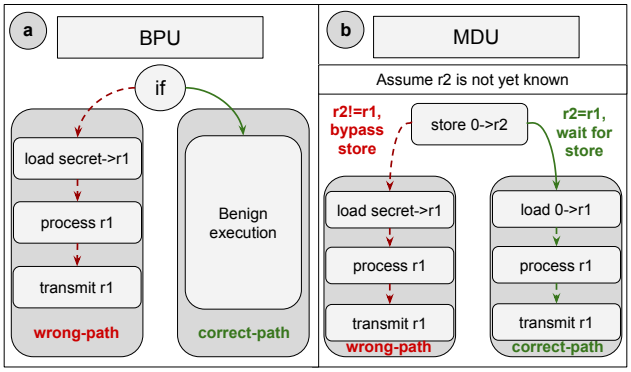

Figure 2: Examples of transient execution arising from hardware mispredictions in the (a) branch prediction unit (BPU) and (b) memory dependency unit (MDU).

Second, the load can be used to leak the (register-based) load address, which is presumed to be a secret. However, speculation does not change the load's address; it only potentially changes the value returned by the load. Even if the load is mispredicted, it will be re-executed with the same operand—the secret. Thus, this scenario is a non-speculative side channel, and is explicitly outside of DOLMA's threat model.

Speculative Load Bypass (SLB): The MDU may induce transient execution for a second reason in memory consistency models that enforce a form of total store ordering. In such models, transient execution can arise when a younger load bypasses an elder, unresolved load [56, 82]. If the elder load resolves to an address used by the younger load—and the cache line for the address is invalidated in the interim—the younger load and its dependants must be squashed to enforce memory consistency.

DOLMA-*Default* only considers dependent micro-ops of SSB loads, and not the dependants of SLB loads. SSB allows a single thread of execution to transiently read secrets explicitly overwritten in program semantics, posing an obvious security threat. On the other hand, an SLB load only reads stale data if the cache line is invalidated by another core. For memory shared among cores, such writes could occur at an arbitrary time. Thus, the programmer cannot assume the stale data has been overwritten before these loads execute, and must therefore reason about the safety of dependent micro-ops irrespective of speculation. As such, DOLMA-*Default* does not consider dependants of SLB loads.

4.2 DOLMA-*Conservative*

Despite the existence of a comprehensive solution for all Meltdown-type attacks (namely, preventing data propagation in the presence of potential microarchitectural exception-like

conditions), faulty data propagation is still possible in recent Intel processors [50, 59, 69, 71, 73]. Therefore, DOLMA-*Conservative* assumes that loads and load-like privileged register reads can transiently bypass exception-like conditions, inducing exception speculation until they retire. Thus, in addition to the speculation considerations of DOLMA-*Default*, DOLMA-*Conservative* prevents leakages stemming from all dependants of a load-like micro-op, until the load-like micro-op retires.

4.3 Simultaneous Multi-Threading

In the context of transient execution attacks, simultaneous multi-threading (SMT) can be used to access secrets (e.g., MDS attacks [10, 59, 71–73] can access secrets from a sibling logical core) or to transmit secrets (e.g., SMotherSpectre [5] can transmit a secret via issue port contention between attacker and victim sibling logical cores). Under DOLMA as well as prior speculative information control flow defenses [3, 76, 88], SMT *accesses* are safe, provided that the accessed data cannot modify a transmission channel (e.g., the D-cache) as a function of its value during speculation.

This leaves the question of how to deal with speculative SMT transmission channels. SMT contention creates a myriad of potential transmission channels—both speculative and non-speculative—via resource contention for core-local resources such as the TLB, L1 cache, and each functional unit. Thus, in the presence of SMT, prior work makes the performance-inhibiting assumptions that (1) *all* unsafe TLB/L1 accesses must be delayed, and (2) *no* unsafe micro-ops may use fast-path optimizations (e.g., variable-time arithmetic) [3, 76, 88].

However, DOLMA shows that these assumptions are unnecessary in the context of existing transient execution attacks. Even without DOLMA, potential SMT transmission channels are comparatively difficult to exploit in production environments. Namely, the attacker and victim must be co-scheduled on the same physical core and contend for the same secret-dependent resource on the exact same processor cycle. Indeed, unlike notoriously-reliable channels such as the D-cache [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 64, 66, 68, 69, 69, 71–73, 77, 81], speculative transmission via SMT contention has only been demonstrated by a single attack (SMotherSpectre [5]). Nonetheless, we show that DOLMA's design naturally mitigates SMotherSpectre in §5.4.

5 Design

DOLMA has two primary goals. First, in the context of each protection policy, the value of a transient operand (i.e., an operand of a micro-op that will be squashed) cannot affect the timing of non-transient micro-ops. Second, in order to make such security tenable for real-world systems, DOLMA must incur as little performance overhead as possible.

At a high level, DOLMA adds state to track the speculation status of each micro-op in the re-order buffer (ROB). DOLMA then uses this state to restrict (e.g., delay) execution, such that transient operands cannot observably affect timing.

Given the overhead of related defenses [3, 76, 88], DOLMA's key contribution is enforcing a novel principle of transient non-observability that obviates the need to delay execution in certain contexts. In doing so, DOLMA enables protection to scale to registers with tenable performance overhead.

In this section, we first introduce the principle of transient non-observability (§5.1). We then provide the classifications for micro-ops that DOLMA uses to enforce this principle (§5.2). With these definitions, we explain DOLMA's optimizations for traditional sources of transmission (§5.3). We subsequently identify a remaining vulnerability in the state of the art [88] and present DOLMA's mitigations for this and related channels (§5.4). Finally, we specify the microarchitectural state and logic used to appropriately restrict speculative execution (§5.5) and lift these restrictions when speculation resolves (§5.6).

5.1 Transient Non-Observability

To prevent transmissions of secrets, DOLMA enforces a novel principle of *transient non-observability*. With regards to DOLMA's timing channel protection policies, transient non-observability is achieved by ensuring that the value of a transient (i.e., destined to squash) operand cannot affect the cycle upon which a non-transient micro-op commits—thereby preventing timing-based leakages.

More precisely, transient operand values must not cause timing variations in non-transient micro-ops via (a) out-of-order contention for core-local resources, (b) simultaneous uncore/offcore resource access, or (c) persistent state modifications—i.e., modifications that survive the transient window. Notably, such leakages can occur both via data flows (e.g., a specific microarchitectural buffer entry is accessed/modified based on a secret operand) or control flows (e.g., state is only modified on a conditional path, revealing the value of a secret conditional predicate).

In this sense, DOLMA's principle of transient non-observability is similar to the principle of speculative non-interference [23, 88, 89]. The key difference is that prior work assumes *all* operand-dependent timing variations (e.g., variable-time arithmetic and TLB/cache accesses) are inherently unsafe, as an SMT adversary (i.e., an adversary executing simultaneously on the same physical core) can observe these variations via core-local contention. This limitation yields designs that stall all variable-time micro-ops until speculation resolves, inhibiting performance [3, 76, 88]. However, as we will demonstrate (§5.4), DOLMA naturally mitigates SMotherSpectre [5]—the only transient execution attack to have demonstrated transmission via SMT contention—enabling a set of secure performance optimizations over prior work.

5.2 Micro-op Classification

Inducive and Resolvent Micro-ops. In order to identify the beginning and end of each speculation window, DOLMA requires the manufacturer to denote a set of *inducive* and *resolvent* micro-ops. An inducive micro-op is any micro-op

that can induce speculation, such as a control-flow micro-op (branch prediction) or a load (memory dependency prediction, value prediction, etc.). More specifically, a control-flow micro-op—or branch—is any micro-op that can explicitly alter program control flow (e.g., a jump, call, or return); branch prediction encompasses the BPU structures used to predict the result of these micro-ops (e.g., the branch history table [BHT], branch target buffer [BTB], and return stack buffer [RSB]).

A resolvent micro-op is any micro-op that can resolve speculation. Note that the same micro-op can induce and resolve a speculation window (e.g., a control-flow micro-op induces speculation at fetch and resolves speculation at execute). In other cases, a speculation window can be induced and resolved by different micro-ops (e.g., memory dependency speculation is induced by loads and resolved by stores).

Given a specific microarchitecture, enumerating inducive and resolvent micro-ops is trivial: the manufacturer must already define an exhaustive list of these micro-ops in order to implement their processor according to its ISA specification. If the manufacturer were to omit such a micro-op, transient micro-ops would be able to retire their effects to architectural state, violating the ISA specification and thus program correctness.

Unsafe Micro-ops. In DOLMA, *unsafe* micro-ops are speculative micro-ops whose operand values can be transmitted during transient execution via corresponding timing variations. Unsafe micro-ops can be further classified as *backend-unsafe* (e.g., loads can transmit through backend channels such as the D-cache), *frontend-unsafe* (e.g., control-flow micro-ops can transmit through frontend channels such as the BTB), or both.

Because DOLMA considers timing channels, micro-ops are only classified as unsafe in the context of timing leakages. However, mitigating other operand-dependent channels would simply require the manufacturer to denote additional micro-ops as unsafe (e.g., via microcode updates).

While the exact set of unsafe micro-ops is microarchitecture-specific, we discuss common examples in modern processors. We precisely define the set of unsafe micro-ops for the microarchitecture used in our evaluation in §7, manually enumerating this set using the aforementioned criteria for transient non-observability (i.e., operand-dependent out-of-order contention for core-local resources, simultaneous uncore/offcore resource access, and persistent state modifications). Notably, this set includes all micro-ops classified as high covert channel risk (CCR) in prior work [3]. Furthermore, unlike the state of the art [88] evaluated on the same processor, DOLMA's set of unsafe micro-ops includes all applicable micro-ops whose operands are leaked in documented transient execution attacks.

For an arbitrary microarchitecture, exhaustively identifying unsafe micro-ops requires a formal timing analysis of the RTL code, and is ongoing work. The state of the art [18] requires the programmer to manually annotate portions of the circuit description, limiting scalability to modern processors. Therefore, formal verification of DOLMA's security on an arbitrary processor necessitates advancements in these methods.

5.3 Optimizations for Traditional Backend Channels

Existing speculative information flow control defenses [3, 76, 88] delay all unsafe micro-ops until speculation resolves. In select cases, DOLMA likewise delays unsafe micro-ops. However, DOLMA's principle of non-observability—combined with minor modifications to the processor—allows a restricted form of speculative execution in two key scenarios. We describe the optimizations here, and show that they do not directly produce backend timing variations. We demonstrate that the optimizations cannot influence frontend state (and thus, cannot indirectly produce backend timing variations) in §6.

Variable-Time Execution. On a traditional processor, all micro-ops that vary execution time as a function of their operands would be unsafe. While many micro-ops only produce core-local modifications that are reverted upon squashing, they may still alter the cycle upon which other micro-ops retire due to out-of-order contention for core-local resources.

More precisely, the operand-dependent contention produced by variable-time computation is problematic when it occurs between a younger (transient) micro-op and an elder (non-transient) micro-op. While the pipeline frontend is in-order, such out-of-order contention is indeed possible in the processor backend (i.e., issue and onwards).

Accordingly, to obviate unsafe backend contention, DOLMA employs a simple policy. At a high level, DOLMA's strategy is to ensure that—when an elder and younger micro-op compete for the same backend resource—the elder micro-op is unconditionally granted access to the resource. While we cannot list every possible example of backend contention, we describe our techniques for issue and writeback ports that generalize to other contention sources.

At issue, elder micro-ops can forcibly evict younger (unsafe) micro-ops from execution units when no units would otherwise be available; the younger micro-ops are then re-issued once safe. At writeback, a priority queue ensures that the eldest micro-ops obtain access to writeback ports each cycle. That is, if there are P ports and N micro-ops ready to writeback (where $N > P$), the P eldest micro-ops obtain the ports.

With this policy, the operands of variable-time micro-ops are transiently non-observable if they (a) do not affect uncore/offcore resource accesses, and (b) do not produce operand-dependent persistent state modifications. Although these criteria conventionally include variable-time ALU micro-ops, other micro-ops clearly remain unsafe, even if core-local. For example, NetSpectre [60] shows that AVX micro-ops reset a persistent powerdown timer upon execution, meaning (operand-dependent) timing variations in AVX execution would ultimately produce (operand-dependent) persistent modifications. Thankfully, in such cases where updates are off the critical path (i.e., not required for the speculative computation), DOLMA can mitigate the channels without performance loss by only performing the updates upon commit. We discuss how DOLMA prevents leakages via conditional (e.g., control-dependent)

usage of resources like the AVX powerdown timer in §5.4.

Delay-on-Miss. Memory micro-ops (loads and stores)—produced by a variety of high-level instructions [53]—pose a greater challenge, as they can both access uncore/offcore resources *and* produce persistent state modifications that greatly affect performance. For example, memory micro-ops can produce speculative, operand-dependent contention for or modifications to the D-TLB, D-cache, load-store queue, memory dependency unit, prefetching infrastructure, global staging buffer, and associated metadata for these structures (e.g., replacement policy data). Thus, speculative memory micro-ops would normally be unable to execute without leaking secrets. However, it is possible to avoid delaying memory micro-ops in the common case without loss of security.

DOLMA novelly applies the technique of "delay-on-miss" [56] to speculative stores, building on prior work that uses delay-on-miss to achieve efficient protection for speculative loads. At a high level, delay-on-miss allows speculative memory micro-ops that hit in first-level core-local structures (e.g., the L1 TLB and—in the case of loads—L1 cache) to execute without stalling until speculation resolves. A speculative memory micro-op that misses in these structures vacates its execution unit and is placed into a dedicated stall queue (as can already be done to mask the latency of TLB misses/page table walks). Such a design allows other in-flight memory micro-ops to proceed with execution. When speculation resolves, the stalled memory micro-op is re-issued without restriction.

Importantly, DOLMA ensures that memory micro-ops do not affect replacement policy metadata or memory dependency predictions until speculation resolves, thereby eliminating these potential channels. Furthermore, if a speculative memory micro-op triggers a prefetch, the prefetch is likewise constrained to delay-on-miss behavior. Finally, because only core-local memory micro-ops are legal, the global staging buffer cannot be altered. Thus, delay-on-miss prevents transmission at two levels: the explicit channels of speculative modifications to TLB and cache entries, as well as more subtle channels of speculative updates to associated state.

5.4 Mitigating Remaining Sources of Transmission

Store-to-Load Forwarding. As noted in prior work [88], store-to-load forwarding provides an additional source of backend leakage for memory micro-ops. If a load has a complete match with an unsafe store in the store buffer, the load will not issue a memory request, and will instead use the data from the store buffer. Thus, the decision to (not) issue a memory request reveals the store's address operand.

DOLMA's contribution in this regard is to identify and address another source of leakage via store-to-load forwarding. Namely, prior work [88, 90] does not handle the case of a partial hit (i.e., where a strict subset of the load's address range is found in the store buffer), instead erroneously assuming that the only two possible cases are a complete hit or miss. However, in the case of a partial hit, neither the store buffer nor lower levels of the memory hierarchy hold the correct data in its entirety. Thus, depending on how the microarchitecture handles partial hits, the load may stall until the store completes, revealing information about the store's address via timing.

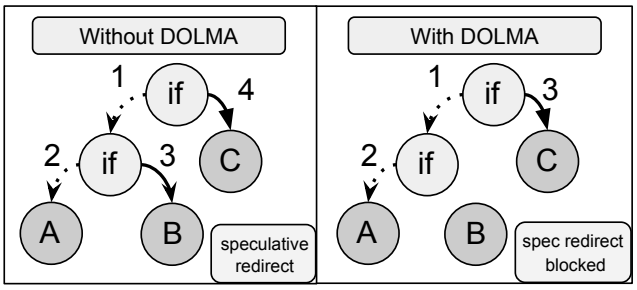

Figure 3: Without DOLMA (left), the processor speculatively redirects fetch from A to B, dependent upon a transient predicate value. With DOLMA (right), speculative fetch redirects are blocked until speculation resolves (C), thereby preventing predicate-dependent execution. Dashed lines indicate predictions, while solid lines indicate fetch redirects.

Fortunately, combined with DOLMA's protections for variable-time execution, the same protection mechanism works for both total and partial store buffer hits in the presence of stalling. That is, the processor unconditionally issues the load to the cache hierarchy, and simply ignores the response in the event of an unsafe buffer hit. If the hit was partial (meaning the buffer does not contain all necessary data), the load re-issues once the store is safe and complete.

Speculative Fetch Redirects. Control-flow micro-ops and any remaining inducive/resolvent micro-ops provide common examples of frontend-unsafe micro-ops, because these micro-ops can leak their operands via speculative fetch redirects [88], as shown in Fig. 3. For instance, if a speculative (e.g., nested) control-flow micro-op resolves as incorrect, the micro-op must signal to the frontend to redirect fetch to the appropriate program counter. However, the new PC is determined by the control-flow micro-op's predicate, meaning such a redirect leaks the predicate via dependent updates to frontend covert timing channels (e.g., the I-TLB, I-cache, and BPU), as well as potential backend covert channels (e.g., resets of the AVX powerdown timer via subsequent conditional execution [60]).

Like the state of the art (STT [88]), DOLMA additionally provides protection against more subtle sources of speculative fetch redirects. Consider the case of redirects caused by memory ordering violations (i.e., load-store aliasing, where an inducive load incorrectly bypasses an unresolved store). Such a redirect can reveal information about the load's address operand (namely, that it conflicted with that of a prior store). Thus, the redirect is clearly unsafe while the load itself is unsafe . However, even if the younger load is safe (for instance, not dependent upon any inducive loads), the elder store can still be unsafe. Accordingly, a redirect in this scenario leaks the store's address operand in an identical fashion to that of an unsafe load and must likewise be delayed. Although STT's

	micro-ops			DOLMA-Default					DOLMA-Conservative				
a	1	load	r0 -> r1	-	-	-	-		U	-	-	-	U: until retirement
	2	add	r1, r2	-	-	-	-		-	-	D	-	D: until line 1 retires
	3	jump	r1	U	-	-	-	U: until executed/squashed	U	-	D	P	D+P: until line 1 retires
b	1	store	r2 -> r3	-	-	-	-		-	-	-	-	
	2	load	r0 -> r1	U	-	-	-	U: until line 1 executes/squashed	U	-	-	-	U: until retirement
	3	add	r2, r3	-	-	-	-		-	-	-	-	
	4	load	r1 -> r4	-	-	D	-	D: until line 2 resolves	U	-	D	-	D: until line 2 retires
c	1	cmp	0x0, r0	-	-	-	-		-	-	-	-	
	2	jne	r1	U	-	-	-	U: until executed/squashed	U	-	-	-	U: until executed/squashed
	3	load	r2 -> r3	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	4	load	r4 -> r5	-	C	-	-	C: until line 2 resolves	U	C	-	-	C: until line 2 resolves
	5	jump	r3	U	C	-	P	P: until line 2 resolves	U	C	D	P	D+P: until lines 3 retires

Figure 4: Comparing DOLMA-*Default*'s and DOLMA-*Conservative*'s handling of speculation status in the ROB in three scenarios. **U** = **U**nresolved, **C** = Control-Dependent, **D** = **D**ata-Dependent, and **P** = **P**ending-Redirect. Example (a) shows a non-retired load. Example (b) shows an unresolved speculative store bypass. Example (c) shows an unresolved branch, with a nested branch blocked due to a speculative fetch redirect (line c5).

implementation code [90] allows the redirect before the store is safe, we note that STT's design correctly mentions the need to delay such redirects until both the load and store are safe [88].

By comprehensively prohibiting speculative fetch redirects, DOLMA mitigates all channels that rely on conditional transient execution to leak data (e.g., the AVX powerdown timer [60]). Notably, this protection likewise mitigates SMotherSpectre [5], the only transient execution attack to have demonstrated transmission via SMT contention. In order to create reliable contention on issue ports, SMotherSpectre uses a secret-dependent speculative redirect to fetch and issue micro-ops. In the context of Fig. 3, the speculative redirect is performed based on the secret being zero or non-zero. When the fetched micro-ops (either A or B) reach issue, they compete with micro-ops from the adversary's sibling logical core for different ports. However, the specific ports contended depend on the (different) opcodes between A and B, thereby revealing the secret value. Under DOLMA, this and similar scenarios are impossible, as speculative fetch redirects are prevented.

5.5 Enforcing Restrictions

Both DOLMA-*Default* and DOLMA-*Conservative* must restrict unsafe micro-ops that are control-dependent or data-dependent upon inducive micro-ops, delaying unsafe micro-ops that would produce observable modifications to the microarchitecture. As previously-mentioned, branch speculation provides an example of control-dependency restriction: any unsafe micro-op following a branch (e.g., jump, call, or return) in the ROB must be restricted. Memory dependency speculation provides an example of data-dependency restriction: DOLMA-*Default* must restrict the dependants of loads that bypass stores during execution. DOLMA-*Conservative* expands this mechanism to all loads (and load-like privileged register reads) in order to additionally handle exception speculation.

In order to track the speculation status of each micro-op in the pipeline, DOLMA conceptually extends each ROB entry with four bits, as shown in Fig. 4: **U**nresolved, **C**ontrol-Dependent, **D**ata-Dependent, and **P**ending-Redirect. If a micro-op is squashed, the extra bits are ignored.

Unresolved. DOLMA marks an inducive micro-op as *unresolved* until (a) its associated speculation window resolves, and (b) all elder micro-ops are also resolved. Assuming all elder micro-ops are resolved, a control micro-op resolves when it is executed. Under DOLMA-*Default*, loads are only inducive if they are issued as a result of a hardware prediction unit (e.g., speculative store bypass). Thus, such loads resolve when the corresponding prediction resolves (e.g., the bypassed store executes). Under DOLMA-*Conservative*, all load-like micro-ops are assumed to be unresolved until they retire, in order to handle exception speculation.

Control-Dependent and Data-Dependent. Speculative control dependencies can be easily tracked in DOLMA: any micro-op following an unresolved branch in the ROB is control-dependent on that branch, until the next branch introduces a new set of control dependencies.

Like prior work [88], DOLMA tracks speculative data dependencies via the register rename table. In particular, if a micro-op X consumes the output of an inducive micro-op (or its dependants), then DOLMA marks X as data-dependent. Data dependency status is propagated during broadcast (i.e., wakeup of dependent micro-ops).

Notably, reservation station entries for unsafe micro-ops are also extended with the OR of their micro-op's control-dependent and data-dependent status bits. The processor uses this signal to ensure that unsafe micro-ops do not transmit information. For instance, outgoing memory requests are tainted for unsafe micro-ops, such that the L1 cache will know to return without fetching from L2 upon a miss. As another example, DOLMA uses the dependency status—along with ordering information from the ROB—to prevent unsafe

backend contention (e.g., issue/writeback port contention between elder micro-ops and younger unsafe micro-ops).

When an unsafe micro-op is issued, a copy of its issue queue entry is placed into a dedicated *unsafe* queue for in-flight unsafe micro-ops. If an unsafe micro-op executes without stalling, its unsafe queue entry is freed. For unsafe micro-ops that cannot complete for safety reasons, each queue entry holds the index of its youngest unresolved inducer. Such a design allows for efficient wakeup when the micro-op becomes safe [88]. Specifically, if a stalled micro-op's youngest inducer is resolved, the inducer broadcasts its ROB index to this queue such that dependent micro-ops are marked as ready to issue.

Pending-Redirect. Finally, when a frontend-unsafe micro-op would initiate a fetch redirect, its ROB entry is instead marked as *pending-redirect*. Like backend-unsafe micro-ops, the frontend-unsafe micro-op also vacates its execution unit and awaits a safety broadcast.

5.6 Clearing Speculative Status

DOLMA only clears micro-ops when they become non-speculative in the context of DOLMA's threat models. For control-dependent micro-ops, this means that all elder control-flow micro-ops must be resolved. For data-dependent micro-ops, this means that all elder loads and associated resolvent micro-ops (e.g., stores) must be resolved.

When stalled backend-unsafe micro-ops are cleared, they are marked as ready to re-issue from the stall queue. When pending frontend-unsafe micro-ops are cleared, they signal their delayed redirect. Cleared micro-ops compete with the regular stream of micro-ops for backend ports. As previously stated, elder micro-ops are given preference during (re-)issue; however, DOLMA does not increase the issue width.

6 Security Analysis

The goal of our supplemental security analysis is to show that the optimizations afforded by our notion of non-observability do not introduce speculative timing channels in the context of DOLMA's protection policies. We base our reasoning on features of the baseline processor [6, 28] used in similar defenses [76, 88] (including our own), and argue that the same logic can be applied to any microarchitecture satisfying the general properties we describe here.

DOLMA introduces two optimizations due to non-observability. First, DOLMA allows for variable-time arithmetic. Second, DOLMA uses delay-on-miss [56] for speculative loads and stores. We demonstrated that these optimizations cannot directly produce timing variations in processor backend state in §5. Here, we demonstrate that transient execution cannot influence frontend timing on a DOLMA-protected processor (and thus, cannot indirectly produce backend timing variations).

Proof Sketch. On our processor, four events can influence frontend state on any given cycle. We show each event is invariant of transient values in the context of DOLMA's protection policies.

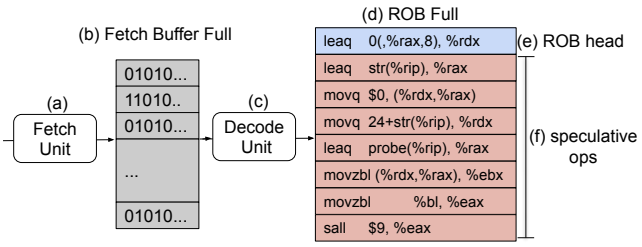

Figure 5: A simplified example of how a pipeline backup can cause the fetch buffer to fill.

(1) Backend Redirect: The backend can redirect fetch to a new PC as a result of a predicate resolution (e.g., branch or memory dependency). DOLMA delays fetch redirects until speculation resolves, meaning transient micro-ops in the backend cannot initiate a fetch redirect. Furthermore, since elder micro-ops are given preference for backend resources, a transient micro-op cannot affect the length of the speculation window (and thus, cannot influence the cycle upon which a backend redirect is performed). Thus, backend fetch redirects are invariant of transient data.

(2) Frontend Redirect: The frontend can redirect fetch to a new PC as a result of a branch prediction. Since DOLMA delays fetch redirects until speculation resolves, DOLMA prevents transient data from entering the BPU. Thus, frontend fetch redirects are invariant of transient data.

(3) Full Fetch Buffer: The processor may not increment the PC on a given cycle if the fetch buffer (i.e., the buffer for fetched instructions, before they are decoded and inserted into the ROB) is full. Delaying fetch redirects until speculation resolves—coupled with giving elder micro-ops priority in the backend— prevents transient operands from affecting the the processor frontend state (including the fetch buffer). Thus, it suffices to show that transient micro-ops cannot indirectly influence the state of the fetch buffer via a pipeline backup.

We trace back from the "full fetch buffer" scenario shown in Fig. 5 to demonstrate that only non-speculative micro-ops can cause the fetch buffer to fill. The fetch unit **(a)** fetches instructions into the fetch buffer **(b)**. The fetch buffer becomes full when the decode unit **(c)** cannot process instructions on a given cycle. The decode unit cannot process instructions if the ROB **(d)** is full. Finally, the ROB is full if the micro-op at the head of the ROB **(e)** cannot retire.

However, the head of the ROB is—by definition—non-speculative. Thus, this is only a concern if younger (speculative) micro-ops prevents the head from retiring. Since DOLMA gives elder micro-ops priority in the backend, such a scenario is impossible. Therefore, only non-speculative micro-ops can cause the fetch buffer to fill, meaning the fetch buffer is invariant of speculative micro-ops **(f)**, and thus transient data.

(4) Variable Fetch Latency: The processor may not increment the PC on a given cycle if a fetch request is delayed (e.g., due to an I-TLB or I-cache miss). Fetch latency is a

Parameter	Value
Architecture	x86-64 at 2.0 GHz
OoO Core (No SMT)	8-issue, 32 LQ entries, 32 SQ entries, 192 ROB entries, 4096 BTB entries, 16 RAS entries
OoO Core (2-SMT)	8-issue, 16 LQ entries per thread, 16 SQ entries per thread, 91 ROB entries per thread, 4096 BTB entries (dynamically partitioned), 16 RAS entries per thread
L1-I/L1-D Cache	32 KB, 64B line, 8-way set associative (SA), 4 cycle round-trip (RT) latency, 1 port
L2 Cache	2 MB, 64 B line, 16-way SA, 40 cycle RT latency
DRAM	50 ns response latency

Table 1: gem5 simulation configuration.

function of frontend state (e.g., the PC, BPU, I-TLB, and I-cache), which by (1)–(3), is invariant of transient data. Thus, fetch latency is invariant of transient data.

Therefore, processor frontend state is invariant of transient operand values in the context of DOLMA*'s threat models.*

7 Evaluation

We evaluate DOLMA's gem5 [6] implementation against the SPEC 2017 [63] benchmark suite. We estimate area and energy with McPAT [37], incorporating recommended changes for increased accuracy [80]. We sample performance throughout each benchmark's execution via the Lapidary simulation sampling framework [45, 46], which employs the SMARTS methodology [79]. More specifically, Lapidary converts periodic GDB coredumps from each benchmark's execution on real hardware into gem5 checkpoints. Following the methodology used in NDA [76] (a prior speculative information flow control defense), we configure Lapidary to warm microarchitectural structures for 5,000,000 instructions before measuring the performance of 100,000 instructions, repeated for each checkpoint.

We evaluate DOLMA with and without simultaneous multi-threading (SMT) enabled. We generate SMT workload pairings from the SPEC 2017 benchmarks using the "Balanced Random" methodology developed by Velasquez et al. [74]. This methodology ensures that each benchmark appears an equal number of times across all pairings.

In line with prior speculative information flow control defenses [76, 83, 88], we use gem5's OoO processor as our baseline. The processor's set of inducive micro-ops includes control-flow micro-ops (i.e., jumps, calls, and returns) and loads, while its resolvent micro-ops include control-flow micro-ops and stores. Its set of unsafe micro-ops includes control-flow micro-ops, loads, and stores—consistent with the micro-ops identified as high covert channel risk (CCR) in prior work [3]. The processor configuration is listed in Table 1.

We additionally compare the performance of DOLMA to the state-of-the-art speculative information flow control defense, STT [88]. As STT provides memory-only protection, we extend STT to enable optional protection for registers. We compare DOLMA to STT under both memory-only protection modes (M) as well as memory and register (M+R) modes.

Although STT's gem5 implementation is publicly-available [90], it was necessary to port STT as modifications

Defense	Overhead	Overhead-SMT	Speculation: Control (M)	Speculation: Control (R)	Speculation: Data	Speculation: Exception
Baseline OoO	0±3.8%	0±3.0%				
STT-*Spectre* (M)	8.7±4.2%	3.2±3.2%	○			
DOLMA-*Default* (M)	10.2±4.3%	3.4±3.2%	●		●	
STT-*Futuristic* (M)	44.5±4.6%	25.5±3.6%	○		○	○
DOLMA-*Conservative* (M)	29.7±4.7%	16.2±3.5%	●		●	●
STT-*Spectre* (M+R)	30.8±5.0%	17.3±3.6%	○	○		
DOLMA-*Default* (M+R)	22.6±4.8%	9.8±3.4%	●	●	●	
STT-*Futuristic* (M+R)	63.4±5.0%	36.8±3.8%	○	○	○	○
DOLMA-*Conservative* (M+R)	42.2±5.4%	22.4±3.7%	●	●	●	●

● Mitigates all existing attacks
○ Mitigates all existing attacks, except select transmissions via stores

Table 2: DOLMA compared to STT [88] in terms of total CPI overheads and mitigated attacks, using memory-only protection variants (M) as well as memory and register protection variants (M+R). *Control* transient execution attacks refer to transient execution arising from branch predictions, differentiated by whether memory or registers are leaked. *Data* transient execution attacks refer to transient execution arising from data predictions (e.g., memory dependency speculations). *Exception* transient execution attacks refer to Meltdown-type attacks that exploit delayed microarchitectural exception handling. Overhead ranges reflect 95% confidence intervals.

to DOLMA for two key reasons. First, STT's baseline performance differs significantly from that of prior speculative information flow control defenses, rendering fair comparisons impossible. For example, we found that for the *mcf* benchmark, STT's baseline yielded approximately 30% higher average cycles-per-instruction compared to the baseline of NDA (and our own), significantly skewing results. Second, SMT support for x86-64 is not functional in the STT prototype.

7.1 Performance Evaluation

Single Thread. The per-benchmark geometric mean cycles per instruction (CPI) for DOLMA-*Default* and DOLMA-*Conservative* across SPEC 2017 are shown in Fig. 6, provided for both memory-only (M) as well as memory and register (M+R) protection variants. We display these numbers alongside corresponding STT variants, and depict 95% confidence intervals for the reported CPIs.

For protection against Spectre-type attacks, STT provides STT-*Spectre*. However, unlike DOLMA-*Default*, STT-*Spectre* does not mitigate Spectre-type attacks exploiting data speculation, such as speculative store bypass [47], nor various transmissions via stores. STT-*Spectre* (M) incurs 8.7% overhead, while STT-*Spectre* (M+R) incurs 30.8% overhead. Thus, despite offering greater protection, DOLMA-*Default* (M) (10.2%) yields comparable overhead to STT-*Spectre* (M) (8.7%), and DOLMA-*Default* (M+R) (22.6%) scales to registers significantly better than STT-*Spectre* (M+R) (30.8%).

To provide the additional protection against Meltdown-type attacks offered by DOLMA-*Conservative*, STT-*Futuristic* incurs 44.5% (M) and 63.4% (M+R) overhead, but fails to

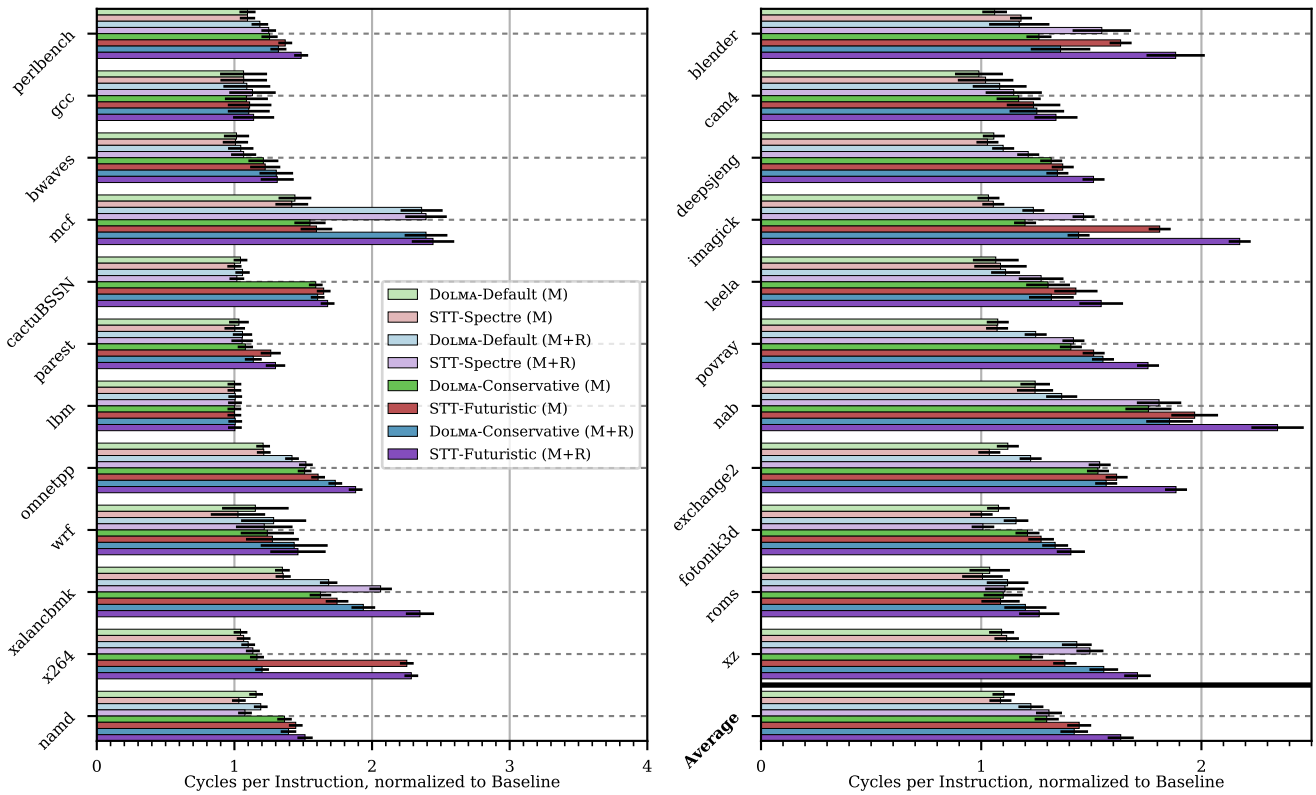

Figure 6: DOLMA's single thread performance on SPEC 2017, compared to STT [88]. Error bars depict the 95% confidence intervals.

protect select store-based transmissions. In contrast, DOLMA-*Conservative* only incurs 29.7% (M) and 42.2% (M+R) overhead to protect against all existing Meltdown-type and Spectre-type attacks on data in memory and registers, respectively.

DOLMA's ability to provide protection at lower overhead than STT primarily arises from the use of delay-on-miss for memory micro-ops. While STT insecurely allows all speculative stores to execute, STT conservatively delays all unsafe loads. In contrast, DOLMA only delays unsafe loads and stores when they miss in the TLB and—in the case of loads—the L1 cache.

With SMT (2 Threads). We compare the geometric mean of total CPI overhead across 2 threads between DOLMA and STT in Table 2, alongside single thread means. Reported CPIs are listed with 95% confidence intervals. For both DOLMA and STT, we find that the performance overhead of protection decreases with SMT enabled. This arises due to the fact that when a micro-op from some thread *A* is stalled for protection, some other thread *B* can potentially still make progress.

As with single-threaded configurations, we find that both DOLMA-*Default* and DOLMA-*Conservative*—unlike STT—prevent all existing transient execution attacks at mostly lower overheads. DOLMA-*Default* (M+R) (9.8%) again scales to registers far better than STT-*Spectre* (M+R) (17.3%), and DOLMA-*Conservative* likewise achieves lower overhead than STT-*Futuristic*—16.2% versus STT's 25.5% (M) and 22.4% versus STT's 36.8% (M+R). The only exception to this trend is DOLMA-*Default* (M) (3.4%) versus STT-*Spectre* (M) (3.2%), where performance is still roughly equivalent despite DOLMA's additional protections for store-based transmission and data speculation.

7.2 Security Evaluation

We additionally compare the effectiveness of DOLMA against transient execution attacks with STT in Table 2. DOLMA-*Default* blocks all documented Spectre-type attacks, and DOLMA-*Conservative* blocks all documented Spectre-type and Meltdown-type attacks. STT-*Spectre* variants fail to address any Spectre-type attack that exploits data speculation [47]. Additionally, all STT variants fail to comprehensively address store-based transmission—including the speculative TLB modifications and speculative partial store buffer hits mentioned in this paper.

Penetration Testing. Although simulating every known transient execution attack is not possible in gem5, we have ported a diverse set of transient execution attacks into an open-source, gem5-compatible test suite [39]. The goal of this test suite is to directly demonstrate the ability of DOLMA—as well as future defenses—to mitigate transient execution attacks across a wide range of covert timing channels (e.g., backend channels such as the D-cache and D-TLB, as well as frontend channels such as the I-cache and BTB), unsafe

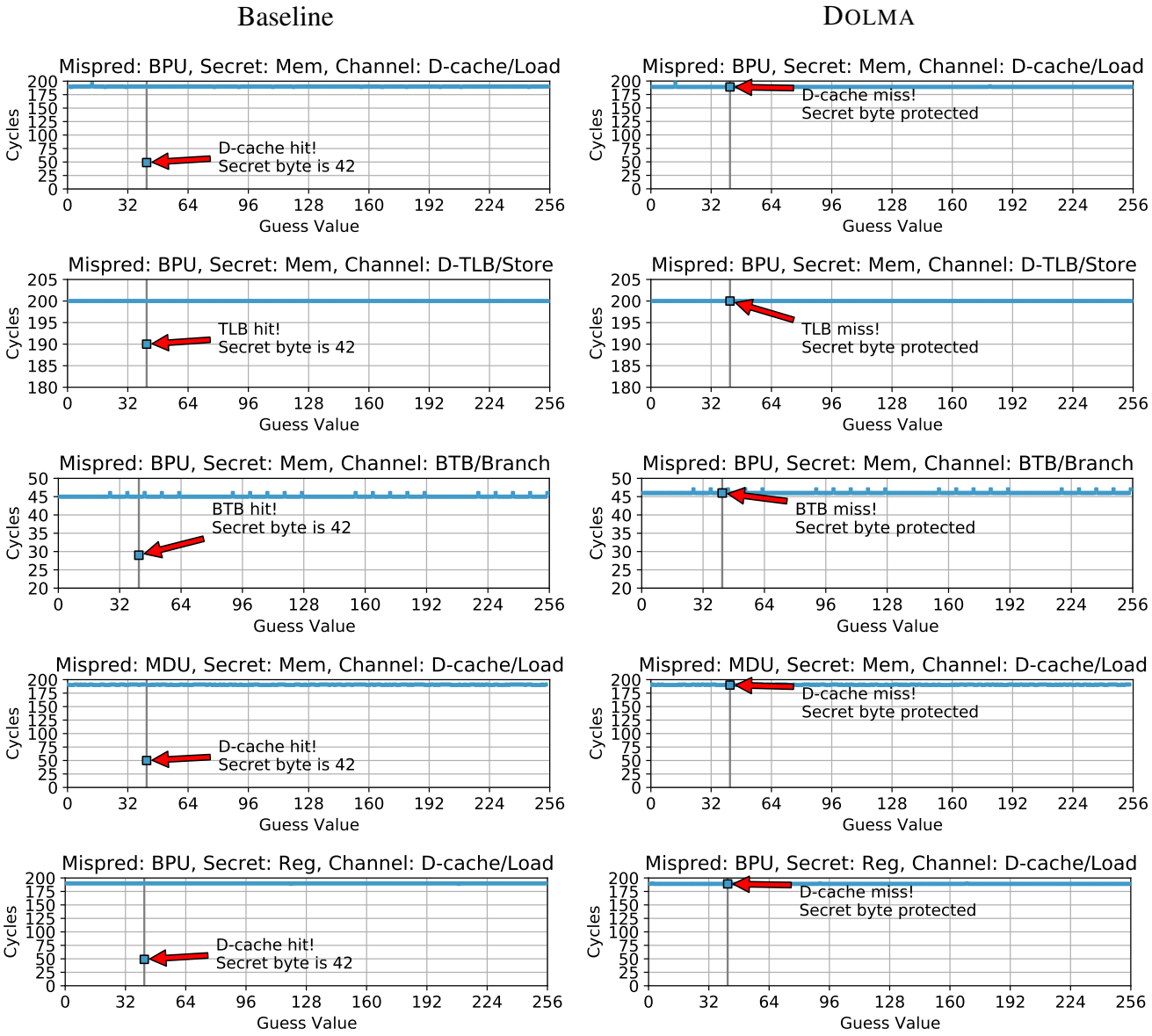

Figure 7: A demonstration of DOLMA's effectiveness in mitigating various covert timing channels. Each of the attacks leaks the value of the secret byte (42) on a baseline OoO processor (left) in gem5 [6]. In contrast, a DOLMA-protected processor prevents data from entering these channels during transient execution, thereby mitigating the attacks (right).

micro-ops (e.g., memory micro-ops and branches), types of speculation (e.g., control and data), and locations of secrets (e.g., in-register and in-memory).

We depict DOLMA's effectiveness in mitigating a sampling of these transient execution attacks in Fig. 7. Namely, we show that DOLMA mitigates timing-based transmission of speculatively-loaded data through the D-cache (load micro-op), BTB (branch), and D-TLB (store). We additionally show that DOLMA's protection applies to speculative store bypass [47] as well as attacks on non-speculative register data. As pictured, DOLMA defeats all attempted transient execution attacks, regardless of the covert channel, unsafe micro-op, type of speculation, and location of the secret.

7.3 Area and Energy Estimates

We provide area and energy estimates for DOLMA using McPAT [37] with recommended changes for increased accuracy [80]. We model DOLMA's conceptual changes to the microarchitecture as follows. The unsafe queue is conservatively implemented as a second copy of the issue queue, extended with $1 + \log_2(\text{sizeof(ROB)})$ bits per entry to hold the pending redirect bit as well as ROB index of the youngest unresolved inducer (set to 0 if all are resolved, meaning the micro-op may re-issue). Each functional unit (e.g., ALUs and FPUs) and entry in the load-store queue is extended with $\log_2(\text{sizeof(ROB)})$ bits to indicate the corresponding

Mode	Energy	Energy (SMT)
DOLMA-*Default* (M)	10.8%	4.46%
DOLMA-*Conservative* (M)	29.2%	16.4%
DOLMA-*Default* (M+R)	22.4%	10.4%
DOLMA-*Conservative* (M+R)	40.9%	21.9%

Table 3: DOLMA's normalized total energy usage (processor and caches) compared to a baseline single thread and SMT processor, respectively.

micro-op's position in the ROB; such a design allows DOLMA to enforce its backend contention policy. Finally, similar to prior work [88], entries in the frontend register alias table are extended with $\log_2(\text{sizeof(ROB)})$ bits to indicate an operand's youngest unresolved inducer (either directly produced by a data dependency, or indirectly via a control dependency).

For area, we find that DOLMA's overhead is negligible when configured for either single threaded or SMT execution, incurring 0.9% overhead compared to respective baselines. For energy, as shown in Table 3, DOLMA's normalized total energy usage is dominated by its increase in execution time; energy usage overheads for both single thread and SMT configurations roughly correspond to performance overheads for the respective baselines. Therefore, in line with performance overheads, normalized energy usage for SMT configurations incurs lower overhead (normalized to an SMT baseline) than a single thread configuration (normalized to a single thread baseline).

7.4 Limitations

First, as the gem5 baseline processor does not allow faulty data propagation, we are unable to directly demonstrate the effectiveness of DOLMA-*Conservative* against Meltdown-type attacks. However, given that DOLMA-*Default* clearly prevents SSB [47]—and the restriction policy DOLMA-*Default* applies to SSB load dependants is extended to *all* load dependants in DOLMA-*Conservative*—we argue that DOLMA-*Conservative* indeed mitigates Meltdown-type attacks.

Second, we only demonstrate transmission via the BTB using the simpler of gem5's two BTB implementations (i.e., one that uses a less complex indexing function). However, as speculative BTB transmission has been demonstrated on real hardware [42], it is clearly a viable channel.

Third, the gem5 baseline only features constant-time ALU and FPU operations, meaning DOLMA's benefits over prior work [88] for these operations are not modelled.

Fourth, because gem5's system emulation mode does not add latency for TLB misses, our figures include an artificial TLB miss latency of 10 cycles for visualization purposes. We conservatively calculated this latency by assuming a 2-cycle penalty for the initial miss, plus a 4-cycle L1 lookup for each TLB stage. We verified that a TLB hit only occurs for the secret value in the simulator.

Fifth, modeling hardware in software simulators limits evaluation accuracy in the name of implementation feasibility. This limitation is particularly prevalent for total energy estimates, which depend on the accuracy of gem5 performance numbers and McPAT calculations.

8 Related Work

Transient execution attacks. Spectre [32] and Meltdown [38] are the first known attacks that exploit speculative execution to leak data via microarchitectural timing side channels. Since then, a wave of attacks have emerged. Most of these attacks use the D-cache as a timing side channel [10, 13, 31, 32, 35, 38, 40, 41, 47, 48, 50, 57, 59, 61, 64, 66, 68, 69, 71–73, 77, 81]. Attackers have also demonstrated speculative data leaks through the AVX unit [60], issue ports [5], I-cache [42], BTB [42], and global staging buffer [53], as well as suggested the possibility of speculative data leaks through the TLB [56, 83]. Recent work demonstrates that TSX Asynchronous Aborts can also be exploited to leak secrets [59, 71].

Software Mitigations. Due to the difficulty of patching deployed hardware, numerous software patches for transient execution attacks exist. Unfortunately, no software-only techniques provide comprehensive protection.

For Meltdown-type attacks, software mitigations tend to focus on enforcing stronger isolation between security domains. For example, kernel address space layout randomization (KASLR) increases the difficulty of finding kernel data to leak [21]. However, while KASLR makes Meltdown more difficult to exploit, it does not altogether prevent it. Kernel page table isolation (KPTI) defeats the original Meltdown variant by placing kernel data in a separate address space [14, 21]. However, KPTI does not prevent other Meltdown-type attacks [10, 48, 59, 61, 64, 68, 72, 77]. Other proposed defenses offer attack- or channel-specific OS/VMM code modifications. For instance, flushing the cache on context switches between privilege levels only mitigates the cache channel [24, 77].

A wider variety of software mitigations have been proposed for Spectre-type attacks. Compiler techniques include modifying vulnerable code patterns to prevent a subset of transient execution. For example, Retpoline [67] protects call and return instructions from speculatively leaking values on the return stack buffer as in Spectre-RSB [35]. Unfortunately, Retpoline fails to protect against other Spectre variants.

Other compiler techniques insert LFENCEs or add artificial data dependencies to prevent transient loads [11, 51, 62, 65, 75], potentially using program analysis techniques or hardware-software contracts to identify information flows [22, 23, 75]. These techniques can mitigate attacks on memory-based secrets, such as Spectre v1 and Spectre v2 in some cases. However, they either fail to protect register-based secrets, fail to cover all Spectre variants (e.g., SSB [47]), or incur higher overhead than the state-of-the-art hardware defense [88].

Hardware Mitigations. Hardware defenses offer the ability to mitigate transient execution attacks at their microarchitectural sources [43, 76]. The comprehensive solution for all Meltdown-type attacks is to prohibit potentially faulty micro-ops from propagating their results in future proces-

sors [38, 76]. In the interim, microcode patches have been individually issued for Meltdown-type attacks [24, 26, 72, 77].

Hardware patches also exist for certain Spectre-type attacks. Processors can automatically insert LFENCEs after branches and context switches via microcode [25, 26], as well as disable SSB [2, 27]. SpecCFI [34] prevents Spectre v2 by restricting speculative jumps to an authorized set of targets. None of these techniques, nor their union, can mitigate all Spectre variants.

MI6 [7] provides secure enclaves in an out-of-order processor via microarchitectural resource isolation (e.g., flushing core-local structures on context switches). Compared to DOLMA, MI6 does not support SMT and requires the use of a software monitor executing non-speculatively to manage transitions between the enclave and outside world.

InvisiSpec [83] and others [1, 29, 36, 54–56, 83] only protect select load-based transmission channels (e.g., the D-cache), in contrast to speculative information flow control defenses such as DOLMA. The InvarSpec microarchitecture [91] optimizes these cache-centric defenses, using compiler-generated instruction annotations to help determine when a load's execution would not explicitly reveal speculative operand values.

Manual speculative information flow control defenses [17, 58, 86] require the programmer to annotate secrets for protection, as opposed to the automatic protection provided by DOLMA. The strict timing requirements for annotated data ensure that speculative execution produces neither transient *nor non-transient* side channel leakages (i.e., in the event speculation resolves correctly). While effective in providing protection for annotated data, the security of manual defenses relies on proper programmer annotation of secrets.

Existing automatic speculative information flow control defenses [3, 76, 88] prevent varying sets of speculative dependants from issuing until speculation resolves. Notably, SpecShield [3] only protects speculatively-accessed data (e.g., data in memory), and NDA [76] does not prevent leakage of register-based secrets via a single transient micro-op. STT [88] fails to comprehensively mitigate transmissions via stores, whether secrets are in memory or in registers. In contrast, DOLMA protects against all known transient execution attacks, and incurs 8.2–21.2% less overhead than the state of the art [88] when scaling to protect data in registers.

Finally, during the revision of this paper, the authors of STT published an optimization framework (SDO [87]). Like DOLMA, SDO improves performance over STT primarily by allowing speculative loads to safely execute in certain scenarios. SDO creates a "data-oblivious" load, which behaves independently of its operands as well as other unsafe operands. However, to achieve such a load, SDO requires that (1) for each operand-dependent resource access (e.g., a cache bank access), the load instead accesses all such resources (e.g., all banks), and (2) the load blocks all other accesses to these resources until complete. Accordingly, an attacker could intentionally issue speculative data-oblivious loads to temporarily deny cache access to other tenants, in a similar manner to prior cache denial-of-service attacks [4, 78]. Furthermore, SDO does not address any of the store-based security vulnerabilities present in STT and does not consider the effects of the staging buffer [53]. Therefore, SDO requires additional considerations for multitenant environments.

9 Conclusion

Efficiently mitigating transient execution attacks is challenging. Initial hardware mitigations focused on cache transmission [1, 29, 30, 36, 56, 83]. Manual speculative information flow control defenses [17, 58, 86]—though effective—require error-prone annotations of secrets. Automatic solutions fail to comprehensively protect data in registers [3, 76, 87, 88] or memory [87, 88]. DOLMA introduces a novel principle of transient non-observability, combining a lightweight speculative information flow control design with a set of secure performance optimizations to protect data in memory and registers against all existing transient execution attacks.

Acknowledgements

We thank our shepherd (Kaveh Razavi), Marina Minkin, and the anonymous reviewers for their constructive feedback. Kevin Loughlin has been supported by an NSF Graduate Research Fellowship (award DGE 1256260).

Availability

DOLMA's implementation and evaluation infrastructure is available at https://github.com/efeslab/dolma.

References

[1] S. Ainsworth and T. M. Jones. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *ISCA*, 2020.

[2] AMD. Speculative Store Bypass Disable, 2018. developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf.

[3] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu. Specshield: Shielding speculative data from microarchitectural covert channels. In *PACT*, 2019.

[4] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE RTAS*, 2019.

[5] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *CCS*, 2019.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH CAN*, 2011.

[7] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, S. Devadas, et al. Mi6: Secure enclaves in a speculative out-of-order processor. In *MICRO*, 2019.

[8] J. Bucek, K.-D. Lange, et al. SPEC CPU2017: Next-Generation Compute Benchmark. In *ACM/SPEC ICPE Companion*, 2018.

[9] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[10] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, 2019.

[11] C. Carruth. *Speculative Load Hardening*. Google, 2018. llvm.org/docs/SpeculativeLoadHardening.html.

[12] C. Celio, J. Zhao, A. Gonzalez, and B. Korpan. Riscv-boom documentation, 2019.

[13] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *Euro S&P*, 2019.

[14] J. Corbet. *A page-table isolation update*. LWN, 2018. lwn.net/Articles/752621/.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO*, 2016.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, 2018.

[17] J. Fustos, F. Farshchi, and H. Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, 2019.

[18] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala. Iodine: Verifying constant-time execution of hardware. In *USENIX Security*, 2019.

[19] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 2017.

[21] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: long live kaslr. In *ESSoS*. Springer, 2017.

[22] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. Hardware-software contracts for secure speculation. *arXiv preprint arXiv:2006.03841*, 2020.

[23] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *S&P*, 2020.

[24] Intel. *Deep Dive: Intel Analysis of L1 Terminal Fault*, 2018. software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault.

[25] Intel. Intel Analysis of Speculative Execution Side Channels, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf.

[26] Intel. Speculative Execution Side Channel Mitigations, 2018. software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[27] Intel. Speculative Store Bypass, 2018. software.intel.com/security-software-guidance/advisory-guidance/speculative-store-bypass.

[28] R. E. Kessler. The alpha 21264 microprocessor. *MICRO*, 1999.

[29] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *DAC*, 2019.

[30] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, 2018.

[31] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[33] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, 1999.

[34] E. M. Koruyeh, S. Haji Amin Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation. In *S&P*, 2020.

[35] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT*, 2018.

[36] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *HPCA*, 2019.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[39] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci. DOLMA source code. github.com/efeslab/dolma, 2020.

[40] A. Lutas and D. Lutas. Bypassing kpti using the speculative behavior of the swapgs instruction. *Bitdefender Whitepaper*, 2019.

[41] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, 2018.

[42] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus. Two methods for exploiting speculative control flow hijacks. In *WOOT*, 2019.

[43] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[44] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. Eddie: Em-based detection of deviations in program execution. In *ISCA*, 2017.

[45] I. Neal. Lapidary: Crafting more beautiful gem5 simulations. medium.com/@iangneal/lapidary-crafting-more-beautiful-gem5-simulations-4bc6f6aad717, 2019.

[46] I. Neal. Lapidary: creating beautiful gem5 simulations. github.com/efeslab/lapidary, 2019.

[47] NIST NVD. Cve-2018-3639. nvd.nist.gov/vuln/detail/CVE-2018-3639, 2018.

[48] NIST NVD. Cve-2018-3640. nvd.nist.gov/vuln/detail/CVE-2018-3640, 2018.

[49] NIST NVD. Cve-2018-3693. nvd.nist.gov/vuln/detail/CVE-2018-3693, 2018.

[50] NIST NVD. Cve-2019-11135. nvd.nist.gov/vuln/detail/CVE-2019-11135, 2019.

[51] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[52] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*, 2019.

[53] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *S&P*, 2021. To appear.

[54] G. Saileshwar and M. K. Qureshi. Cleanupspec: An undo approach to safe speculation. In *MICRO*, 2019.

[55] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander. Ghost loads: what is the cost of invisible speculation? In *ACM CF*, 2019.

[56] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA*, 2019.

[57] M. Schwarz, C. Canella, L. Giner, and D. Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus. *arXiv preprint arXiv:1905.05725*, 2019.

[58] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. Context: A generic approach for mitigating spectre. In *NDSS*, 2020.

[59] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[60] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. Netspectre: Read arbitrary memory over network. In K. Sako, S. Schneider, and P. Y. A. Ryan, editors, *Computer Security – ESORICS*, 2019.

[61] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss. Speculative dereferencing of registers: Reviving foreshadow. *arXiv preprint arXiv:2008.02307*, 2020.

[62] Z. Shen, J. Zhou, D. Ojha, and J. Criswell. Restricting control flow during speculative execution. In *CCS*, 2018.

[63] SPEC. Standard Performance Evaluation Corporation SPEC CPU 2017. spec.org/cpu2017/.

[64] J. Stecklina and T. Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv preprint arXiv:1806.07480*, 2018.

[65] M. Taram, A. Venkat, and D. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 2019.

[66] C. Trippel, D. Lustig, and M. Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *MICRO*, 2018.

[67] P. Turner. *Retpoline: a software construct for preventing branch-target-injection*. Google, 2018. support.google.com/faqs/answer/7625886.

[68] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[69] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[70] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, 2018.

[71] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Addendum to RIDL: Rogue In-flight Data Load, 2019. mdsattacks.com/files/ridl-addendum.pdf.

[72] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, 2019.

[73] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *cacheoutattack.com*, 2020.

[74] R. A. Velásquez, P. Michaud, and A. Seznec. Selecting benchmark combinations for the evaluation of multicore throughput. In *ISPASS*, 2013.

[75] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE TSE*, 2019.

[76] O. Weisse, I. Neal, K. Loughlin, T. Wenisch, and B. Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*, 2019.

[77] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical Report*, 2018.

[78] D. H. Woo and H. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[79] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ACM SIGARCH CAN*, 2003.

[80] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *HPCA*, 2015.

[81] W. Xiong and J. Szefer. Leaking information through cache lru states. In *HPCA*, 2020.

[82] W. Xiong and J. Szefer. Survey of transient execution attacks. *arXiv preprint arXiv:2005.13435*, 2020.

[83] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[84] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *ISCA*, 2017.

[85] M. Yan, Y. Shalabi, and J. Torrellas. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *MICRO*, 2016.

[86] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[87] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *ISCA*, 2020.

[88] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

[89] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative Taint Tracking (STT): A Formal Analysis. *Technical Report*, 2019.

[90] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. STT source code. github.com/cwfletcher/stt, 2020.

[91] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas. Speculation invariance (invarspec): Faster safe execution through program analysis. In *MICRO*, 2020.