Ghosts in a Nutshell

Moritz Lipp, Claudio Canella

moritz.lipp@iaik.tugraz.at, claudio.canella@iaik.tugraz.at

Abstract

Modern processors optimize performance by using techniques like branch prediction and out-of-order execution. In the case where the prediction is wrong or an exception occurred, the results of so-called transient instructions need to be reverted. Unfortunately, microarchitectural state changes are not reverted allowing attacks like Spectre or Meltdown to recover secret data.

In the introduction of the talk, we will shortly discuss previous microarchitectural attacks. We will discuss the Spectre and Meltdown attacks and present a consistent and extensible systematization of transient execution attacks. That systematization led to the discovery of 6 new transient execution attacks, 2 of those are new Meltdown variants, and the remaining 4 are new ways to mistrain prediction mechanisms in Spectre-type attacks. One of the new Meltdown attacks is the first on AMD, contradicting all previous statements and beliefs that they are not affected. We will also present a novel classification of gadgets and discuss how they can be combined in different attacks. Finally, we will discuss different mitigations, highlighting their performance impact and whether they can successfully mitigate an attack.

1 Overview

In this whitepaper, we cover the topics of our talk and also provide technical background. The paper is a pre-print of the paper "A Systematic Evaluation of Transient Execution Attacks and Defenses" [1]. Spectre and Meltdown are two attacks that created a new research field for both attacks and defenses. In a short period, many different attack variants have been discovered and ad-hoc defenses created. We provide a clear and extensible systematization of both Spectre and Meltdown-type attacks, which we call *transient execution attacks*, which led to the discovery of 2 new Meltdown variants and 4 new ways to mistrain prediction mechanisms for Spectre. For that, we also present a classification of gadgets and discuss how they can be combined in different attacks. Furthermore, we provide a classification of defenses and analyze their effectiveness in mitigating attacks.

The main takeaways of both the talk and the whitepaper are as follows.

1. Current defenses for transient execution attacks have a substantial performance overhead, and not all of them work as intended.
2. There are new variants of Meltdown and Spectre out there that have not yet been discovered.

3. We can categorize Meltdown and Spectre-type attacks based on what element they exploit and defenses on how they try to stop leakage in order to avoid confusion provoked by the current naming scheme.

References

[1] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. A Systematic Evaluation of Transient Execution Attacks and Defenses. *arXiv:1811.05441* (2018).

A Systematic Evaluation of Transient Execution Attacks and Defenses

Claudio Canella[1], Jo Van Bulck[2], Michael Schwarz[1], Moritz Lipp[1],
Benjamin von Berg[1], Philipp Ortner[1], Frank Piessens[2], Dmitry Evtyushkin[3], Daniel Gruss[1]
[1] *Graz University of Technology,* [2] *imec-DistriNet, KU Leuven,* [3] *College of William and Mary*

Abstract

Modern CPU optimizations such as branch prediction and out-of-order execution are crucial for performance. Recent research on *transient execution* attacks including Spectre and Meltdown showed, however, that exception or branch misprediction events may leave secret-dependent traces in the CPU's microarchitectural state. This observation led to a proliferation of new Spectre and Meltdown attack variants and even more ad-hoc defenses (e.g., microcode and software patches). Unfortunately, both the industry and academia are now focusing on finding efficient defenses that mostly address only one specific variant or exploitation methodology. This is highly problematic as the state-of-the-art provides only limited insight on residual attack surface and the completeness of the proposed defenses.

In this paper, we present a consistent and extensible systematization of transient execution attacks. Our systematization uncovers 6 (new) transient execution attacks that have been overlooked and not been investigated so far. This includes 2 new Meltdown variants: Meltdown-PK on Intel, and Meltdown-BND on Intel and AMD. It also includes 4 new Spectre mistraining strategies. We evaluate *all* attacks in our classification tree through proof-of-concept implementations on 3 major CPU vendors (Intel, AMD, ARM). Our systematization does not only yield a complete picture of the attack surface, but also allows a systematic evaluation of defenses. Through this systematic evaluation, we discover that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches.

1 Introduction

CPU performance over the last decades was continuously improved by shrinking processing technology and increasing clock frequencies, but physical limitations are already hindering this approach. To still increase the performance, vendors shifted the focus to increasing the number of cores and optimizing the instruction pipeline. Modern CPU pipelines are massively parallelized allowing hardware logic in prior pipeline stages to perform operations for subsequent instructions ahead of time or even out-of-order. Intuitively, pipelines may stall when operations have a dependency on a previous instruction which has not been executed (and retired) yet. Hence, to keep the pipeline full at all times, it is essential to predict the control flow, data dependencies, and possibly even the actual data. Modern CPUs, therefore, rely on intricate microarchitectural optimizations to predict and sometimes even re-order the instruction stream. Crucially, however, as these predictions may turn out to be wrong, pipeline flushes may be necessary, and instruction results should always be committed according to the intended in-order instruction stream. Pipeline flushes may occur even without prediction mechanisms, as on modern CPUs virtually any instruction can raise a fault (e.g., page fault or general protection fault), requiring a roll-back of all operations following the faulting instruction. With prediction mechanisms, there are more situations when partial pipeline flushes are necessary, namely on every misprediction. The pipeline flush discards any architectural effects of pending instructions, ensuring functional correctness. Hence, the instructions are executed *transiently* (first they are, and then they vanish), *i.e.*, we call this *transient execution* [59, 52, 90].

While the architectural effects and results of transient instructions are discarded, microarchitectural side effects remain beyond the transient execution. This is the foundation of Spectre [52], Meltdown [59], and Foreshadow [90]. These attacks exploit transient execution to encode secrets through microarchitectural side effects (e.g., cache state) that can later be recovered by an attacker at the architectural level. The field of transient execution attacks emerged suddenly and proliferated, leading to a situation where people are not aware of all variants and their implications. This is apparent from the confusing naming scheme that already led to an arguably wrong classification of at least one attack [50]. Even more important, this confusion leads to misconceptions and wrong assumptions for defenses. Many defenses focus exclusively on hindering exploitation of a specific covert channel, instead of addressing the microarchitectural root cause of the leakage [49, 47, 94, 52]. Other defenses rely on recent CPU features that have not yet been evaluated from a transient security perspective [89]. We also debunk implicit assumptions including that AMD or the latest Intel CPUs are completely immune to Meltdown-type effects, or that serializing instructions mitigate Spectre Variant 1 on any CPU.

In this paper, we present a consistent and extensible systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our decision tree, all known transient execution attacks were accurately classified through an unambiguous naming scheme (cf. Figure 1). The hierarchical and extensible nature of our taxonomy allows to easily identify residual attack surface, leading to 6 previously overlooked transient execution attacks (Spectre and Meltdown variants) first described in this work. Two of the attacks are Meltdown-BND, exploiting a Meltdown-type effect on the x86 `bound` instruction on Intel and AMD, and Meltdown-PK, exploiting a Meltdown-type effect on memory protection keys on Intel. The other 4 attacks are previously overlooked mistraining strategies for Spectre-PHT and Spectre-BTB attacks. We demonstrate *all* attacks in our classification tree through practical proofs-of-concept with vulnerable code patterns evaluated on CPUs of Intel, ARM, and AMD.

Next, we provide a systematization of the state-of-the-art defenses. Based on this, we systematically evaluate defenses with practical experiments and theoretical arguments to show which work and which do not or cannot suffice. This systematic evaluation revealed that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches. Finally, we discuss how defenses can be designed to mitigate entire types of transient execution attacks.

Contributions. The contributions of this work are:

1. We systematize all (known) Spectre- and Meltdown-type attacks, advancing attack surface understanding and highlighting at least one arguable misclassification.
2. We provide a clear distinction between Meltdown/Spectre, required for designing effective countermeasures.
3. We categorize all defenses and show that most, including deployed ones, cannot fully mitigate all attack variants.
4. We exhaustively test x86 exceptions, revealing that only faults, not traps/aborts, cause Meltdown-type leakage.
5. We contribute new branch mistraining strategies, highlighting the difficulty of eradicating Spectre-type attacks.
6. We discover 2 new Meltdown attacks, including the first exploitable Meltdown-type effect on AMD, contradicting previous claims by AMD and previous works citing this.

We responsibly disclosed the work to Intel, ARM, and AMD.

Experimental Setup. Unless noted otherwise, all of the experimental results reported were performed on recent Intel Skylake i5-6200U, Coffee Lake i7-8700K, and Whiskey Lake i7-8565U CPUs. Our AMD test machines were a Ryzen 1950X and a Ryzen Threadripper 1920X. For experiments on ARM, an NVIDIA Jetson TX1 has been used.

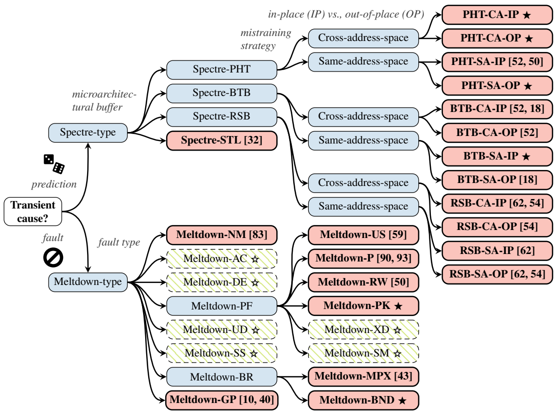

Figure 1: Transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).

Outline. Section 2 provides background. We present the systematization of Spectre in Section 3 and Meltdown in Section 4. We analyze and classify gadgets in Section 5 and defenses in Section 6. We conclude in Section 7.

2 Transient Execution

Out-of-Order Execution. On modern CPUs, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations (μOPs) that are then processed. This design decision allows for superscalar optimizations and to extend or modify the implementation of specific instructions through so-called microcode updates. Furthermore, to increase performance, CPU's usually implement a so-called *out-of-order* design. This allows the CPU to execute μOPs not only in the sequential order provided by the instruction stream but to dispatch them in parallel, utilizing the CPU's execution units as much as possible and, thus, improving the overall performance. If the required operands of a μOP are available, and its corresponding execution unit is not busy, the CPU starts its execution even if μOPs earlier in the instruction stream have not finished yet. As immediate results are only made visible at the architectural level when all previous μOPs have finished, CPUs typically keep track of the status of μOPs in a so-called *Reorder Buffer* (ROB). The CPU takes care to *retire* μOPs in-order, deciding to either discard their results or commit them to the architectural state. For instance, exceptions and external interrupt requests are handled during retirement by flushing any outstanding μOP results from the ROB. Therefore, the CPU may have executed so-called *transient instructions* [59], whose results are never committed to the architectural state.

Speculative Execution. Software is mostly not linear but contains (conditional) branches or data dependencies be-

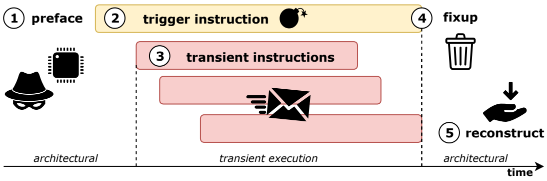

Figure 2: High-level overview of a transient execution attack in 5 phases: (1) put microarchitecture in desired state, (2) execute a *trigger instruction*, (3) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions, (5) reconstruct secret from microarchitectural state.

tween instructions. In theory, the CPU would have to stall until a branch or dependencies are resolved before it can continue the execution. As stalling decreases performance significantly, CPUs deploy various mechanisms to predict the outcome of a branch or a data dependency. Thus, CPUs continue executing along the predicted path, buffering the results again in the ROB until the correctness of the prediction is verified as its dependencies are resolved. In the case of a correct prediction, the CPU can commit the pre-computed results from the reorder buffer, increasing the overall performance. However, if the prediction was incorrect, the CPU needs to perform a roll-back to the last correct state by squashing all pre-computed transient instruction results from the ROB.

Cache Covert Channels. Modern CPUs use caches to hide memory latency. However, these latency differences can be exploited in side-channels and covert channels [53, 70, 95, 29, 63]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks, e.g., on cryptographic algorithms [95, 46, 30], user input [29, 58, 77], and kernel addressing information [28]. For Flush+Reload, the attacker continuously flushes a shared memory address using the `clflush` instruction and afterward reloads the data. If the victim used the cache line, accessing it will be fast; otherwise, it will be slow.

Covert channels are a special use case of side-channel attacks, where the attacker controls both the sender and the receiver. This allows an attacker to bypass all restrictions that exist on the architectural level to leak information.

Transient Execution Attacks. Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. For functional correctness, it is crucial that their results are never committed to the architectural state. However, transient instructions may still leave traces in the CPU's microarchitectural state, which can subsequently be exploited to partially recover unauthorized results [59, 52, 90]. This observation has led to a variety of transient execution attacks, which from a high-level however always follow the same abstract flow, as shown in Figure 2. The attacker first brings the microarchitecture into the desired state, e.g., by flushing and/or populating internal branch predictors or data caches. Next is the execution of a so-called *trigger instruction*. This can be any instruction that causes subsequent operations to be eventually squashed, e.g., due to an exception or a mispredicted branch or data dependency. Before completion of the trigger instruction, the CPU proceeds with the execution of a *transient instruction sequence*. The attacker abuses the transient instructions to act as the sending end of a microarchitectural covert channel, e.g., by loading a secret-dependent memory location into the CPU cache. Ultimately, at retirement of the trigger instruction, the CPU discovers the exception/misprediction and flushes the pipeline to discard any architectural effects of the transient instructions. However, in the final phase of the attack, unauthorized transient computation results are recovered at the receiving end of the covert channel, e.g., by timing memory accesses to deduce the secret-dependent loads from the transient instructions.

High-Level Classification: Spectre vs. Meltdown. All transient execution attacks have in common that they abuse transient instructions (which are never architecturally committed) to encode unauthorized data in the microarchitectural state. With different instantiations of the abstract phases in Figure 2, a wide spectrum of transient execution attack variants emerges. We deliberately based our classification on the root cause of the transient computation (phases 1, 2), abstracting away from the specific covert channel being used to transmit the unauthorized data (phases 3, 5). This leads to a first important split in our classification tree (cf. Figure 1). Attacks of the first type, dubbed Spectre [52], exploit transient execution following control or data flow misprediction. Attacks of the second type, dubbed Meltdown [59], exploit transient execution following a faulting instruction.

Importantly, Spectre and Meltdown exploit fundamentally different CPU properties and hence require orthogonal defenses. Where the former relies on dedicated control or data flow prediction machinery, the latter merely exploits that data from a faulting instruction is forwarded to instructions ahead in the pipeline. Note that, while Meltdown-type attacks so far exploit out-of-order execution, even elementary in-order pipelines may allow for similar effects [91]. Essentially, the different root cause of the trigger instruction (Spectre-type misprediction vs. Meltdown-type fault) determines the nature of the subsequent unauthorized transient computations and hence the scope of the attack.

That is, in the case of Spectre, transient instructions can only compute on data which the application is also allowed to access architecturally. Spectre thus transiently bypasses *software-defined* security policies (e.g., bounds checking, function call/return abstractions, memory stores) to leak secrets out of the program's intended code/data paths. Hence, much like in a "confused deputy" scenario, successful Spectre attacks come down to steering a victim into transiently computing on memory locations the victim is authorized to

Table 1: Spectre-type attacks and the microarchitectural element they exploit (●), partially target (◐), or not affect (○).

Element Attack	BTB	BHB	PHT	RSB	STL
Spectre-PHT (Variant 1) [52]	○	◐	●	○	○
Spectre-PHT (Variant 1.1) [50]	○	◐	●	○	○
Spectre-BTB (Variant 2) [52]	●	◐	○	○	○
Spectre-RSB (ret2spec) [54, 62]	◐	○	○	●	○
Spectre-STL (Variant 4) [32]	○	○	○	○	●

Glossary: Branch Target Buffer (BTB), Branch History Buffer (BHB), Pattern History Table (PHT), Return Stack Buffer (RSB), Store To Load (STL).

access but the attacker not. In practice, this implies that one or more phases of the transient execution attack flow in Figure 2 should be realized through so-called *code gadgets* executing within the victim application. We propose a novel taxonomy of gadgets based on these phases in Section 5.

For Meltdown-type attacks, on the other hand, transient execution allows to completely "melt down" architectural isolation barriers by computing on unauthorized results of faulting instructions. Meltdown thus transiently bypasses *hardware-enforced* security policies to leak data that should always remain architecturally inaccessible for the application. Where Spectre-type leakage remains largely an unintended side-effect of important speculative performance optimizations, Meltdown reflects a failure of the CPU to respect hardware-level protection boundaries for transient instructions. That is, the mere continuation of the transient execution after a fault itself is required, but not sufficient for a successful Meltdown attack. As further explored in Section 6, this has profound consequences for defenses. Overall, mitigating Spectre requires careful hardware-software co-design, whereas merely replacing the data of a faulting instruction with a dummy value is sufficient to block Meltdown-type leakage in silicon (e.g., as it is done for Meltdown-US in AMD CPUs and recent Intel Whiskey Lake CPUs).

3 Spectre-type Attacks

In this section, we provide an overview of all known Spectre-type attacks (cf. Figure 1). Given the versatility of Spectre variants in a variety of adversary models, we propose a novel two-level taxonomy based on the preparatory phases of the abstract transient execution attack flow in Figure 2. First, we distinguish the different microarchitectural buffers that can trigger a prediction (phase 2), and second, the mistraining strategies that can be used to steer the prediction (phase 1).

Systematization of Spectre Variants. To predict the outcome of various types of branches and data dependencies, modern CPUs accumulate an extensive microarchitectural state across various internal buffers and components [24]. Table 1 overviews all known Spectre-type attacks and the corresponding microarchitectural elements they exploit. As the first level of our classification tree, we categorize Spectre attacks based on the microarchitectural root cause that triggers the misprediction leading to the transient execution:

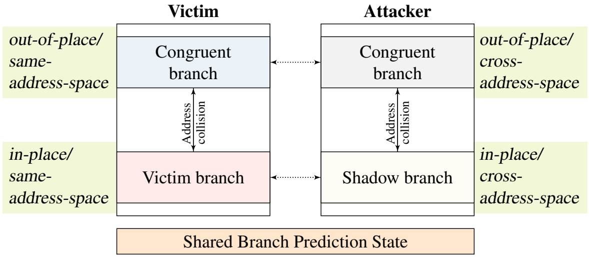
Victim
Attacker
out-of-place/ same-address-space
Congruent branch
Congruent branch
out-of-place/ cross-address-space
Address collision
Address collision
in-place/ same-address-space
Victim branch
Shadow branch
in-place/ cross-address-space
Shared Branch Prediction State

Figure 3: A branch can be mistrained either by the victim process (*same-address-space*) or by an attacker-controlled process (*cross-address-space*). Mistraining can be achieved either using the vulnerable branch itself (*in-place*) or a branch at a congruent virtual address (*out-of-place*).

- Spectre-PHT [52, 50] exploits the *Pattern History Table* (PHT) that predicts the outcome of conditional branches.
- Spectre-BTB [52] exploits the *Branch Target Buffer* (BTB) for predicting branch destination addresses.
- Spectre-RSB [62, 54] primarily exploits the *Return Stack Buffer* (RSB) for predicting return addresses.
- Spectre-STL [32] exploits memory disambiguation for predicting *Store To Load* (STL) data dependencies.

Note that NetSpectre [79], SGXSpectre [66], and SGXPectre [18] focus on applying one of the above Spectre variants in a specific exploitation scenario. Hence, we do not consider them separate variants in our classification.

Systematization of Mistraining Strategies. We now propose a second-level classification scheme for Spectre variants that abuse history-based branch prediction (*i.e.*, all of the above except Spectre-STL). These Spectre variants first go through a preparatory phase (cf. Figure 2) where the microarchitectural branch predictor state is "poisoned" to cause intentional misspeculation of a particular victim branch. Since branch prediction buffers in modern CPUs [52, 24] are commonly indexed based on the virtual address of the branch instruction, mistraining can happen either within the same address space or from a different attacker-controlled process. Furthermore, as illustrated in Figure 3, when only a subset of the virtual address is used in the prediction, mistraining can be achieved using a branch instruction at a congruent virtual address. We thus complete the field of Spectre-type branch poisoning attacks with 4 distinct mistraining strategies:

1. Executing the victim branch in the victim process (*same-address-space in-place*).
2. Executing a congruent branch in the victim process (*same-address-space out-of-place*).
3. Executing a shadow branch in a different process (*cross-address-space in-place*).
4. Executing a congruent branch in a different process (*cross-address-space out-of-place*).

In current literature, several of the above branch poisoning strategies have been overlooked for different Spectre variants. We summarize the results of an exhaustive vulnerability assessment under all mistraining strategies in Table 2. Our systematization thus reveals clear blind spots that allow an attacker to mistrain branch predictors in previously unknown ways. As explained further, depending on the adversary's capabilities (e.g., in-process, sandboxed, remote, enclave, etc.) these previously unknown mistraining strategies may lead to new attacks and/or bypass existing defenses.

3.1 Spectre-PHT (Input Validation Bypass)

Microarchitectural Element. Kocher et al. [52] first introduced Spectre Variant 1, an attack that poisons the Pattern History Table (PHT) to mispredict the direction (taken or not-taken) of conditional branches. Depending on the underlying microarchitecture, the PHT is accessed based on a combination of virtual address bits of the branch instruction plus a hidden Branch History Buffer (BHB) that accumulates global behavior for the last N branches [24, 23]

Reading Out-of-Bounds. Conditional branches are commonly used by programmers and/or compilers to maintain memory safety invariants at runtime. For example, consider the following code snippet for bounds checking [52]:

```
if (x < len(array1)) { y = array2[array1[x] * 4096]; }
```

At the architectural level, this program clearly ensures that the index variable x always lies within the bounds of the fixed-length buffer array1. However, after repeatedly supplying valid values of x, the PHT will reliably predict that this branch evaluates to true. When the adversary now supplies an invalid index x, the CPU continues along a mispredicted path and transiently performs an out-of-bounds memory access. The above code snippet features an explicit example of a "leak gadget" that may act as a microarchitectural covert channel: depending on the out-of-bounds value being read, the transient instructions load another memory page belonging to array2 into the cache.

Writing Out-of-Bounds. Kiriansky and Waldspurger [50] showed that transient writes are also possible by following the same principle. Consider the following code line:

```
if (x < len(array)) { array[x] = value; }
```

After mistraining the PHT component, attackers controlling the untrusted index x can transiently write to arbitrary out-of-bounds addresses. This creates a transient buffer overflow, allowing the attacker to bypass both type and memory safety. Ultimately, when repurposing traditional techniques from return-oriented programming [80] attacks, adversaries may even gain arbitrary code execution in the transient domain by overwriting return addresses or code pointers.

Overlooked Mistraining Strategies. All Spectre-PHT attacks so far [52, 66, 50] rely on a same-address-space in-place branch poisoning strategy. However, our results (cf. Table 2) reveal that all Intel, ARM, and AMD CPUs we tested are vulnerable to all four PHT mistraining strategies. In this, we are the first to successfully demonstrate Spectre-PHT-style branch misprediction attacks *without prior execution of the victim branch*. This is an important contribution as it may open up previously unknown attack avenues for restricted adversaries.

Table 2: Spectre-type attacks performed in-place, out-of-place, same-address-space, or cross-address-space.

	Method	Attack	Spectre-PHT	Spectre-BTB	Spectre-RSB	Spectre-STL
Intel	same-address-space	in-place	● [52, 50]	★	● [62]	● [32]
		out-of-place	★	● [18]	● [62, 54]	○
	cross-address-space	in-place	★	● [52, 18]	● [62, 54]	○
		out-of-place	★	● [52]	● [54]	○
ARM	same-address-space	in-place	● [52, 50]	★	● [6]	● [6]
		out-of-place	★	☆	● [6]	○
	cross-address-space	in-place	★	● [6, 52]	☆	○
		out-of-place	★	☆	☆	○
AMD	same-address-space	in-place	● [52]	★	★	● [32]
		out-of-place	★	☆	★	○
	cross-address-space	in-place	★	● [52]	★	○
		out-of-place	★	☆	★	○

Symbols indicate whether an attack is possible and known (●), not possible and known (○), possible and previously unknown or not shown (★), or tested and did not work and previously unknown or not shown (☆). All tests performed with no defenses enabled.

Cross-address-space PHT poisoning may, for instance, enable advanced attacks against a privileged daemon process that does not directly accept user input. Likewise, for Intel SGX technology, remote attestation schemes have been developed [81] to enforce that a victim enclave can only be run exactly once. This effectively rules out current state-of-the-art SGXSpectre [66] attacks that repeatedly execute the victim enclave to mistrain the PHT branch predictor. Our novel out-of-place PHT poisoning strategy, on the other hand, allows us to perform the training phase entirely *outside* the enclave by repeatedly executing a congruent branch in the untrusted enclave host process (cf. Figure 3).

3.2 Spectre-BTB (Branch Target Injection)

Microarchitectural Element. In Spectre Variant 2 [52], the attacker poisons the Branch Target Buffer (BTB) to steer the transient execution to a mispredicted branch target. For direct branches, the CPU indexes the BTB using a subset of the virtual address bits of the branch instruction to yield the predicted jump target. For indirect branches, CPUs use different mechanisms [33], which may take into account global branching history accumulated in the BHB when indexing the BTB. We refer to both types as Spectre-BTB.

Hijacking Control Flow. Contrary to Spectre-PHT, where transient instructions execute along a restricted mispredicted path, Spectre-BTB allows redirecting transient control flow to an arbitrary destination. Adopting established techniques from return-oriented programming (ROP) attacks [80], but abusing BTB poisoning instead of application-level vulnerabilities, selected code "gadgets" found in the victim address space may be chained together to construct arbitrary transient instruction sequences. Hence, where the success of Spectre-PHT critically relies on unintended leakage along the mispredicted code path, ROP-style gadget abuse in Spectre-BTB allows to more directly construct covert channels that expose secrets from the transient domain (cf. Figure 2). We discuss gadget types in more detail in Section 5.

Overlooked Mistraining Strategies. Spectre-BTB was initially demonstrated on Intel, AMD, and ARM CPUs using a cross-address-space in-place mistraining strategy [52]. With SGXPectre [18], Chen et al. extracted secrets from Intel SGX enclaves using either a cross-address-space in-place or same-address-space out-of-place BTB poisoning strategy. We experimentally reproduced these mistraining strategies through a systematic evaluation presented in Table 2. On AMD and ARM, we could not demonstrate out-of-place BTB poisoning. Possibly, these CPUs use an unknown (sub)set of virtual address bits which we were not able to reverse engineer.

We are the first to recognize that Spectre-BTB mistraining can also proceed by *repeatedly executing the vulnerable indirect branch with valid inputs*. Much like Spectre-PHT, such same-address-space in-place BTB poisoning abuses the victim's own execution to mistrain the underlying branch target predictor. Hence, as an important contribution to understanding attack surface and defenses, in-place mistraining *within* the victim domain may allow bypassing widely deployed mitigations [3, 43] that flush and/or partition the BTB before entering the victim. Since the branch destination address is now determined by the victim code and not under direct control of the attacker, however, Spectre-BTB-SA-IP cannot offer the full power of arbitrary transient control flow redirection. Yet, in higher-level languages like C++ that commonly rely on indirect branches to implement polymorph abstractions, Spectre-BTB-SA-IP may lead to subtle "speculative type confusion" vulnerabilities. For example, a victim that repeatedly executes a virtual function call with an object of `TypeA` may inadvertently mistrain the branch target predictor to cause misspeculation when finally executing the virtual function call with an object of another `TypeB`.

3.3 Spectre-RSB (Return Address Injection)

Microarchitectural Element. Maisuradze and Rossow [62] and Koruyeh et al. [54] introduced a new Spectre variant that exploits the Return Stack Buffer (RSB). The RSB is a small per-core microarchitectural buffer that stores the virtual addresses following the *N* most recent `call` instructions. When encountering a `ret` instruction, the CPU pops the topmost element from the RSB to predict the return flow.

Hijacking Return Flow. Misspeculation arises whenever the RSB layout diverges from the actual return addresses on the software stack. Such disparity for instance naturally occurs when restoring kernel/enclave/user stack pointers upon protection domain switches. Furthermore, same-address-space adversaries may explicitly overwrite return addresses on the software stack, or transiently execute `call` instructions which update the RSB without committing architectural effects [54]. This may allow untrusted code executing in a sandbox to transiently divert return control flow to interesting code gadgets outside of the sandboxed environment.

Due to the fixed-size nature of the RSB, a special case of misspeculation occurs for deeply nested function calls [54, 62]. Since the RSB can only store return addresses for the *N* most recent calls, an underfill occurs when the software stack is unrolled. In this case, the RSB can no longer provide accurate predictions. Starting from Skylake, Intel CPUs use the BTB as a fallback then [24, 54], thus allowing Spectre-BTB-style attacks triggered by `ret` instructions.

Overlooked Mistraining Strategies. Spectre-RSB is the only variant that has been demonstrated with all four mistraining strategies, but only on Intel [62, 54]. Our experimental results presented in Table 2 generalize these strategies to AMD CPUs. Furthermore, in line with ARM's own analysis [6], we successfully poisoned RSB entries within the same-address-space, but did not observe any cross-address-space leakage on ARM CPUs. We expect this may be a limitation of our current proof-of-concept code.

3.4 Spectre-STL (Speculative Store Bypass)

Microarchitectural Element. Speculation in modern CPUs is not restricted to control flow but also includes predicting dependencies in the data flow. A common type of Store To Load (STL) dependencies require that a memory load shall not be executed before all preceding stores that write to the same location have completed. However, even before the addresses of all prior stores in the pipeline are known, the CPUs' memory disambiguator [38, 2] may predict which loads can already be executed speculatively.

When the disambiguator predicts that a load does not have a dependency on a prior store, the load reads data from the L1 data cache. When the addresses of all prior stores are known, the prediction is verified. If any overlap is found, the load and all succeeding instructions are re-executed.

Reading Stale Values. Horn [32] showed how mispredictions by the memory disambiguator can be abused to speculatively bypass store instructions. Like previous attacks, Spectre-STL adversaries rely on an appropriate transient instruction sequence to leak unsanitized stale values via a microarchitectural covert channel. Furthermore, operating on stale pointer values may speculatively break type and memory safety guarantees in the transient execution domain [32].

4 Meltdown-type Attacks

This section overviews known Meltdown-type attacks, and presents a classification scheme that led to the discovery

Table 3: Demonstrated Meltdown-type attacks by their original names and the exception type or permission bit they exploit (●) or not (○). The systematic names are derived from the exception type (and permission bit) they exploit.

	Exception Type				Permission Bit					
Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
Variant 3a [10]	●	○	○	○						
Lazy FP [83]	○	●	○	○						
Meltdown-BR	○	○	●	○						
Meltdown [59]	○	○	○	●	●	○	○	○	○	○
Foreshadow [90]	○	○	○	●	○	●	○	●	○	○
Foreshadow-NG [93]	○	○	○	●	○	●	○	●	○	○
Meltdown-RW [50]	○	○	○	●	○	○	●	○	○	○
Meltdown-PK	○	○	○	●	○	○	○	○	○	●

Table 4: Secrets recoverable via Meltdown-type attacks and whether they cross the current privilege level (CPL).

Leaks Attack	Memory	Cache	Register	Cross-CPL
Meltdown-US (Meltdown) [59]	●	●	○	✓
Meltdown-P (Foreshadow) [90, 93]	◐	●	○	✓
Meltdown-GP (Variant 3a) [10]	○	○	●	✓
Meltdown-NM (Lazy FP) [83]	○	○	●	✓
Meltdown-RW (Variant 1.2) [50]	●	●	○	✗
Meltdown-PK	☆	★	☆	✗
Meltdown-BR	★	★	☆	✗

Symbols indicate whether an attack can leak secrets from a target (●) or not (○), respectively (★ and ☆) if we are the first to show it and whether it violates a security property (✓) or not (✗).

of two previously overlooked Meltdown variants (cf. Figure 1). Importantly, where Spectre-type attacks exploit (branch) misprediction events to trigger transient execution, Meltdown-type attacks rely on transient instructions following a CPU exception. Essentially, Meltdown exploits that exceptions are only raised (*i.e.*, become architecturally visible) upon the retirement of the faulting instruction. In some microarchitectures, this property allows transient instructions ahead in the pipeline to compute on unauthorized results of the instruction that is about to suffer a fault. The CPU's in-order instruction retirement mechanism takes care to discard any architectural effects of such computations, but as with the Spectre-type attacks above, secrets may leak through microarchitectural covert channels.

Systematization of Meltdown Variants. We introduce an extensible classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception that causes transient execution. Second, for page faults, we further categorize based on page-table entry protection bits (cf. Table 3). We also categorize attacks based on which storage locations can be reached, and whether it crosses a privilege boundary (cf. Table 4). Supporting the completeness of our systematization, we present several previously unknown Meltdown variants exploiting different exception types as well as page-table protection bits, including two exploitable ones. Our systematic analysis furthermore resulted in the first demonstration of exploitable Meltdown-type delayed exception handling effects on AMD CPUs.

4.1 Meltdown-US (Supervisor-only Bypass)

Modern CPUs commonly feature a "user/supervisor" page-table attribute to denote a virtual memory page as belonging to the OS kernel. The original Meltdown attack [59] reads kernel memory from user space on CPUs that do *not* transiently enforce the user/supervisor flag. In the trigger phase (cf. Figure 2) an unauthorized kernel address is dereferenced, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that for instance accesses a cache line based on the privileged data read by the trigger instruction. In the final phase, after the exception has been raised, the privileged data is reconstructed at the receiving end of the covert channel (e.g., Flush+Reload).

The attacks bandwidth can be improved by suppressing exceptions through transaction memory CPU features such as Intel TSX [35], exception handling [59], or hiding it in another transient execution [33, 59]. By iterating byte-by-byte over the kernel space and suppressing or handling exceptions, an attacker can dump the entire kernel. This includes the entire physical memory if the operating system has a direct physical map in the kernel. While extraction rates are significantly higher when the kernel data resides in the CPU cache, Meltdown has even been shown to successfully extract uncached data from memory [59].

4.2 Meltdown-P (Virtual Translation Bypass)

Foreshadow. Van Bulck et al. [90] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [34]. Unauthorized accesses to enclave memory usually do not raise a #PF exception but are instead silently replaced with abort page dummy values (cf. Section 6.2). In the absence of a fault, plain Meltdown cannot be mounted against SGX enclaves. To overcome this limitation, a Foreshadow attacker clears the "present" bit in the page-table entry mapping the enclave secret, ensuring that a #PF will be raised for subsequent accesses. Analogous to Meltdown-US, the adversary now proceeds with a transient instruction sequence to leak the secret (e.g., through a Flush+Reload covert channel).

Intel [39] named *L1 Terminal Fault* (L1TF) as the root cause behind Foreshadow. A terminal fault occurs when accessing a page-table entry with either the present bit cleared or a "reserved" bit set. In such cases, the CPU immediately aborts address translation. However, since the L1 data cache is indexed in parallel to address translation, the page table entry's physical address field (*i.e.*, frame number) may still be passed to the L1 cache. Any data present in L1 and tagged with that physical address will now be forwarded to the transient execution, regardless of access permissions.

Although Meltdown-P-type leakage is restricted to the L1 data cache, the original Foreshadow [90] attack showed how

SGX's secure page swapping mechanism may first be abused to prefetch arbitrary enclave pages into the L1 cache.

Foreshadow-NG. Foreshadow-NG [93] generalizes Foreshadow from the attack on SGX enclaves to bypass operating system or hypervisor isolation. The generalization builds on the observation that the physical frame number in a page-table entry is sometimes under direct or indirect control of an adversary. For instance, when swapping pages to disk, the kernel is free to use all but the present bit to store meta-data (e.g., the offset on the swap partition). However, if this offset is a valid physical address, any cached memory at that location leaks to an unprivileged Foreshadow-OS attacker.

Even worse is the Foreshadow-VMM variant, which allows an untrusted virtual machine, controlling guest-physical addresses, to extract the host machine's entire L1 data cache (including data belonging to the hypervisor or other virtual machines). The underlying problem is that a terminal fault in the guest page-tables early-outs the address translation process, such that guest-physical addresses are erroneously passed to the L1 data cache, without first being translated into a proper host physical address [39].

4.3 Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) allows an attacker to read privileged system registers. It was first discovered and published by ARM [10] and subsequently Intel [40] determined that their CPUs are also susceptible to the attack. Unauthorized access to privileged system registers (e.g., via `rdmsr`) raises a general protection fault (`#GP`). Similar to previous Meltdown-type attacks, however, the attack exploits that the transient execution following the faulting instruction can still compute on the unauthorized data, and leak the system register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.4 Meltdown-NM (FPU Register Bypass)

During a context switch, the OS has to save all the registers, including the floating point unit (FPU) and SIMD registers. These latter registers are large and saving them would slow down context switches. Therefore, CPUs allow for a lazy state switch, meaning that instead of saving the registers, the FPU is simply marked as "not available". The first FPU instruction issued after the FPU was marked as "not available" causes a device-not-available (`#NM`) exception, allowing the OS to save the FPU state of previous execution context before marking the FPU as available again.

Stecklina and Prescher [83] propose an attack on the above lazy state switch mechanism. The attack consists of three steps. In the first step, a victim performs operations loading data into the FPU registers. Then, in the second step, the CPU switches to the attacker and marks the FPU as "not available". The attacker now issues an instruction that uses the FPU, which generates an `#NM` fault. Before the faulting instruction retires, however, the CPU has already transiently executed the following instructions using data from the previous context. As such, analogous to previous Meltdown-type attacks, a malicious transient instruction sequence following the faulting instruction can encode the unauthorized FPU register contents through a microarchitectural covert channel (e.g., Flush+Reload).

4.5 Meltdown-RW (Read-only Bypass)

Where the above attacks [59, 90, 10, 83] focussed on stealing information across privilege levels, Kiriansky and Waldspurger [50] presented the first Meltdown-type attack that bypasses page-table based access rights *within* the current privilege level. Specifically, they showed that transient execution does not respect the "read/write" page-table attribute. The ability to transiently overwrite read-only data within the current privilege level can bypass software-based sandboxes which rely on hardware enforcement of read-only memory.

Confusingly, the above Meltdown-RW attack was originally named "Spectre Variant 1.2" [50]. Our systematization revealed, however, that the transient cause exploited above is clearly a `#PF` exception. Hence, this attack must be considered of Meltdown-type, but *not* a variant of Spectre.

4.6 Meltdown-PK (Protection Key Bypass)

Intel Skylake-SP server CPUs support memory-protection keys for user space (PKU) [37]. This feature allows processes to change the access permissions of a page directly from user space, *i.e.*, without requiring a syscall/hypercall. Thus, with PKU, user-space applications can implement efficient hardware-enforced isolation of trusted parts [89, 31].

We present a novel Meltdown-PK attack to bypass both read and write isolation provided by PKU. Meltdown-PK works if an attacker has code execution in the containing process, even if the attacker cannot execute the `wrpkru` instruction (e.g., blacklisting). Moreover, in contrast to cross-privilege level Meltdown attack variants, there is no software workaround. Meltdown-PK can be mitigated in hardware in future CPUs and possibly also in microcode.

Experimental Results. We tested Meltdown-PK on an Amazon EC2 C5 instance running Ubuntu 18.04 with PKU support. We created a memory mapping and used PKU to remove both read and write access. As expected, protected memory accesses produce a `#PF`. However, our proof-of-concept manages to leak the data via an adversarial transient instruction sequence with a Flush+Reload covert channel.

4.7 Meltdown-BR (Bounds Check Bypass)

To facilitate efficient software instrumentation, x86 CPUs come with dedicated hardware instructions that raise a bound

Table 5: CPU vendors vulnerable to Meltdown-type attacks.

Vendor \ Attack	Meltdown-US [59]	Meltdown-P [90, 93]	Meltdown-GP [10, 40]	Meltdown-NM [83]	Meltdown-RW [50]	Meltdown-PK	Meltdown-BR	Meltdown-DE	Meltdown-AC	Meltdown-UD	Meltdown-SS	Meltdown-XD	Meltdown-SM
Intel	●	●	●	●	●	★	★	☆	☆	☆	☆	☆	☆
ARM	●	○	●	_	●	_	_	☆	☆	☆	_	☆	☆
AMD	○	○	○	○	○	_	★	☆	☆	☆	☆	☆	☆

Symbols indicate whether at least one CPU model is vulnerable (filled) vs. no CPU is known to be vulnerable (empty). Glossary: reproduced (● vs. ○), first showed in this paper (★ vs. ☆), not applicable (_). All tests performed without defenses enabled.

range exceeded exception (#BR) when encountering out-of-bound array indices. The IA-32 ISA, for instance, defines a bound opcode for this purpose. While the bound instruction was omitted in the subsequent x86-64 ISA, modern Intel CPUs ship with Memory Protection eXtensions (MPX) for efficient array bounds checking.

Our systematic evaluation revealed that Meltdown-type effects of the #BR exception have not been thoroughly investigated yet. Specifically, Intel's analysis [43] only briefly mentions MPX-based bounds check bypass as a possibility, and recent defensive work by Dong et al. [21] highlights the need to introduce a memory lfence after MPX bounds check instructions. They classify this as a Spectre-type attack, implying that the lfence is needed to prevent the branch predictor from speculating on the outcome of the bounds check. According to Oleksenko et al. [67], neither bndcl nor bndcu exert pressure on the branch predictor, indicating that there is no prediction happening. Based on that, we argue that the classification as a Spectre-type attack is misleading as no prediction is involved. The observation by Dong et al. [21] indeed does not shed light on the #BR exception as the root cause for the MPX bounds check bypass, and they do not consider IA32 bound protection at all. Similar to Spectre-PHT, Meltdown-BR is a bounds check bypass, but instead of mistraining a predictor it exploits the lazy handling of the raised bound-range-exceeded exception.

Experimental Results. We introduce the Meltdown-BR attack which exploits transient execution following a #BR exception to encode out-of-bounds secrets that are never architecturally visible. As such, Meltdown-BR is an exception-driven alternative for Spectre-PHT. Our proofs-of-concept demonstrate out-of-bounds leakage through a Flush+Reload covert channel for an array index safeguarded by either IA32 bound (Intel, AMD), or state-of-the-art MPX protection (Intel-only). For Intel, we ran the attacks on a Skylake i5-6200U CPU with MPX support, and for AMD we evaluated both a E2-2000 and a Ryzen Threadripper 1920X. In this, we are the first to practically showcase a Meltdown-type transient execution attack exploiting delayed exception handling on AMD CPUs [4, 59].

4.8 Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. Following Intel's [35] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that all known Meltdown variants so far have exploited faults, but not traps or aborts. We consistently found no traces of transient execution beyond traps or aborts, which leads us to the hypothesis that Meltdown is only possible with faults (as they can occur at any moment during instruction execution). Table 5 and Figure 1 summarize experimental results for fault types tested on Intel, ARM, and AMD.

Division Errors. For the divide-by-zero experiment, we leveraged the signed division instruction (idiv on x86 and sdiv on ARM). On the ARMs we tested, there is no exception, but the division yields merely zero. On x86, the division raises a divide exception (#DE). Both on the AMD and Intel we tested, the CPU continues with the transient execution after the exception. In both cases, the result register is set to '0', which is the same result as on the tested ARM. Thus, Meltdown-DE is not possible, as no real values are leaked.

Supervisor Access. Although supervisor mode access prevention (SMAP) raises a page fault (#PF) when accessing user-space memory from the kernel, it seems to be free of any Meltdown effect. Thus, Meltdown-SM is not possible.

Alignment Faults. Upon detecting an unaligned memory operand, the CPU can (optionally) generate an alignment check exception (#AC). We found that the results of unaligned memory accesses never reach the transient execution. We suspect that this is because #AC is generated early-on (even before the operand's virtual address is translated to a physical one). Thus, Meltdown-AC is not possible.

Segmentation Faults. We consistently found that out-of-limit segment accesses never reach transient execution. We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a #GP or #SS exception, without sending the offending instruction to the ROB. Thus, Meltdown-SS is not possible.

Instruction Fetch. To yield a complete picture, we investigated Meltdown-type effects during the instruction fetch and decode phases. On all of our test systems, we did not succeed in transiently executing instructions residing in non-executable memory (*i.e.*, Meltdown-XD), or following an invalid opcode exception (*i.e.*, Meltdown-UD). We suspect that exceptions during instruction fetch or decode are immediately handled by the CPU, without first buffering the offending instruction in the ROB. Moreover, as invalid opcodes have an undefined length, the CPU does not even know where the next instruction starts. Hence, we suspect that invalid opcodes only leak if the microarchitectural effect is already an effect caused by the invalid opcode itself, not by subsequent transient instructions.

Table 6: Gadget classification according to the attack flow and whether executed by the attacker (●), victim (○), or either (◐).

Attack	1. Preface	2. Trigger example	3. Transient	5. Reconstruction
Covert channel [95, 1, 79]	◐ Flush/Prime/Evict	-	◐ Load/AVX/Port/...	◐ Reload/Probe/Time
Meltdown-US/RW/GP/NM/PK [59, 50, 10, 83]	● (Exception suppression)	● `mov/rdmsr/FPU`	● Controlled encode	● Exception handling
Meltdown-P [90, 93]	○ (L1 prefetch)	● `mov`	● Controlled encode	& controlled decode
Meltdown-BR	-	○ `bound/bndclu`	○ Inadvertent leak	— ” —
Spectre-PHT [52]	◐ PHT poisoning	○ `jz`	○ Inadvertent leak	● Controlled decode
Spectre-BTB/RSB [52, 18, 62, 54]	◐ BTB/RSB poisoning	○ `call/jmp/ret`	○ ROP-style encode	— ” —
Spectre-STL [32]	-	○ `mov`	○ Inadvertent leak	— ” —
NetSpectre [79]	○ Thrash/reset	○ `jz`	○ Inadvertent leak	○ Inadvertent transmit

Table 7: Spectre-type attacks on real-world software.

Attack	Gadgets	JIT	Description
Spectre-PHT [52]	2	✓	Chrome Javascript, Linux eBPF
Spectre-BTB [52]	2	✓/✗	Windows `ntdll`, Linux eBPF
Spectre-BTB [18]	336	✗	SGX SDK Intel/Graphene/Rust
Spectre-RSB [62]	1	✓	Firefox WebAssembly
Spectre-STL [32]	1	✓	Partial PoC on Linux eBPF

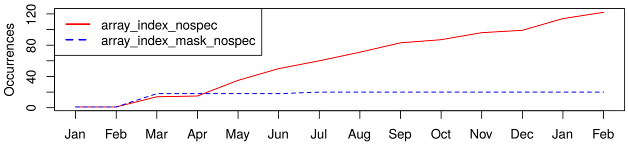

Figure 4: Evolution of Spectre v1 speculative load hardening patches in the Linux kernel over time (2018-2019).

5 Gadget Analysis and Classification

We deliberately oriented our attack tree (cf. Figure 1) on the microarchitectural root causes of the transient computation, abstracting away from the underlying covert channel and/or code *gadgets* required to successfully carry out the attack. In this section, we further dissect transient execution attacks by categorizing known gadget types and overviewing current results on their exploitability in real-world software.

Gadget Classification. We define a "gadget" as a series of instructions executed by either the attacker or the victim. Table 6 shows how all gadget types discussed in literature can be unambiguously assigned to one of the abstract attack phases from Figure 2. New gadgets can be added straightforwardly after determining their execution phase and objective.

Importantly, our classification table highlights that gadget choice largely depends on the attacker's capabilities. By plugging in different gadget types to compose the required attack phases, an almost boundless spectrum of adversary models can be covered. For local adversaries with arbitrary code execution (e.g., Meltdown-US [59]), all of the gadget functionality can be explicitly implemented by the attacker. For sandboxed adversaries (e.g., Spectre-PHT [52]), on the other hand, much of the gadget functionality has to be provided by "confused deputy" code executing in the victim domain. Ultimately, as demonstrated by NetSpectre [79], even remote adversaries can attack a fully isolated microarchitecture given that enough gadgets are found in the victim code to realize *each* of the individual attack phases.

Prevalence and Exploitability. While for Meltdown-type attacks, convincing real-world exploits have been developed to dump arbitrary process [59] and enclave [90] memory, most Spectre-type attacks have so far only been demonstrated in controlled environments. The most significant barrier to mounting a successful Spectre attack is to find exploitable gadgets in real-world software, which at present remains an important open research question in itself [62, 79].

To date, only 4 academic papers have demonstrated Spectre-type gadget exploitation in real-world software. Table 7 reveals that they either abuse ROP-style gadgets in larger code bases, or more commonly rely on Just-In-Time (JIT) compilation to indirectly provide the vulnerable gadget code. JIT compilers as commonly used in e.g., JavaScript, WebAssembly, or the eBPF Linux kernel interface, create a software-defined sandbox by extending untrusted attacker-provided code with runtime checks. However, the attacks in Table 7 demonstrate that such JIT checks can be transiently circumvented to leak memory contents outside of the sandbox. Furthermore, in the case of Spectre-BTB/RSB, even non-JIT compiled real-world code has been shown to be exploitable when the attacker controls sufficient inputs to the victim application. Kocher et al. [52] constructed a minimalist proof-of-concept that reads attacker-controlled inputs into registers before calling a function. Next, they rely on BTB poisoning to redirect transient control flow to a gadget they identified in the Windows `ntdll` library that allows leaking abitrary memory from the victim process. Likewise, Chen et al. [18] analyzed various trusted enclave runtimes for Intel SGX and found several instances of vulnerable branches with attacker-controlled input registers, plus numerous exploitable gadgets to which transient control flow may be directed to leak unauthorized enclave memory.

To further assess the prevalence of Spectre gadgets in real-world software, we selected the Linux kernel as a relevant case study of a major open-source project that underwent numerous Spectre-related security patches over the last year. To guide this effort, Linux kernel developers extended the Smatch static analysis tool to automatically discover potential Spectre-PHT-style out-of-bounds access gadgets [14]. Specifically, Smatch finds all instances of user-supplied array indices that have not been explicitly hardened. Unfortunately, Smatch's false positive rate is quite high. According to Carpenter [14], the tool reported 736 gadget candidates in April 2018, whereas the kernel only featured about 15 Spectre-resistant array in-

Table 8: Categorization of Spectre defenses and systematic overview of their microarchitectural target.

Defense	InvisiSpec [94]	SafeSpec [47]	DAWG [49]	Taint Tracking [52]	Timer Reduction [52]	RSB Stuffing [42]	Retpoline [88]	SLH [16, 22]	YSNB [68]	IBRS [3, 43]	STIPB [3, 43]	IBPB [3, 43]	Serialization [4, 40]	Sloth [50]	SSBD/SSBB [2, 43, 6]	Poison Value [74]	Index Masking [74]	Site Isolation [86]
Cache	●	●	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○
TLB	◐	●	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
BTB	○	○	○	○	○	○	●	○	○	●	●	●	○	○	○	○	○	○
BHB	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
PHT	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
RSB	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○
AVX	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FPU	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Execution Ports	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Category:	**C1**					**C2**										**C3**		

A defense considers the microarchitectural element (●), partially considers it or same technique possible for it (◐) or does not consider it at all (○).

dices at that time. We analyzed the number of occurrences of the newly introduced `array_index_nospec` and `array_index_mask_nospec` macros in the Linux kernel per month. Figure 4 shows that the number of Spectre-PHT patches has been continuously increasing over the past year. Providing further evidence that patching Spectre gadgets in real-world software is an ongoing effort and that automated detection methods pose an important research challenge.

6 Defenses

In this section, we discuss proposed defenses in software and hardware for the known Spectre and Meltdown variants. We propose a classification scheme for defenses based on their attempt to stop leakage, similar to Miller [65]. Our work differs from Miller in three points. First, ours extends to newer transient execution attacks. Second, we consider Meltdown and Spectre as two problems with different root causes, leading to a different classification. Third, it helped uncover problems that were not clear with the previous classification.

We categorize Spectre-type defenses into three categories:

C1: Mitigating or reducing the accuracy of covert channels used to extract the secret data.

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

C3: Ensuring that secret data cannot be reached.

Table 8 lists all proposed defenses against Spectre-type attacks and assigns them to the category they belong.

We categorize Meltdown-type defenses into two categories:

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

D2: Preventing the occurrence of faults.

6.1 Defenses for Spectre

C1: Mitigating or reducing accuracy of covert channels. Transient execution attacks use a covert channel to transfer a microarchitectural state change induced by the transient instruction sequence to the architectural level. One approach in mitigating Spectre-type attacks is reducing the accuracy of covert channels or preventing them.

Hardware. One enabler of transient execution attacks is that the transient execution sequence introduces a microarchitectural state change the receiving end of the covert channel observes. To secure CPUs, SafeSpec [47] introduces shadow hardware structures used during transient execution. Thereby, any microarchitectural state change can be squashed if the prediction of the CPU was incorrect. While their prototype implementation protects only caches (and the TLB), other channels, e.g., DRAM buffers [73], or execution unit congestion [59, 1], remain open.

Yan et al. [94] proposed InvisiSpec, a method to make transient loads invisible in the cache hierarchy. By using a *speculative buffer*, all transiently executed loads are stored in this buffer instead of the cache. Similar to SafeSpec, the buffer is invalidated if the prediction was incorrect. However, if the prediction was correct, the content of the buffer is loaded into the cache. For data coherency, InvisiSpec compares the loaded value during this process with the most recent, up-to-date value from the cache. If a mismatch occurs, the transient load and all successive instructions are reverted. Since InvisSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [49] securely partition the cache across its ways. With protection domains that isolate on a cache hit, cache miss and metadata level, cache-based covert channels are mitigated. This does not only require changes to the cache and adaptions to the coherence protocol but also enforces the correct management of these domains in software. Kocher et al. [52] proposed to limit data from entering covert channels through a variation of taint tracking. The idea is that the CPU tracks data loaded during transient execution and prevents their use in subsequent operations.

Software. Many covert channels require an accurate timer to distinguish microarchitectural states, e.g., measuring the memory access latency to distinguish between a cache hit and cache miss. With reduced timer accuracy an attacker cannot distinguish between microarchitectural states any longer, the receiver of the covert channel cannot deduce the sent information. To mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [64, 74, 85, 92]. However, Schwarz et al. [78] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [76]. While Chrome initially disabled `SharedArrayBuffers` in response to Meltdown and Spectre [85], this timer source has been re-enabled with the introduction of site-isolation [82].

NetSpectre requires different strategies due to its remote nature. Schwarz et al. [79] propose to detect the attack using DDoS detection mechanisms or adding noise to the network latency. By adding noise, an attacker needs to record more traces. Adding enough noise makes the attack infeasible as the amount of traces required becomes too large.

C2: Mitigating or aborting speculation if data is potentially accessible during transient execution.

Since all Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [52, 84]. As the loss of performance for commodity computers and servers would be too drastic, another proposal is to disable speculation only while processing secret data.

Hardware. A building blocks for some variants of Spectre is branch poisoning (an attacker mistrains a prediction mechanism, cf. Section 3). To deal with mistraining, both Intel and AMD extended the instruction set architecture (ISA) with a mechanism for controlling indirect branches [3, 43]. The proposed addition to the ISA consists of three controls:

- Indirect Branch Restricted Speculation (IBRS) prevents indirect branches executed in privileged code from being influenced by those in less privileged code. To enforce this, the CPU enters the IBRS mode which cannot be influenced by any operations outside of it.
- Single Thread Indirect Branch Prediction (STIBP) restricts sharing of branch prediction mechanisms among code executing across hyperthreads.
- The Indirect Branch Predictor Barrier (IBPB) prevents code that executes before it from affecting the prediction of code following it by flushing the BTB.

For existing ARM implementations, there are no generic mitigation techniques available. However, some CPUs implement specific controls that allow invalidating the branch predictor which should be used during context switches [6]. On Linux, those mechanisms are enabled by default [48]. With the ARMv8.5-A instruction set [9], ARM introduces a new barrier (`sb`) to limit speculative execution on following instructions. Furthermore, new system registers allow to restrict speculative execution and new prediction control instructions prevent control flow predictions (`cfp`), data value prediction (`dvp`) or cache prefetch prediction (`cpp`) [9].

To mitigate Spectre-STL, ARM introduced a new barrier called `SSBB` that prevents a load following the barrier from bypassing a store using the same virtual address before it [6]. For upcoming CPUs, ARM introduced Speculative Store Bypass Safe (SSBS); a configuration control register to prevent the re-ordering of loads and stores [6]. Likewise, Intel [43] and AMD [2] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

As an academic contribution, plausible hardware mitigations have furthermore been proposed [50] to prevent transient computations on out-of-bounds writes (Spectre-PHT).

Software. Intel and AMD proposed to use serializing instructions like `lfence` on both outcomes of a branch [4, 40]. ARM introduced a full data synchronization barrier (`DSB SY`) and an instruction synchronization barrier (ISB) that can be used to prevent speculation [6]. Unfortunately, serializing every branch would amount to completely disabling branch prediction, severely reducing performance [40]. Hence, Intel further proposed to use static analysis [40] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [71] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [69] use a similar approach using static analysis. Kocher [51] showed that this approach misses many gadgets that can be exploited.

Serializing instructions can also reduce the effect of indirect branch poisoning. By inserting it before the branch, the pipeline prior to it is cleared, and the branch is resolved quickly [4]. This, in turn, reduces the size of the speculation window in case that misspeculation occurs.

While `lfence` instructions stop speculative execution, Schwarz et al. [79] showed they do not stop microarchitectural behaviors happening before execution. This, for instance, includes powering up the AVX functional units, instruction cache fills, and iTLB fills which still leak data.

Evtyushkin et al. [23] propose a similar method to serializing instructions, where a developer annotates potentially leaking branches. When indicated, the CPU should not predict the outcome of these branches and thus stop speculation.

Additionally to the serializing instructions, ARM also introduced a new barrier (`CSDB`) that in combination with conditional selects or moves controls speculative execution [6].

Speculative Load Hardening (SLH) is an approach used by LLVM and was proposed by Carruth [16]. Using this idea, loads are checked using branchless code to ensure that they are executing along a valid control flow path. To do this, they transform the code at the compiler level and introduce a data dependency on the condition. In the case of misspeculation, the pointer is zeroed out, preventing it from leaking data through speculative execution. One prerequisite for this approach is hardware that allows implementation of a branchless and unpredicted conditional update of a register's value. As of now, the feature is only available in LLVM for x86 as the patch for ARM is still under review. GCC adopted the idea of SLH for their implementation, supporting both x86 and ARM. They provide a builtin function to either emit a speculation barrier or return a safe value if it determines that the instruction is transient [22].

Oleksenko et al. [68] propose an approach similar to Carruth [16]. They exploit that CPUs have a mechanism to detect data dependencies between instructions and introduce such a dependency on the comparison arguments. This ensures that the load only starts when the comparison is either in registers or the L1 cache, reducing the speculation window to a non-exploitable size. They already note that their approach is highly dependent on the ordering of instructions as the CPU might perform the load before the comparison. In that case, the attack would still be possible.

Google proposes a method called *retpoline* [88], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning. This method ensures that return instructions always speculate into an endless loop through the RSB. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [42] notes that in future CPUs that have Control-flow Enforcement Technology [36] (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future CPUs also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [42] to supersede retpoline.

On Skylake and newer architectures, Intel [42] proposes RSB stuffing to prevent an RSB underfill and the ensuing fallback to the BTB. Hence, on every context switch into the kernel, the RSB is filled with the address of a benign gadget. This behavior is similar to retpoline. For Broadwell and older architectures, Intel [42] provided a microcode update to make the `ret` instruction predictable, enabling retpoline to be a robust defense against Spectre-BTB.

C3: Ensuring that secret data cannot be reached. Different projects use different techniques to mitigate the problem of Spectre. WebKit employs two such techniques to limit the access to secret data [74]. WebKit first replaces array bound checks with index masking. By applying a bit mask, WebKit cannot ensure that the access is always in bounds, but introduces a maximum range for the out-of-bounds violation. In the second strategy, WebKit uses a pseudo-random *poison value* to protect pointers from misuse. Using this approach, an attacker would first have to learn the poison value before he can use it. The more significant impact of this approach is that mispredictions on the branch instruction used for type checks results in the wrong type being used for the pointer.

Google proposes another defense called *site isolation* [86], which is now enabled in Chrome by default. Site isolation executes each site in its own process and therefore limits the amount of data that is exposed to side-channel attacks. Even in the case where the attacker has arbitrary memory reads, he can only read data from its own process.

Kiriansky and Waldspurger [50] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [35]. They note that by using Spectre-PHT an attacker can first disable the protection before reading the data. To prevent this, they propose to include an `lfence` instruction in `wrpkru`, an instruction used to modify protection keys.

6.2 Defenses for Meltdown

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.

The fundamental problem of Meltdown-type attacks is that the CPU allows the transient instruction stream to compute on architecturally inaccessible values, and hence, leak them. By assuring that on a fault the execution does not continue or respectively does not continue with the otherwise inaccessible value, such attacks can be mitigated in future hardware designs. However, mitigations for existing microarchitectures are necessary, either through microcode updates, or operating-system-level software workarounds. These approaches aim to keep architecturally inaccessible data also inaccessible at the microarchitectural level.

Gruss et al. originally proposed KAISER [27, 28] to mitigate side-channel attacks defeating KASLR. However, it also defends against Meltdown-US attacks by preventing kernel secrets from being mapped in user space. Besides its performance impact, KAISER has one practical limitation [59, 27]. For x86, some privileged memory locations must always be mapped in user space. KAISER is implemented in Linux as kernel page-table isolation (KPTI) [61] and has also been backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [45] and Mac OS X and iOS have similar patches [44].

For Meltdown-GP, where the attacker leaks the contents of system registers that are architecturally not accessible in its current privilege level, Intel released microcode updates [40]. While AMD is not susceptible [5], ARM incorporated mitigations in future CPU designs and suggests to substitute the register values with dummy values on context switches for CPUs where mitigations are not available [6].

Preventing the access-control race condition exploited by Foreshadow and Meltdown may not be feasible with microcode updates [90]. Thus, Intel proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current CPUs [39, 93]. First, to maintain process isolation, the operating system has to sanitize the physical address field of unmapped page-table entries. The kernel clears the physical address field, or set it to non-existent physical memory. In case of the former, Intel suggests placing 4 KB dummy data at physical address `0`, and clearing the PS bit in page tables, preventing attackers from exploiting huge pages.

For SGX enclaves or hypervisors, which cannot trust the address translation performed by an untrusted OS, Intel proposes to either store secrets in uncacheable memory (as specified in the PAT or the MTRRs), or flush the L1 data cache when switching protection domains. With recent microcode updates, L1 is automatically flushed upon enclave exit, and hypervisors can additionally flush L1 before handing over control to an untrusted virtual machine. Flushing the cache is also done upon exiting System Management Mode (SMM) to mitigate Foreshadow-NG attacks on SMM.

To mitigate attacks across logical cores, Intel supplied a microcode update to ensure that different SGX attestation keys are derived when hyperthreading is enabled or disabled. To ensure that no non-SMM software runs while data belonging to SMM are in the L1 data cache, SMM software must rendezvous all logical cores upon entry and exit. According to Intel, this is expected to be the default behavior for most SMM software [39]. To protect against Foreshadow-NG attacks when hyperthreading is enabled, the hypervisor must ensure that no hypervisor thread runs on a sibling core with an untrusted VM.

D2: Preventing the occurrence of faults. Since Meltdown-type attacks exploit delayed exception handling in the CPU, another mitigation approach is to prevent the occurrence of a fault in the first place. Thus, accesses which would normally fault, become (both architecturally and microarchitecturally) valid accesses but do not leak secret data.

One example of such behavior are SGX's abort page semantics, where accessing enclave memory from the outside returns -1 instead of faulting. Thus, SGX has inadvertent protection against Meltdown-US. However, the Foreshadow [90] attack showed that it is possible to actively provoke another fault by unmapping the enclave page, making SGX enclaves susceptible to the Meltdown-P variant.

Preventing the fault is also the countermeasure for Meltdown-NM [83] that is deployed since Linux 4.6 [60]. By replacing lazy switching with eager switching, the FPU is always available, and access to the FPU can never fault. Here, the countermeasure is effective, as there is no other way to provoke a fault when accessing the FPU.

6.3 Evaluation of Defenses

Spectre Defenses. We evaluate all defenses based on their capabilities of mitigating Spectre attacks. Defenses that require hardware modifications are only evaluated theoretically. In addition, we discuss which vendors have CPUs vulnerable to what type of Spectre- and Meltdown-type attack.

InvisiSpec, SafeSpec, and DAWG are similar in how they approach the problem. Unfortunately, they only consider a cache-based covert channel. An attacker can easily substitute the covert channel and once again leak data through it. Based on that, we do not consider these three techniques as a reliable defense. DAWG has the additional problem that it does not mitigate an attack like NetSpectre, simply because the leak and transmit gadget are in the same domain.

WebKit's poison value prevents Spectre-PHT-based attacks as during speculation the type is confused, making the secret inaccessible. Index masking is only a partial solution; it only limits how far beyond the bound an access is possible.

Site isolation still allows data leakage within the same process and is therefore not a full solution. With SLH, we were not able to observe any leakage, indicating that it successfully prevents Spectre-PHT-based attacks, although it is possible that our experiments were simply not able to bypass the mitigation. This does not hold for YSNB as we were still able to observe leakage after introducing a data dependency for the same reason that Oleksenko et al. [68] mention.

IBRS, STIBP, and IBPB are depended on the specific hardware and OS. As of Linux 4.19, enhanced IBRS supersedes retpoline. If it is not available, the kernel is protected by retpoline if compiled correspondingly. IBRS is only activated for firmware calls as retpoline has a lower performance impact and the kernel does not contain any indirect branches.

The IBPB support on Linux is incomplete as the BTB is not flushed for dumpable processes [19] using the proc filesystem. As the default behavior on Linux is to mark a process as dumpable, all processes that do not explicitly change this remain vulnerable to Spectre-BTB. On AMD, IBPB also flushes the RSB [4]. We were not able to verify IBPB, IBRS, and STIBP on AMD as our machine does not support them.

Also, on current systems including Linux Kernel 4.20, STIBP is not enabled [19]. There is a patch enabling it if three conditions are met [55]: The CPU has to be vulnerable to Spectre-BTB; hyperthreading must be supported and a sibling be online; and auto-selection of Spectre-BTB defenses must be enabled, *i.e.*, the default case. We verified whether a cross-address-space Spectre-BTB attack still works on a patched Linux system and did not observe any leakage, indicating that STIBP seems to work on Intel as long as IBPB is also enabled.

In our tests, RSB stuffing only proved to be a reasonable approach against Spectre-RSB from different processes. Otherwise, we are able to circumvent it.

To use SSBD in user space, the process to be protected must issue a `prctl` system call. If the kernel has been compiled with seccomp support, SSBD is enabled for all seccomp-enabled processes. Our tests showed that SSBD is a functional defense for Spectre-STL. We searched projects on GitHub but found none using this method except Linux kernels. As only few projects support seccomp, we conclude that SSBD is not commonly used. On ARM, we verfied that SSBB works if it is explicitly added by the developer before the data is used in the transient execution window.

Our experiments did not show any leakage after a bounds check in the presence of a serializing instruction on AMD, as opposed to observations on `lfence` on Intel [79]. For ARM, we also observed no leakage following a barrier instruction (`CSDB`) in combination with conditional selects or moves, but on some ARM implementations, we were able to leak data from a single memory access through the TLB after the `DSB SY+ISH` instructions. As a result, the static analysis approach of Microsoft and others is only a valid defense technique on ARM if a `CSDB` in combination with conditional selects or moves is emitted. As the observed leakage is only caused by one access and the common Spectre-PHT sequence consists of two loads, `DSB SY+ISH` still works in most cases. On AMD, `lfence` is not serializing by default. Instead, an MSR has to be set for the instruction to serialize [3].

Taint tracking [52] theoretically mitigates all forms of Spectre-type attacks as data that has been tainted cannot be used in a transient execution. Therefore, the data does not enter a covert channel and can subsequently not be leaked.

Reducing the accuracy of timers [52] is only a partial solution as Schwarz et al. [78] have shown that different methods can be used to generate a new, accurate timer. Additionally,

Table 9: Spectre-type defenses and what they mitigate.

Attack		InvisiSpec [94]	SafeSpec [47]	DAWG [49]	RSB Stuffing [42]	Retpoline [88]	Poison Value [74]	Index Masking [74]	Site Isolation [86]	SLH [16, 22]	YSNB [68]	IBRS [3, 43]	STIPB [3, 43]	IBPB [3, 43]	Serialization [4, 40]	Taint Tracking [52]	Timer Reduction [52]	Sloth [50]	SSBD/SSBB [2, 43, 6]
Intel	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	●	◐	◐	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
ARM	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
AMD	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	■	◧	◧	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◧	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

it only makes it harder for an attacker to get the information, but that can be circumvented by taking more measurements.

While the Sloth [50] family of defenses was initially proposed to mitigate Spectre-PHT attacks, we argue that they should also be able to theoretically mitigate Spectre-STL.

Meltdown Defenses. We verified whether we can still execute Meltdown-type attacks on a fully-patched system. On a Ryzen Threadripper 1920X, we were still able to execute Meltdown-BND. On an i5-6200U (Skylake), an i7-8700K (Coffee Lake), and an i7-8565U (Whiskey Lake), we were able to successfully run a Meltdown-MPX, Meltdown-BND, and Meltdown-RW attack. Additionally to those, we were also able to run a Meltdown-PK attack on an Amazon EC2 C5 instance (Skylake-SP). Our results indicate that current mitigations only prevent Meltdown-type attacks that do not cross the current privilege level. We also tested whether we can still successfully execute a Meltdown-US attack on an Intel Whiskey Lake CPU without KPTI enabled as Intel claims that it is no longer susceptible to it. Our results show that it is indeed no longer possible to mount such an attack.

6.4 Performance impact of countermeasures

There have been a number of reports on performance impacts of selected countermeasures. As there is no standard benchmark used it is hard to quantify and compare the performance impact of countermeasures. Some countermeasures, for instance InvisiSpec [94], require hardware modifications that are not available and it is therefore hard to verify the performance loss. We show the results of our analysis in Table 10.

Table 10: Reported performance impacts of countermeasures

Defense \ Impact	Performance Loss	Benchmark
InvisiSpec [94]	22% [94]	SPEC
SafeSpec [47]	3% (improvement) [47]	SPEC2017 on MARSSx86 [72]
DAWG [49]	2–12%, 1–15% [49]	PARSEC [12], GAPBS [11]
RSB Stuffing [42]	no reports	
Retpoline [88]	5–10% [15]	real-world workload servers
Site Isolation [86]	only memory overhead [86]	
SLH [16, 22]	36.4%, 29% [16]	Google microbenchmark suite
YSNB [68]	60% [68]	Phoenix [75]
IBRS [3, 43]	20–30% [87]	two sysbench 1.0.11 benchmarks
STIPB [3, 43]	30– 50% [56]	Rodinia OpenMP [17], DaCapo [13]
IBPB [3, 43]	no individual reports	
Serialization [4, 40]	62%, 74.8% [16]	Google microbenchmark suite
SSBD/SSBB [2, 43, 6]	2–8% [20]	SYSmark®2014 SE & SPEC integer
KAISER/KPTI [27]	0–2.6% [26]	system call rates [25]
L1TF mitigations [90]	-3–31% [41]	various SPEC

One observation is the large variance between different countermeasures, ranging from a 0% decrease up to 74.8%. Some countermeasures even seem to improve performance. One countermeasure that stands out with a huge decrease in performance is serialization and highlights the importance of speculative execution to improve CPU performance. Another interesting countermeasure is KPTI. While it was initially reported to have a huge impact on performance, recent work shows that the decrease is almost negligible on systems that support PCID [25]. To mitigate Spectre and Meltdown, current systems rely on a combination of countermeasures. To show the overall decrease on a Linux 4.19 kernel with the default mitigations enabled, Larabel [57] performed multiple benchmarks to determine the impact. One of those benchmarks was CompileBench, which is suitable to determine the performance loss. On Intel, the slowdown was 7-16% compared to a non-mitigated kernel, on AMD it was 3-4%.

Naturally, the question arises which countermeasures to enable. For most users, the risk of exploitation is low and default software mitigations as provided by Linux, Microsoft, or Apple likely are sufficient. This is likely the optimum between potential attacks and reasonable performance. For data centers, it is harder as it depends on the needs of their customers and one has to evaluate this on an individual basis.

7 Conclusion

Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. We presented a consistent and extensible systematization of transient execution attacks. Our systematization uncovered 6 (new) transient execution attacks (Spectre and Meltdown variants) which have been overlooked and have not been investigated so far. We demonstrated *all* these variants in practical proof-of-concept attacks and evaluated their applicability to Intel, AMD, and ARM CPUs. We also presented a short analysis and classification of gadgets as well as their prevalence in real-world software. We also systematically evaluated all defenses, discovering that some transient execution attacks are not successfully mitigated by the rolled out patches and others are not mitigated because they have been overlooked. Hence, we need to think about future defenses carefully and plan to mitigate attacks and variants that are yet unknown.

Acknowledgments

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This research received funding from the Research Fund KU Leuven, and Jo Van Bulck is supported by the Research Foundation – Flanders (FWO). Evtyushkin acknowledges the start-up grant from the College of William and Mary. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] ALDAYA, A. C., BRUMLEY, B. B., UL HASSAN, S., GARCÍA, C. P., AND TUVERI, N. Port contention for fun and profit, https://eprint.iacr.org/2018/1060 2018.

[2] AMD. AMD64 Technology: Speculative Store Bypass Disable, 2018. https://developer.amd.com/wp-content/resources/124441_AMD64_SpeculativeStoreBypassDisable_Whitepaper_final.pdf Revision 5.21.18.

[3] AMD. Software techniques for managing speculation on AMD processors, 2018.

[4] AMD. Software techniques for managing speculation on AMD processors, 2018. https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf Revison 7.10.18.

[5] AMD. Spectre mitigation update, https://www.amd.com/en/corporate/security-updates July 2018.

[6] ARM. Cache speculation side-channels, 2018. Version 2.4.

[7] ARM LIMITED. *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. ARM Limited, 2012.

[8] ARM LIMITED. *ARM Architecture Reference Manual ARMv8*. ARM Limited, 2013.

[9] ARM LIMITED. ARM A64 Instruction Set Architecture (Beta), https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf Sep 2018.

[10] ARM LIMITED. Vulnerability of speculative processors to cache timing side-channel mechanism, https://developer.arm.com/support/security-update 2018.

[11] BEAMER, S., ASANOVIC, K., AND PATTERSON, D. A. The GAP benchmark suite. *arXiv:1508.03619* (2015).

[12] BIENIA, C. *Benchmarking modern multiprocessors*. 2011.

[13] BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., ET AL. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (2006).

[14] CARPENTER, D. Smatch check for Spectre stuff, https://lwn.net/Articles/752409/ Apr. 2018.

[15] CARRUTH, C., https://reviews.llvm.org/D41723 Jan. 2018.

[16] CARRUTH, C. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation, https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html Mar. 2018.

[17] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *International Symposium on Workload Characterization* (2009).

[18] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv:1802.09085* (2018).

[19] CORBET, J. Strengthening user-space Spectre v2 protection, https://lwn.net/Articles/764209/ Sept. 2018.

[20] CULBERTSON, L. Addressing new research for side-channel analysis. Intel.

[21] DONG, X., SHEN, Z., CRISWELL, J., COX, A., AND DWARKADAS, S. Spectres, virtual ghosts, and hardware support. In *Workshop on Hardware and Architectural Support for Security and Privacy* (2018).

[22] EARNSHAW, R. Mitigation against unsafe data speculation (CVE-2017-5753), https://lwn.net/Articles/759438/ July 2018.

[23] EVTYUSHKIN, D., RILEY, R., ABU-GHAZALEH, N. C., ECE, AND PONOMAREV, D. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS'18* (2018).

[24] FOG, A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2016.

[25] GREGG, B. KPTI/KAISER Meltdown Initial Performance Regressions, http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html 2018.

[26] GRUSS, D., HANSEN, D., AND GREGG, B. Kernel isolation: From an academic idea to an efficient patch for every computer. *USENIX ;login* (2018).

[27] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAURICE, C., AND MANGARD, S. KASLR is Dead: Long Live KASLR. In *ESSoS* (2017).

[28] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[29] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[30] GÜLMEZOĞLU, B., INCI, M. S., EISENBARTH, T., AND SUNAR, B. A Faster and More Realistic Flush+Reload Attack on AES. In *Constructive Side-Channel Analysis and Secure Design* (2015).

[31] HEDAYATI, M., GRAVANI, S., JOHNSON, E., CRISWELL, J., SCOTT, M., SHEN, K., AND MARTY, M. Janus: Intra-process isolation for high-throughput data plane libraries, 2018.

[32] HORN, J. speculative execution, variant 4: speculative store bypass, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 2018.

[33] HORN, JANN. Reading privileged memory with a side-channel, https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html Jan. 2018.

[34] INTEL. Intel Software Guard Extensions (Intel SGX), https://software.intel.com/en-us/sgx 2016.

[35] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. Order Number 325384.

[36] Intel. Control-flow Enforcement Technology Preview, June 2017. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf Revision 2.0.

[37] Intel. Intel Xeon Processor Scalable Family Technical Overview, https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview Sept. 2017.

[38] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2017.

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

[40] Intel. Intel Analysis of Speculative Execution Side Channels , July 2018. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf Revision 4.0.

[41] Intel. Resources and Response to Side Channel L1 Terminal Fault, https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html Aug. 2018.

[42] Intel. Retpoline: A Branch Target Injection Mitigation, June 2018. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf Revision 003.

[43] Intel. Speculative Execution Side Channel Mitigations, May 2018. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf Revision 3.0.

[44] Ionescu, A. Twitter: Apple Double Map, https://twitter.com/aionescu/status/948609809540046849 2017.

[45] Ionescu, A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)., https://twitter.com/aionescu/status/930412525111296000 2017.

[46] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[47] Khasawneh, K. N., Koruyeh, E. M., Song, C., Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *arXiv:1806.05179* (2018).

[48] King, R. ARM: spectre-v2: harden branch predictor on context switches , https://patchwork.kernel.org/patch/10427513/ May 2018.

[49] Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., and Emer, J. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *Cryptology ePrint Archive: Report 2018/418* (May 2018).

[50] Kiriansky, V., and Waldspurger, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[51] Kocher, P. Spectre mitigations in microsoft's c/c++ compiler, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html 2018.

[52] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019).

[53] Kocher, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[54] Koruyeh, E. M., Khasawneh, K., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *WOOT* (2018).

[55] Kosina, Jiri. x86/speculation: Enable cross-hyperthread spectre v2 STIBP mitigation, https://lore.kernel.org/patchwork/patch/983954/ Sept. 2018.

[56] Larabel, M. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower, https://www.phoronix.com/scan.php?page=article&item=linux-420-bisect&num=1 Nov. 2018.

[57] Larabel, M. The performance cost of spectre / meltdown / foreshadow mitigations on linux 4.19, https://www.phoronix.com/scan.php?page=article&item=linux-419-mitigations&num=2 Aug. 2018.

[58] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[59] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security* (2018).

[60] Lutomirski, Andy. x86/fpu: Hard-disable lazy FPU mode, https://lkml.org/lkml/2018/6/14/509 June 2018.

[61] LWN. The current state of kernel page-table isolation, https://lwn.net/Articles/741878/ Dec. 2017.

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

[63] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., and Römer, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[64] Microsoft Edge Team. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, https://blogs.windows.com/msedgedev/2018/01/03/s Jan. 2018.

[65] Miller, M. Mitigating speculative execution side channel hardware vulnerabilities, https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating/ Mar. 2018.

[66] O'Keeffe, Dan and Muthukumaran, Divya and Aublin, Pierre-Louis and Kelbert, Florian and Priebe, Christian and Lind, Josh and Zhu, Huanzhou and Pietzuch, Peter. Spectre attack against SGX enclave, https://github.com/lsds/spectre-attack-sgx Jan. 2018.

[67] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *arXiv:1702.00719* (2017).

[68] Oleksenko, O., Trach, B., Reiher, T., Silberstein, M., and Fetzer, C. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *arXiv:1805.08506* (2018).

[69] Open Source Security Inc. Respectre: The state of the art in spectre defenses, https://www.grsecurity.net/respectre_announce.php Oct. 2018.

[70] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[71] Pardoe, A. Spectre mitigations in msvc, https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre/ 2018.

[72] Patel, A., Afram, F., Chen, S., and Ghose, K. Marss: a full system simulator for multicore x86 cpus. In *Design Automation Conference* (2011).

[73] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[74] Pizlo, F. What Spectre and Meltdown mean for WebKit, https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ Jan. 2018.

[75] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture (HPCA)* (2007).

[76] Schwarz, M., Lipp, M., and Gruss, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS* (2018).

[77] Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., and Mangard, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS* (2018).

[78] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC* (2017).

[79] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. Netspectre: Read arbitrary memory over network. *arXiv:1807.10535* (2018).

[80] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS* (2007).

[81] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[82] Smith, Ben. Enable SharedArrayBuffer by default on non-android, `https://chromium.googlesource.com/chromium/src/+/4dbb4407b8a64dd9463ae34b1e9c19475acc1128` Aug. 2018.

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

[84] SUSE. Security update for kernel-firmware, `https://www.suse.com/support/update/announcement/2018/suse-su-20180008-1/` 2018.

[85] The Chromium Project. `https://www.chromium.org/Home/chromium-security/ssca` Actions required to mitigate Speculative Side-Channel Attack techniques.

[86] The Chromium Projects. `http://www.chromium.org/Home/chromium-security/site-isolation` Site Isolation.

[87] Tkachenko, V. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu, `https://www.percona.com/blog/2018/01/23/20-30/` Jan. 2018.

[88] Turner, P. Retpoline: a software construct for preventing branch-target-injection, 2018.

[89] Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., and Druschel, P. ERIM: secure and efficient in-process isolation with memory protection keys. *arXiv:1801.06822* (2018).

[90] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium* (2018).

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

[92] Wagner, L. Mitigations landing for new class of timing attack, `https://blog.mozilla.org/security/2018/01/03/mitigations` Jan. 2018.

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

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

[95] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

A Consistency of the Naming Scheme

While our naming and classification scheme (cf. Figure 1) is based on the names of the microarchitectural elements and the exceptions found on modern x86 processors, this does not limit the generality or consistency of our systematization. Generally, microarchitectural elements which have equivalent functionality are equivalent in our classification scheme. Other microarchitectural elements with different functionality, e.g., other prediction mechanisms, can extend the given classification scheme. Exception names are typically specific to one architecture. However, ARM also has equivalent exceptions types, such as instruction aborts (formerly prefetch aborts) and data aborts which correspond to the class of page faults [7, 8]. Still, any exception which does not have a corresponding one in our classification scheme can be added in a consistent way by following the existing classification scheme up to the point where no alternative fits.

B Consistency of the Systematization

We can consistently classify all currently known Spectre and Meltdown attacks. Our classification is easily extensible if a new variant is discovered by answering the following three questions: (1) What is the cause of the transient execution? (2) Who/what is responsible? (3) Where does the adversary influence whoever/whatever is responsible? As we were able to do this for all currently known Meltdown- and Spectre-type attacks, we claim that our systematization is correct and consistent. Our decision is tree is easily extensible in case a new variant is discovered.

C Exception Mnemonics

Table 11: Exceptions and their corresponding mnemonic.

Exception	Description
`#NM`	Device Not Available
`#AC`	Alignment Check
`#DE`	Divide Error
`#PF`	Page Fault
`#UD`	Invalid Opcode
`#SS`	Stack-Segment Fault
`#BR`	Bound Range Exceeded
`#GP`	General Protection Fault