# 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

Research on *transient execution* attacks including Spectre and Meltdown showed that exception or branch misprediction events might 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). Both the industry and academia are now focusing on finding effective defenses for known issues. However, we only have limited insight on residual attack surface and the completeness of the proposed defenses.

In this paper, we present a systematization of transient execution attacks. Our systematization uncovers 6 (new) transient execution attacks that have been overlooked and not been investigated so far: 2 new exploitable Meltdown effects: Meltdown-PK (Protection Key Bypass) on Intel, and Meltdown-BND (Bounds Check Bypass) on Intel and AMD; and 4 new Spectre mistraining strategies. We evaluate the attacks in our classification tree through proof-of-concept implementations on 3 major CPU vendors (Intel, AMD, ARM). Our systematization yields a more complete picture of the attack surface and allows for a more systematic evaluation of defenses. Through this systematic evaluation, we discover that most defenses, including deployed ones, cannot fully mitigate all attack variants.

## 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* [50, 56, 85].

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 [50], Meltdown [56], and Foreshadow [85]. 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 [48]. 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 [45,47,50,91]. Other defenses rely on recent CPU features that have not yet been evaluated from a transient security perspective [84]. 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 systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our decision tree, transient execution attacks are 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 the 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 Spectre- and Meltdown-type attacks, advancing attack surface understanding, highlighting misclassifications, and revealing new attacks.
2. We provide a clear distinction between Meltdown/Spectre, required for designing effective countermeasures.
3. We categorize defenses and show that most, including deployed ones, cannot fully mitigate all attack variants.
4. We describe new branch mistraining strategies, highlighting the difficulty of eradicating Spectre-type attacks.

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

**Experimental Setup.** Unless noted otherwise, 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.

**Outline.** Section 2 provides background. We systematize Spectre in Section 3 and Meltdown in Section 4. We analyze and classify gadgets in Section 5 and defenses in Section 6. We discuss future work and conclude in Section 7.

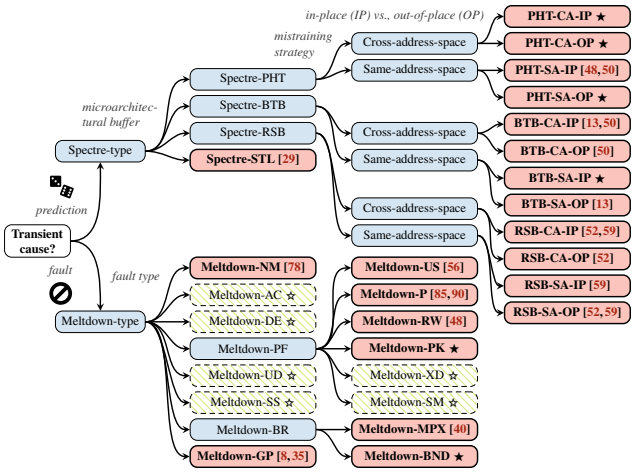



Figure 1: Transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).

## 2 Transient Execution

**Instruction Set Architecture and Microarchitecture.** The instruction set architecture (ISA) provides an interface between hardware and software. It defines the instructions that a processor supports, the available registers, the addressing mode, and describes the execution model. Examples of different ISAs are x86 and ARMv8. The microarchitecture then describes how the ISA is implemented in a processor in the form of pipeline depth, interconnection of elements, execution units, cache, branch prediction. The ISA and the microarchitecture are both stateful. In the ISA, this state includes, for instance, data in registers or main memory after a successful computation. Therefore, the architectural state can be observed by the developer. The microarchitectural state includes, for instance, entries in the cache and the translation lookaside buffer (TLB), or the usage of the execution units. Those microarchitectural elements are transparent to the programmer and can not be observed directly, only indirectly.

**Out-of-Order Execution.** On modern CPUs, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations ($\mu$OPs) 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 $\mu$OPs 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 $\mu$OP are available, and its corresponding execution unit is not busy, the CPU starts its execution even if $\mu$OPs earlier in the instruction stream have not finished yet. As immediate results are only made visible at the architectural level when all previous $\mu$OPs have finished, CPUs typically keep track of the status of $\mu$OPs in a so-called *Reorder Buffer* (ROB). The CPU takes care to *retire* $\mu$OPs in-order, deciding to either discard their results or commit them to the architectural state. For instance, exceptions and external interrupt requests are

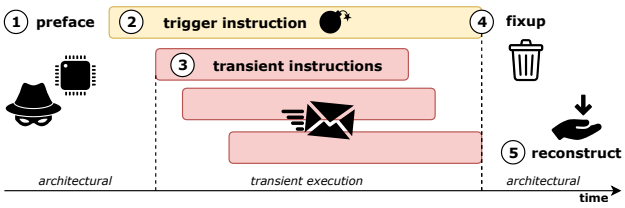



Figure 2: High-level overview of a transient execution attack in 5 phases: (1) prepare microarchitecture, (2) execute a *trigger instruction*, (3) *transient instructions* encode unauthorized data through a microarchitectural covert channel, (4) CPU retires trigger instruction and flushes transient instructions,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

handled during retirement by flushing any outstanding $\mu$OP results from the ROB. Therefore, the CPU may have executed so-called *transient instructions* [56], whose results are never committed to the architectural state.

**Speculative Execution.** Software is mostly not linear but contains (conditional) branches or data dependencies between 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 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 rollback 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 [24,51,60,67,92]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks, e.g., on cryptographic algorithms [26, 43, 92], user input [24, 55, 72], and kernel addressing information [23]. 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 many restrictions that exist at 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 [50, 56, 85]. This observation has led to a variety of transient execution attacks, which from a high-level 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 the 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.** 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 [50], exploit transient execution following control or data flow misprediction. Attacks of the second type, dubbed Meltdown [56], 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 [86]. 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,

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

| Attack \ Element | BTB | BHB | PHT | RSB | STL |
|---|---|---|---|---|---|
| Spectre-PHT (Variant 1) [50] | ○ | ◐ | ● | ○ | ○ |
| Spectre-PHT (Variant 1.1) [48] | ○ | ◐ | ● | ○ | ○ |
| Spectre-BTB (Variant 2) [50] | ● | ◐ | ○ | ○ | ○ |
| Spectre-RSB (ret2spec) [52, 59] | ◐ | ○ | ○ | ● | ○ |
| Spectre-STL (Variant 4) [29] | ○ | ○ | ○ | ○ | ● |

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

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 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 suffices to block Meltdown-type leakage in silicon, e.g., as it is done in AMD processors, or with the Rogue Data Cache Load resistance (RDCL_NO) feature advertised in recent Intel CPUs from Whiskey Lake onwards [40].

## 3 Spectre-type Attacks

In this section, we provide an overview of 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).

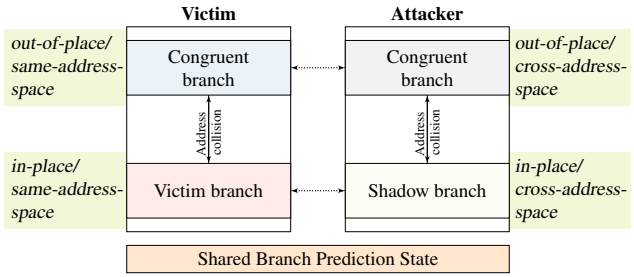



Figure 3: A branch can be mistrained either by the victim process (*same-address-space*) or by an attacker-controlled process (*cross-address-space*). Mistraining can be achieved either using the vulnerable branch itself (*in-place*) or a branch at a congruent virtual address (*out-of-place*).

**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 [19]. Table 1 overviews 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:

- Spectre-PHT [48, 50] exploits the *Pattern History Table* (PHT) that predicts the outcome of conditional branches.
- Spectre-BTB [50] exploits the *Branch Target Buffer* (BTB) for predicting branch destination addresses.
- Spectre-RSB [52, 59] primarily exploits the *Return Stack Buffer* (RSB) for predicting return addresses.
- Spectre-STL [29] exploits memory disambiguation for predicting *Store To Load* (STL) data dependencies.

Note that NetSpectre [74], SGXSpectre [63], and SGXPectre [13] 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 [19, 50] 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 enhance 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 [6, 13, 48, 50], several of the above branch poisoning strategies have been overlooked for different Spectre variants. We summarize the results of an assessment of vulnerabilities under 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.

Table 2: Spectre-type attacks performed in-place, out-of-place, same-address-space (*i.e.*, intra-process), or cross-address-space (*i.e.*, cross-process).

| | Method \ Attack | | Spectre-PHT | Spectre-BTB | Spectre-RSB | Spectre-STL |
|---|---|---|---|---|---|---|
| Intel | intra-process | in-place | ● [48, 50] | ★ | ● [59] | ● [29] |
| | | out-of-place | ★ | ● [13] | ● [52, 59] | ○ |
| | cross-process | in-place | ★ | ● [13, 50] | ● [52, 59] | ○ |
| | | out-of-place | ★ | ● [50] | ● [52] | ○ |
| ARM | intra-process | in-place | ● [48, 50] | ★ | ● [6] | ● [6] |
| | | out-of-place | ★ | ☆ | ● [6] | ○ |
| | cross-process | in-place | ★ | ● [6, 50] | ☆ | ○ |
| | | out-of-place | ★ | ☆ | ☆ | ○ |
| AMD | intra-process | in-place | ● [50] | ★ | ★ | ● [29] |
| | | out-of-place | ★ | ☆ | ★ | ○ |
| | cross-process | in-place | ★ | ● [50] | ★ | ○ |
| | | 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.

## 3.1 Spectre-PHT (Input Validation Bypass)

**Microarchitectural Element.** Kocher et al. [50] 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 on the same physical core [18, 19]

**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 [50]:

```
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 [48] 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 [75] attacks, adversaries may even gain arbitrary code execution in the transient domain by overwriting return addresses or code pointers.

**Overlooked Mistraining Strategies.** Spectre-PHT attacks so far [48, 50, 63] rely on a same-address-space in-place branch poisoning strategy. However, our results (cf. Table 2) reveal that the 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.

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 [76] to enforce that a victim enclave can only be run exactly once. This effectively rules out current state-of-the-art SGXSpectre [63] 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 on the same physical core 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 [50], 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 dif-

ferent mechanisms [28], 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 [75], 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 [50]. With SGXPectre [13], 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 or a function of bits which we were not able to reverse engineer. We encourage others to investigate whether a different (sub)set of virtual address bits is required to enable the attack.

To the best of our knowledge, 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 (Spectre-BTB-SA-IP) 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 [4, 40] 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 the 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 [59] and Koruyeh et al. [52] introduced a 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 [52]. 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 [52,59]. 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 [19,52], thus allowing Spectre-BTB-style attacks triggered by `ret` instructions.

**Overlooked Mistraining Strategies.** Spectre-RSB has been demonstrated with all four mistraining strategies, but only on Intel [52, 59]. 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 and encourage others to investigate this further.

### 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 [3, 33, 44] 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 following instructions are re-executed.

Table 3: Demonstrated Meltdown-type (MD) attacks.

| Attack | #GP | #NM | #BR | #PF | U/S | P | R/W | RSVD | XD | PK |
|---|---|---|---|---|---|---|---|---|---|---|
| MD-GP (Variant 3a) [8] | ● | ○ | ○ | ○ | | | | | | |
| MD-NM (Lazy FP) [78] | ○ | ● | ○ | ○ | | | | | | |
| MD-BR | ○ | ○ | ● | ○ | | | | | | |
| MD-US (Meltdown) [56] | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ |
| MD-P (Foreshadow) [85, 90] | ○ | ○ | ○ | ● | ○ | ● | ○ | ● | ○ | ○ |
| MD-RW (Variant 1.2) [48] | ○ | ○ | ○ | ● | ○ | ○ | ● | ○ | ○ | ○ |
| MD-PK | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ | ● |

Symbols (● or ○) indicate whether an exception type (left) or permission bit (right) is exploited. Systematic names are derived from what is exploited.

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

| Attack \ Leaks | Memory | Cache | Register | Cross-CPL |
|---|---|---|---|---|
| Meltdown-US (Meltdown) [56] | ● | ● | ○ | ✓ |
| Meltdown-P (Foreshadow-NG) [90] | ○ | ● | ○ | ✓ |
| Meltdown-P (Foreshadow-SGX) [85] | ◐ | ● | ◐ | ✓ |
| Meltdown-GP (Variant 3a) [8] | ○ | ○ | ● | ✓ |
| Meltdown-NM (Lazy FP) [78] | ○ | ○ | ● | ✓ |
| Meltdown-RW (Variant 1.2) [48] | ● | ● | ○ | ✗ |
| Meltdown-PK | ☆ | ★ | ☆ | ✗ |
| Meltdown-BR | ★ | ★ | ☆ | ✗ |

Symbols indicate whether an attack crosses a processor privilege level (✓) or not (✗), whether it can leak secrets from a buffer (●), only with additional steps (◐), or not at all (○). Respectively (★ vs. ☆) if first shown in this work.

**Reading Stale Values.** Horn [29] showed how mispredictions by the memory disambiguator could 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 [29].

## 4 Meltdown-type Attacks

This section overviews Meltdown-type attacks, and presents a classification scheme that led to the discovery 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 a classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception that causes transient execution. Following Intel's [31] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that Meltdown-type attacks so far have exploited faults, but not traps or aborts. The CPU generates faults if a correctable error has occurred, *i.e.*, they allow the program to continue after it has been resolved. Traps are reported immediately after the execution of the instruction, *i.e.*, when the instruction retires and becomes architecturally visible. Aborts report some unrecoverable error and do not allow a restart of the task that caused the abort.

In the second level, for page faults (`#PF`), 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). Through this systematization, we discovered several previously unknown Meltdown variants that exploit 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 [56] 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 [31], exception handling [56], or hiding it in another transient execution [28, 56]. 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 [56].

## 4.2 Meltdown-P (Virtual Translation Bypass)

**Foreshadow.** Van Bulck et al. [85] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [30]. 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 [34] 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 [85] attack showed how SGX's secure page swapping mechanism might first be abused to prefetch arbitrary enclave pages into the L1 cache, including even CPU registers stored on interrupt. This highlights that SGX's privileged adversary model considerably amplifies the transient execution attack surface.

**Foreshadow-NG.** Foreshadow-NG [90] 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 metadata (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 [34].

## 4.3 Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) [37] allows an attacker to read privileged system registers. It was first discovered and published by ARM [8] and subsequently Intel [35] 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 [78] 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 [8, 56, 78, 85] focussed on stealing information across privilege levels, Kiriansky and Waldspurger [48] 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" [48] as the authors followed a Spectre-centric naming scheme. Our systematization

revealed, however, that the transient cause exploited above is a #PF exception. Hence, this attack is 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) [32]. 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 [27, 84].

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. According to Intel [36], Meltdown-PK can be mitigated using address space isolation. Recent Meltdown-resistant Intel processors enumerating RDCL_NO plus PKU support furthermore mitigate Meltdown-PK in silicon. With those mitigations, the memory addresses that might be revealed by transient execution attacks can be limited.

**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-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 had not been thoroughly investigated yet. Specifically, Intel's analysis [40] only briefly mentions MPX-based bounds check bypass as a possibility, and recent defensive work by Dong et al. [16] 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. [64], 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. [16] 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 #BR exception.

Table 5: CPU vendors vulnerable to Meltdown (MD).

| Vendor \ Attack | MD-US [56] | MD-P [85, 90] | MD-GP [8, 35] | MD-NM [78] | MD-RW [48] | MD-PK | MD-BR | MD-DE | MD-AC | MD-UD | MD-SS | MD-XD | MD-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 shown in this paper (★ vs. ☆), not applicable (_). All tests performed without defenses enabled.

**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 an E2-2000 and a Ryzen Threadripper 1920X. This is the first experiment demonstrating a Meltdown-type transient execution attack exploiting delayed exception handling on AMD CPUs [4, 56].

## 4.8 Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. In our experiments, 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). Still, the possibility remains that our experiments failed and that they are possible. 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-by-zero* 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, according to our experiments Meltdown-DE is not possible, as no real values are leaked.

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 [1,74,92] | ◐ Flush/Prime/Evict | - | ◐ Load/AVX/Port/... | ◐ Reload/Probe/Time |
| Meltdown-US/RW/GP/NM/PK [8,48,56,78] | ● (Exception suppression) | ● `mov/rdmsr/FPU` | ● Controlled encode | ● Exception handling |
| Meltdown-P [85,90] | ○ (L1 prefetch) | ● `mov` | ● Controlled encode | & controlled decode |
| Meltdown-BR | - | ○ `bound/bndclu` | ○ Inadvertent leak | *same as above* |
| Spectre-PHT [50] | ◐ PHT poisoning | ○ `jz` | ○ Inadvertent leak | ● Controlled decode |
| Spectre-BTB/RSB [13,50,52,59] | ◐ BTB/RSB poisoning | ○ `call/jmp/ret` | ○ ROP-style encode | ● Controlled decode |
| Spectre-STL [29] | - | ○ `mov` | ○ Inadvertent leak | ● Controlled decode |
| NetSpectre [74] | ○ Thrash/reset | ○ `jz` | ○ Inadvertent leak | ○ Inadvertent transmit |

**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 in our experiments. Thus, we were not able to leak any data using Meltdown-SM in our experiments.

**Alignment Faults.** Upon detecting an unaligned memory operand, the CPU may generate an *alignment check* exception (`#AC`). In our tests, 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. Hence, our experiments with Meltdown-AC were unsuccessful in showing any leakage.

**Segmentation Faults.** We consistently found that out-of-limit segment accesses never reach transient execution in our experiments. We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a `#GP` or `#SS` (*stack-segment fault*) exception, without sending the offending instruction to the ROB. Therefore, we observed no leakage in our experiments with Meltdown-SS.

**Instruction Fetch.** To yield a complete picture, we investigated Meltdown-type effects during the instruction fetch and decode phases. On 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* (`#UD`) 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.

# 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 carry out the attack successfully. In this section, we further dissect transient execution attacks by categorizing gadget types in two tiers and overviewing current results on their exploitability in real-world software.

Table 7: Spectre-PHT gadget classification and the number of occurrences per gadget type in Linux kernel v5.0.

| Gadget | Example (Spectre-PHT) | #Occurrences |
|---|---|---|
| Prefetch | `if(i<LEN_A){a[i];}` | 172 |
| Compare | `if(i<LEN_A){if(a[i]==k){};}` | 127 |
| Index | `if(i<LEN_A){y = b[a[i]*x];}` | 0 |
| Execute | `if(i<LEN_A){a[i](void);}` | 16 |

## 5.1 Gadget Classification

**First-Tier: Execution Phase.** We define a "gadget" as a series of instructions executed by either the attacker or the victim. Table 6 shows how 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 that is only limited by the attacker's capabilities. For local adversaries with arbitrary code execution (e.g., Meltdown-US [56]), the gadget functionality can be explicitly implemented by the attacker. For sandboxed adversaries (e.g., Spectre-PHT [50]), on the other hand, much of the gadget functionality has to be provided by "confused deputy" code executing in the victim domain. Ultimately, as claimed by Schwarz et al. [74], even fully remote attackers may be able to launch Spectre attacks given that sufficient gadgets would be available inside the victim code.

**Second-Tier: Transient Leakage.** During our analysis of the Linux kernel (see Section 5.2), we discovered that gadgets required for Spectre-PHT can be further classified in a second tier. A second tier is required in this case as those gadgets enable different types of attacks. The first type of gadget we found is called *Prefetch*. A Prefetch gadget consists of a single array access. As such it is not able to leak data, but can be used to load data that can then be leaked by another gadget as was demonstrated by Meltdown-P [85]. The second type of gadget, called *Compare*, loads a value like in the Prefetch

gadget and then branches on it. Using a contention channel like execution unit contention [2, 9] or an AVX channel as claimed by Schwarz et al. [74], an attacker might be able to leak data. We refer to the third gadget as *Index* gadget and it is the double array access shown by Kocher et al. [50]. The final gadget type, called *Execute*, allows arbitrary code execution, similar to Spectre-BTB. In such a gadget, an array is indexed based on an attacker-controlled input and the resulting value is used as a function pointer, allowing an attacker to transiently execute code by accessing the array out-of-bounds. Table 7 gives examples for all four types.

## 5.2 Real-World Software Gadget Prevalence

While for Meltdown-type attacks, convincing real-world exploits have been developed to dump arbitrary process [56] and enclave [85] 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 [59, 74].

**Automated Gadget Analysis.** Since the discovery of transient execution attacks, researchers have tried to develop methods for the automatic analysis of gadgets. One proposed method is called oo7 [89] and uses taint tracking to detect Spectre-PHT Prefetch and Index gadgets. oo7 first marks all variables that come from an untrusted source as tainted. If a tainted variable is later on used in a branch, the branch is also tainted. The tool then reports a possible gadget if a tainted branch is followed by a memory access depending on the tainted variable. Guarnieri et al. [25] mention that oo7 would still flag code locations that were patched with Speculative Load Hardening [12] as it would still match the vulnerable pattern.

Another approach, called Spectector [25], uses symbolic execution to detect Spectre-PHT gadgets. It tries to formally prove that a program does not contain any gadgets by tracking all memory accesses and jump targets during execution along all different program paths. Additionally, it simulates the path of mispredicted branches for a number of steps. The program is run twice to determine whether it is free of gadgets or not First, it records a trace of memory accesses when no misspeculation occurs (*i.e.*, runs the program in its intended way). Second, it records a trace of memory accesses with misspeculation of a certain number of instructions. Spectector then reports a gadget if it detects a mismatch between the two traces. One problem with the Spectector approach is scalability as it is currently not feasible to symbolically execute large programs.

The Linux kernel developers use a different approach. They extended the Smatch static analysis tool to automatically discover potential Spectre-PHT out-of-bounds access gadgets [10]. 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 [10], the tool reported 736 gadget candidates in April 2018, whereas the kernel only featured about 15 Spectre-PHT-resistant array indices at that time. We further investigated this by analyzing 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. This provides further evidence that patching Spectre-PHT gadgets in real-world software is an ongoing effort and that automated detection methods and gadget classification pose an important research challenge.

Table 8: Spectre-type attacks on real-world software.

| Attack | Gadgets | JIT | Description |
|---|---|---|---|
| Spectre-PHT [50] | 2 | ✓ | Chrome Javascript, Linux eBPF |
| Spectre-BTB [50] | 2 | ✓/✗ | Linux eBPF, Windows `ntdll` |
| Spectre-BTB [13] | 336 | ✗ | SGX SDK Intel/Graphene/Rust |
| Spectre-BTB [9] | 690 | ✗ | OpenSSL, glibc, pthread, ... |
| Spectre-RSB [59] | 1 | ✓ | Firefox WebAssembly |
| Spectre-STL [29] | 1 | ✓ | Partial PoC on Linux eBPF |

**Academic Review.** To date, only 5 academic papers have demonstrated Spectre-type gadget exploitation in real-world software [9, 13, 29, 50, 59]. Table 8 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 the untrusted attacker-provided code with runtime checks. However, the attacks in Table 8 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. [50] 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 arbitrary memory from the victim process. Likewise, Chen et al. [13] 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. Bhattacharyya et al. [9] analyzed common software libraries that are likely to be linked against a victim program for gadgets. They were able to find numerous gadgets and were able to exploit one in OpenSSL to leak information.

**Case Study: Linux Kernel.** To further assess the prevalence of Spectre gadgets in real-world software, we selected the Linux kernel (Version 5.0) as a relevant case study of a major

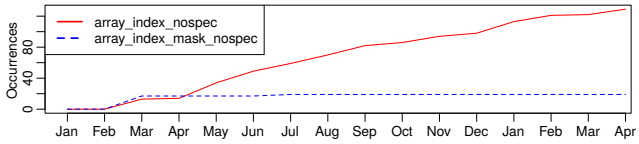



Figure 4: Evolution of Spectre-PHT patches in the Linux kernel over time (2018-2019).

open-source project that underwent numerous Spectre-related security patches over the last year. We opted for an in-depth analysis of one specific piece of software instead of a breadth-first approach where we do a shallow analysis of multiple pieces of software. This allowed us to analyse historical data (*i.e.*, code locations the kernel developers deemed necessary to protect) that led to the second tier classification discussed in Section 5.1.

There are a couple of reasons that make analysis difficult. The first is that Linux supports many different platforms. Therefore, particular gadgets are only available in a specific configuration. The second point is that the number of instructions that can be transiently executed depends on the size of the ROB [89]. As we analyze high-level code, we can only estimate how far ahead the processor can transiently execute.

Table 7 shows the number of occurrences of each gadget type from our second tier classification. While Figure 4 shows around 120 occurrences of `array_index_nospec`, the number of gadgets in our analysis is higher. The reason behind that is that multiple arrays are indexed with the same masked index as well as multiple branches that contain a value that was loaded with a potential malicious index. Our analysis also shows that more dangerous gadgets that either allow more than 1-bit leakage or even arbitrary code execution are not frequently occurring. Even if one is found, it might still be hard to exploit. During our analysis, we also discovered that in 13 locations the patch had been reverted, indicating that there is also some confusion with the kernel developers what needs to be fixed.

# 6 Defenses

In this section, we discuss proposed defenses in software and hardware for Spectre and Meltdown variants. We propose a classification scheme for defenses based on their attempt to stop leakage, similar to Miller [62]. 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.

Table 9: Categorization of Spectre defenses and systematic overview of their microarchitectural target.

| Microarchitectural Element | Defense | InvisiSpec | SafeSpec | DAWG | Taint Tracking | Timer Reduction | RSB Stuffing | Retpoline | SLH | Y-SNB | IBRS | STIBP | IBPB | Serialization | Sloth | SSBD/SSBB | Poison Value | Index Masking | Site Isolation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C1 | | | | | C2 | | | | | | | | | | C3 | | |
| Cache | | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| TLB | | ◐ | ● | ◐ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| BTB | | ○ | ○ | ○ | ○ | ○ | ○ | ● | ○ | ○ | ● | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| BHB | | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| PHT | | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| RSB | | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| AVX | | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| FPU | | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| Ports | | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

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

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

**C3**: Ensuring that secret data cannot be reached.

Table 9 lists 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 [45] 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 [69], or execution unit congestion [1, 9, 56], remain open.

Yan et al. [91] 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 InvisiSpec

only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [47] 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. [50] 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 [61, 70, 80, 88]. However, Schwarz et al. [73] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [71]. While Chrome initially disabled `SharedArrayBuffers` in response to Meltdown and Spectre [80], this timer source has been re-enabled with the introduction of site-isolation [77].

NetSpectre requires different strategies due to its remote nature. Schwarz et al. [74] 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 in practice as the amount of traces as well as the time required for averaging it out becomes too large [87].

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

Since Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [50, 79]. 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 [4, 40]. 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 [46]. With the ARMv8.5-A instruction set [7], 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`) [7].

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 [40] and AMD [3] provide Speculative Store Bypass Disable (SSBD) microcode updates that mitigate Spectre-STL.

As an academic contribution, plausible hardware mitigations have furthermore been proposed [48] 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, 35]. 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 [35]. Hence, Intel further proposed to use static analysis [35] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [68] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [66] use a similar approach using static analysis. Kocher [49] 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. [74] 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. [18] 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 [12]. 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 the 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 [17].

Oleksenko et al. [65] propose an approach similar to Carruth [12]. 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* [83], 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 [39] notes that in future CPUs that have Control-flow Enforcement Technology (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* [39].

On Skylake and newer architectures, Intel [39] 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 [39] provided a microcode update to make the `ret` instruction predictable, enabling retpoline to be a robust defense against Spectre-BTB. Windows has also enabled retpoline on their systems [14].

**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 [70]. 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* [81], 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 [48] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [31]. 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 execution does not continue with unauthorized data after a fault, such attacks can be mitigated directly in silicon. This design is enforced in AMD processors [4], and more recently also in Intel processors from Whiskey Lake onwards that enumerate RDCL_NO support [40]. 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 [22, 23] 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 [22, 56]. For x86, some privileged memory locations must always be mapped in user space. KAISER is implemented in Linux as kernel page-table isolation (KPTI) [58] and has also been backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [42] and Mac OS X and iOS have similar patches [41].

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 [35]. 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 [85]. Thus, Intel proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current CPUs [34, 90]. First, to maintain process isolation, the operating system has to sanitize the physical address field of unmapped page-table entries. The kernel either clears the physical address field, or sets it to non-existent physical memory. In the case of the former, Intel suggests placing 4 KB of dummy data at the start of the physical address space, and clearing the PS bit in page tables to prevent 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 [34]. 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 [85] 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 [78] that is deployed since Linux 4.6 [57]. 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.

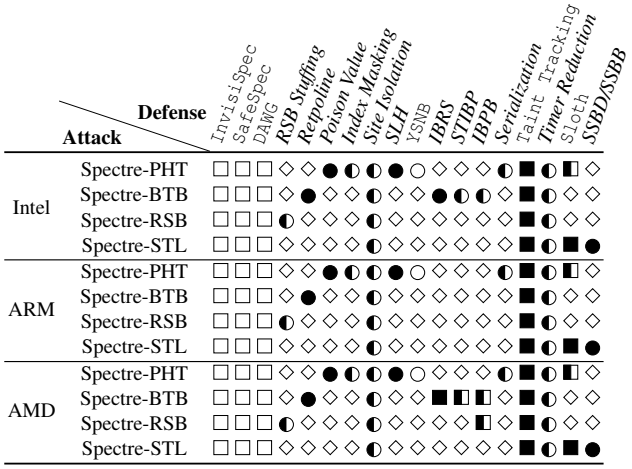
## 6.3 Evaluation of Defenses

**Spectre Defenses.** We evaluate 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. The results of our evaluation are shown in Table 10.

Several defenses only consider a specific covert channel (see Table 9), *i.e.*, they only try to prevent an attacker from recovering the data using a specific covert channel instead of targeting the root cause of the vulnerability. Therefore, they can be subverted by using a different one. As such, they can not be considered a reliable defense. Other defenses only limit the amount of data that can be leaked [70,81] or simply require more repetitions on the attacker side [74, 87]. Therefore, they are only partial solutions. RSB stuffing only protects a cross-process attack but does not mitigate a same-process attack. Many of the defenses are not enabled by default or depend on the underlying hardware and operating system [3, 4, 6, 40]. With serializing instructions [4, 6, 35] after a bounds check, we were still able to leak data on Intel and ARM (only with `DSB SY+ISH` instruction) through a single memory access and the TLB. On ARM, we observed no leakage following a `CSDB` barrier in combination with conditional selects or moves. We also observed no leakage with SLH, although the possibility remains that our experiment failed to bypass the mitigation. Taint tracking 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.

**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 cross the current privilege level. We also tested whether we can still successfully execute a Meltdown-US attack on a recent Intel Whiskey Lake CPU without KPTI enabled, as Intel claims these processors are no longer vulnerable. In our experiments, we were indeed not able to leak any data on such CPUs but encourage other researchers to further investigate newer processor generations.

Table 10: Spectre defenses and which attacks they mitigate.




Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇). Defenses in *italics* are production-ready, while `typeset` defenses are academic proposals.

## 6.4 Performance Impact of Countermeasures

There have been several reports on performance impacts of selected countermeasures. Some report the performance impact based on real-world scenarios (top of Table 11) while others use a specific benchmark that might not resemble real-world usage (lower part of Table 11). Based on the different testing scenarios, the results are hard to compare. To further complicate matters, some countermeasures require hardware modifications that are not available, and it is therefore hard to verify the performance loss.

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 [20]. 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 [54] performed multiple benchmarks to determine the impact. 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.

Table 11: Reported performance impacts of countermeasures. Top shows performance impact in real-world scenarios while the bottom shows it on a specific benchmark.

| Defense Evaluation | Penalty | Benchmark |
|---|---|---|
| KAISER/KPTI [21] | 0–2.6 % | System call rates |
| Retpoline [11] | 5–10 % | Real-world workload servers |
| Site Isolation [81] | 10–13 % | Memory overhead |
| InvisiSpec [91] | 22 % | SPEC |
| SafeSpec [45] | -3 % | SPEC on MARSSx86 |
| DAWG [47] | 1–15 % | PARSEC , GAPBS |
| SLH [12] | 29–36.4 % | Google microbenchmark suite |
| YSNB [65] | 60 % | Phoenix |
| IBRS [82] | 20–30 % | Sysbench 1.0.11 |
| STIBP [53] | 30–50 % | Rodinia OpenMP, DaCapo |
| Serialization [12] | 62–74.8 % | Google microbenchmark suite |
| SSBD/SSBB [15] | 2–8 % | SYSmark 2018, SPEC integer |
| L1TF Mitigations [38] | -3–31 % | SPEC |

## 7 Future Work and Conclusion

**Future Work.** For Meltdown-type attacks, it is important to determine where data is actually leaked from. For instance, Lipp et al. [56] demonstrated that Meltdown-US can not only leak data from the L1 data cache and main memory but even from memory locations that are explicitly marked as "uncacheable" and are hence served from the Line Fill Buffer (LFB). [1] In future work, other Meltdown-type attacks should be tested to determine whether they can also leak data from different microarchitectural buffers. In this paper, we presented a small evaluation of the prevalence of gadgets in real-world software. Future work should develop methods for automating the detection of gadgets and extend the analysis on a larger amount of real-world software. We have also discussed mitigations and shown that some of them can be bypassed or do not target the root cause of the problem. We encourage both offensive and defensive research that may use our taxonomy as a guiding principle to discover new attack variants, and develop mitigations that target the root cause of transient information leakage.

**Conclusion.** Transient instructions reflect unauthorized computations out of the program's intended code and/or data paths. We presented a 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 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

[1] The initial Meltdown-US disclosure (December 2017) and subsequent paper [56] already made clear that Meltdown-type leakage is *not* limited to the L1 data cache. We sent Intel a PoC leaking uncacheable-typed memory locations from a concurrent hyperthread on March 28, 2018. We clarified to Intel on May 30, 2018, that we attribute the source of this leakage to the LFB. In our experiments, this works identically for Meltdown-P (Foreshadow). This issue was acknowledged by Intel, tracked under CVE-2019-11091, and remained under embargo until May 14, 2019.

gadgets as well as their prevalence in real-world software. We also systematically evaluated 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

We want to thank the anonymous reviewers and especially our shepherd, Jonathan McCune, for their helpful comments and suggestions that substantially helped in improving the paper. 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 work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of 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, 2018.

[2] ALDAYA, A. C., BRUMLEY, B. B., UL HASSAN, S., GARCÍA, C. P., AND TUVERI, N. Port Contention for Fun and Profit. *ePrint 2018/1060* (2018).

[3] AMD. AMD64 Technology: Speculative Store Bypass Disable, 2018. Revision 5.21.18.

[4] AMD. Software Techniques for Managing Speculation on AMD Processors, 2018. Revison 7.10.18.

[5] AMD. Spectre mitigation update, July 2018.

[6] ARM. Cache Speculation Side-channels, 2018. Version 2.4.

[7] ARM LIMITED. ARM A64 Instruction Set Architecture, Sep 2018.

[8] ARM LIMITED. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism, 2018.

[9] BHATTACHARYYA, A., SANDULESCU, A., NEUGSCHWANDTNER, M., SORNIOTTI, A., FALSAFI, B., PAYER, M., AND KURMUS, A. Smotherspectre: exploiting speculative execution through port contention. *arXiv:1903.01843* (2019).

[10] CARPENTER, D. Smatch check for Spectre stuff, Apr. 2018.

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

[12] CARRUTH, C. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation), Mar. 2018.

[13] 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).

[14] CORP., M., https://support.microsoft.com/en-us/help/4482887/windows-10-update-kb4482887 Mar. 2019.

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

[16] 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).

[17] EARNSHAW, R. Mitigation against unsafe data speculation (CVE-2017-5753), July 2018.

[18] 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).

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

[20] GREGG, B. KPTI/KAISER Meltdown Initial Performance Regressions, 2018.

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

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

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

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

[25] GUARNIERI, M., KÖPF, B., MORALES, J. F., REINEKE, J., AND SÁNCHEZ, A. SPECTECTOR: Principled Detection of Speculative Information Flows. *arXiv:1812.08639* (2018).

[26] 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).

[27] 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.

[28] HORN, J. Reading privileged memory with a side-channel, Jan. 2018.

[29] HORN, J. speculative execution, variant 4: speculative store bypass, 2018.

[30] INTEL. Intel Software Guard Extensions (Intel SGX), 2016.

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

[32] INTEL. Intel Xeon Processor Scalable Family Technical Overview, Sept. 2017.

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

[34] INTEL. Deep Dive: Intel Analysis of L1 Terminal Fault, Aug. 2018.

[35] INTEL. Intel Analysis of Speculative Execution Side Channels , July 2018. Revision 4.0.

[36] INTEL. More Information on Transient Execution Findings, https://software.intel.com/security-software-guidance/insights/more-information-transient-execution-findings 2018.

[37] INTEL. Q2 2018 Speculative Execution Side Channel Update, May 2018.

[38] INTEL. Resources and Response to Side Channel L1 Terminal Fault, Aug. 2018.

[39] INTEL. Retpoline: A Branch Target Injection Mitigation, June 2018. Revision 003.

[40] INTEL. Speculative Execution Side Channel Mitigations, May 2018. Revision 3.0.

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

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

[43] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[44] ISLAM, S., MOGHIMI, A., BRUHNS, I., KREBBEL, M., GULMEZOGLU, B., EISENBARTH, T., AND SUNAR, B. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *arXiv:1903.00446* (2019).

[45] 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).

[46] KING, R. ARM: spectre-v2: harden branch predictor on context switches, May 2018.

[47] KIRIANSKY, V., LEBEDEV, I., AMARASINGHE, S., DEVADAS, S., AND EMER, J. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. *ePrint 2018/418* (May 2018).

[48] KIRIANSKY, V., AND WALDSPURGER, C. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).

[49] KOCHER, P. Spectre mitigations in Microsoft's C/C++ compiler, 2018.

[50] 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).

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

[52] KORUYEH, E. M., KHASAWNEH, K., SONG, C., AND ABU-GHAZALEH, N. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT* (2018).

[53] LARABEL, M. Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower, Nov. 2018.

[54] LARABEL, M. The performance cost of spectre / meltdown / foreshadow mitigations on linux 4.19, Aug. 2018.

[55] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[56] 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 Symposium* (2018).

[57] LUTOMIRSKI, A. x86/fpu: Hard-disable lazy FPU mode, June 2018.

[58] LWN. The current state of kernel page-table isolation, https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/ Dec. 2017.

[59] MAISURADZE, G., AND ROSSOW, C. ret2spec: Speculative execution using return stack buffers. In *CCS* (2018).

[60] 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).

[61] MICROSOFT. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, Jan. 2018.

[62] MILLER, M. Mitigating speculative execution side channel hardware vulnerabilities, Mar. 2018.

[63] O'KEEFFE, D., MUTHUKUMARAN, D., AUBLIN, P.-L., KELBERT, F., PRIEBE, C., LIND, J., ZHU, H., AND PIETZUCH, P. Spectre attack against SGX enclave, Jan. 2018.

[64] 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).

[65] 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).

[66] OPEN SOURCE SECURITY INC. Respectre: The state of the art in spectre defenses, Oct. 2018.

[67] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[68] PARDOE, A. Spectre mitigations in MSVC, 2018.

[69] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[70] PIZLO, F. What Spectre and Meltdown mean for WebKit, Jan. 2018.

[71] SCHWARZ, M., LIPP, M., AND GRUSS, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *NDSS* (2018).

[72] 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).

[73] 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).

[74] SCHWARZ, M., SCHWARZL, M., LIPP, M., AND GRUSS, D. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

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

[76] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[77] SMITH, B. Enable SharedArrayBuffer by default on non-android, Aug. 2018.

[78] STECKLINA, J., AND PRESCHER, T. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).

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

[80] THE CHROMIUM PROJECTS. Actions required to mitigate Speculative Side-Channel Attack techniques, 2018.

[81] THE CHROMIUM PROJECTS. Site Isolation, 2018.

[82] TKACHENKO, V. 20-30% Performance Hit from the Spectre Bug Fix on Ubuntu, Jan. 2018.

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

[84] 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).

[85] 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).

[86] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS* (2018).

[87] VARDA, K. WebAssembly's post-MVP future, https://news.ycombinator.com/item?id=18279791 2018.

[88] WAGNER, L. Mitigations landing for new class of timing attack, Jan. 2018.

[89] WANG, G., CHATTOPADHYAY, S., GOTOVCHITS, I., MITRA, T., AND ROYCHOUDHURY, A. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *arXiv:1807.05843* (2018).

[90] 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.

[91] 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).

[92] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).