A Systematic Evaluation of Transient Execution Attacks and Defenses

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

Abstract

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

In this paper, we present a sound and extensible systematization of transient execution attacks. Our systematization uncovers 7 (new) transient execution attacks that have been overlooked and not been investigated so far. This includes 2 new Meltdown variants: Meltdown-PK on Intel, and Meltdown-BR on Intel and AMD. It also includes 5 new Spectre mistraining strategies. We evaluate *all* 7 attacks in proof-of-concept implementations on 3 major processor vendors (Intel, AMD, ARM). Our systematization does not only yield a complete picture of the attack surface, but also allows a systematic evaluation of defenses. Through this systematic evaluation, we discover that we can still mount transient execution attacks that are supposed to be mitigated by rolled out patches.

1 Introduction

Processor 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 processors have deep pipelines allowing operations to be performed in parallel in different pipeline stages or different units of the execution stage. Many processors additionally have a mechanism which allows executing instructions not only in parallel but even out-of-order. These processors have a reordering element, which keeps track of all instructions and commits them in order, *i.e.*, there is no functional difference to regular in-order execution. Intuitively, instructions executed on in-order and out-of-order pipelines cannot be executed, if they have a dependency on a previous instruction which has not been executed (and committed) 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. Flushed instructions, those whose results are not made visible to the architectural level due to a roll-back, are called transient instructions [56, 50, 84]. On modern processors, virtually any instruction can raise a fault (e.g., page fault or general protection fault), requiring a roll-back. Already without prediction mechanisms, processors sometimes have to flush the pipeline, e.g., upon interrupts. With prediction mechanisms, there are more situations when partial pipeline flushes are necessary, namely on every misprediction. The pipeline flush reverts any architectural effects of instructions, ensuring functional correctness. Hence, the instructions are executed *transiently* (first they are, and then they vanish), *i.e.*, we call this *transient execution*.

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 [84]. These attacks exploit transient execution and encode secrets in the microarchitectural side effects (e.g., cache state) to transmit them (to the architectural level) to an attacker. The field of transient execution attacks emerged suddenly and grew rapidly, 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, e.g., , focus exclusively on hindering exploitation of a specific covert channel, instead of addressing the microarchitectural root cause of the leakage [47, 45, 88, 50]. Other defenses critically rely on state-of-the-art CPU features that have not yet been thoroughly evaluated from a transient security perspective [83]. We also debunk implicit assumptions including that AMD processors are immune to Meltdown-type effects, or that serializing instructions mitigate Spectre Variant 1 on any CPU.

In this paper, we present a sound and extensible systematization of transient execution attacks, *i.e.*, Spectre, Meltdown, Foreshadow, and related attacks. Using our universal decision tree, all known transient execution attacks were accurately classified through a universal and unambiguous naming scheme (cf. Figure 1). The hierarchical and extensible nature of our classification methodology allows to easily identify residual attack surface, leading to 7 new transient execution attacks (Spectre and Meltdown variants) that we describe in this work. These 7 new attacks have been overlooked and not been investigated so far. Two of the attacks are Meltdown-BR, 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 5 attacks are previously overlooked mistraining strategies for Spectre-PHT and Spectre-BTB attacks. We demonstrate *all* 7 attacks in practical proof-of-concept attacks on vulnerable code patterns and evaluate them on processors of Intel, ARM, and AMD.

We also 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 work. 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 provide a concise overview of all known transient execution attacks and defenses.
2. We systematize all transient execution attacks based on the element (Spectre-types) or the exception (Meltdown-types) they exploit, revealing clear gaps in the attack surface. Exploring these gaps, we identify 7 new attacks.
3. We systematize all defenses against transient execution attacks based on which precondition they try to eliminate, again revealing clear gaps. Hence, we find existing and new variants which are not mitigated by current defenses.

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

Responsible Disclosure. We responsibly disclosed our research to Intel, ARM, and AMD. Intel and ARM acknowledged our findings.

2 Transient Execution

Out-of-Order Execution. On modern processors, individual instructions of a complex instruction set are first decoded and split-up into simpler micro-operations (μOPs) that are then processed. This design decision allows for superscalar optimizations and to extend or modify the implementation of certain instructions through so-called microcode updates. Furthermore, to increase performance, CPU's usually implement a so-called *out-of-order* design. This allows the processor to execute μOPs not only in the sequential order provided by the instruction stream but to dispatch them in parallel, exhausting the CPUs execution units as much as possible and, thus, improving the overall performance. If the required operands of a μOP are available, and its corresponding execution unit is not busy, the processor starts its execution even if μOPs earlier in the instruction stream have not finished yet. As immediate results are only made visible on the architectural level if all previous μOPs have finished, CPUs typically keep track of the status of μOPs in a so-called *Reorder Buffer* (ROB). When μOPs finish their execution, they *retire* in-order, and their results are committed to the architectural state. Furthermore, exceptions and interrupts that occurred during the execution of the μOP are handled during the retirement. Therefore, it is possible that the CPU executed instructions whose results are never committed to the architectural state. These instructions are called *transient instructions* [56].

Speculative Execution. Complex software is mostly not linear but instead contains (conditional) branches or data dependencies between instructions. In theory, the processor would have to stall until a branch or dependencies are resolved before it can continue the execution. As stalling decreases the performances significantly, processors deploy various mechanisms to predict the outcome of a branch or a data dependency. Thus, processors continue executing along the predicted path, buffering the results again in the ROB until the correctness of the prediction is verified as its dependencies are resolved. In the case of a correct prediction, the processor can retire the precomputed results from the reorder buffer, increasing the overall performance. However, if the prediction was incorrect, the processor needs to squash the pre-computed results and perform a roll-back to the last correct state by flushing the pipeline and the ROB. Thus, the processor also executed transient instructions, which are never committed to the architectural state.

Transient Execution Attacks. While the execution of transient instructions does not influence the architectural state and, thus, cannot be observed architecturally, the microarchitectural state can change. Transient execution attacks exploit these microarchitectural state changes to extract sensitive information by transferring the microarchitectural state into an architectural state. Usually, these attacks use the CPU's cache, as changes in it can be observed rather eas-

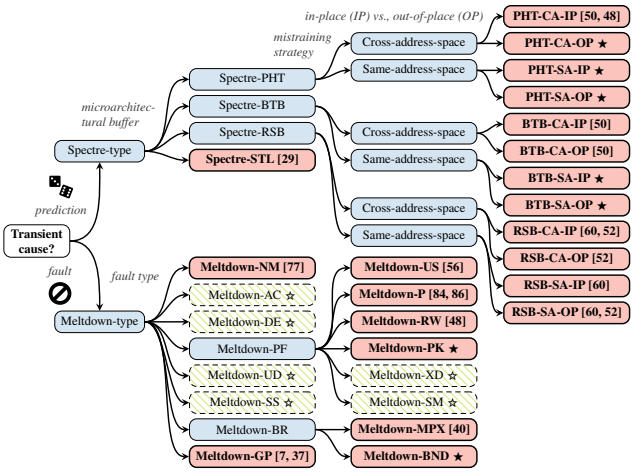

Figure 1: Universal transient execution attack classification tree with demonstrated attacks (red, bold), negative results (green, dashed), some first explored in this work (★ / ☆).

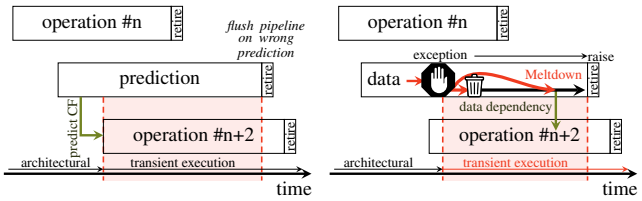

Figure 2: Spectre (left) exploits side effects of instructions predicted to execute next which only operate on architecturally accessible data. Meltdown (right) exploits that exceptions are enforced lazily (*i.e.*, at retirement). In the transient execution window (highlighted area) between exception and retirement, the architecturally inaccessible results of the faulting instruction are not discarded but can be used.

ily. However, transient execution attacks are not limited to the cache: any microarchitectural state that can be changed and observed can be used.

2.1 Cache Covert Channels

Modern CPUs use caches to hide memory latency. However, these latency differences can be exploited in side channels and covert channels [51, 66, 89, 26, 61]. In particular, Flush+Reload allows observations across cores at cache-line granularity, enabling attacks e.g., on cryptographic algorithms [89, 44, 27], user input [26, 55, 71], and kernel addressing information [23]. For Flush+Reload, the attacker continuously flushes data using the `clflush` instruction and reloads the data. If the victim used the cache line, accessing it will be fast; otherwise, it will be slow.

Covert channels are a special use case of side-channel attacks, where the attacker controls both the sender and the receiver. This allows an attacker to bypass all restrictions that exist on the architectural level to leak information. Cache attack senders can be implemented in very little code [57, 61, 24], for instance via "`tmp = array[secret_byte * 4096];`", *i.e.*, a code pattern which also frequently occurs in real-world software. In this paper, we focus on transient execution attacks using covert channels for transmitting data from transient execution to a persistent architectural state.

2.2 Difference between Spectre and Meltdown

Before we discuss the details of Spectre-type and Meltdown-type attacks, we pinpoint the difference between them, as the first level of our classification tree (cf. Figure 1). "Speculative execution" is often falsely used as an umbrella term for attacks based on speculation of the outcome of a particular event (*i.e.*, conditional branches, return addresses, or memory disambiguation), out-of-order execution, and pipelining. The confusion is reinforced as the same method for encoding and transferring secrets to the architectural domain (*i.e.*, a Flush+Reload covert channel) is used in many Meltdown- and Spectre-type attacks. However, Spectre and Meltdown exploit fundamentally different properties of CPUs. A CPU can be vulnerable to Spectre but not Meltdown (e.g., AMD), and vice versa. The only common property of both attacks is that they exploit side effects within the transient execution domain, *i.e.*, within never-committed execution. Hence, the more accurate term is *transient execution* [56, 50], accurately describing the common underlying effect.

For Spectre, transient instructions only work with data which the application can access architecturally. Transiently executed instructions could also be executed in the normal control flow of the application, e.g., if the branch condition is different. Note that although Spectre-BTB can jump to an arbitrary location in the current address space, it still works with data which the application can access architecturally.

For Meltdown, in contrast, the transient instructions work with data which are architecturally inaccessible for the application. In a typical control flow, the data is never accessible for the application. Thus, Meltdown exploits the property that transient instructions following a fault have access to the data of the faulting instruction.

Figure 2 illustrates the difference between Meltdown and Spectre. The mere continuation of the transient execution after a fault itself is required, but not sufficient for Meltdown. Replacing the data of a faulting instruction with a dummy value (*i.e.*, as it is done for Meltdown-US in AMD CPUs) is sufficient to mitigate Meltdown in hardware.

While Meltdown-type attacks so far exploit out-of-order execution, in-order pipelines may allow similar attacks. As such, Meltdown-type attacks are a category separate from Spectre-type attacks. This becomes even more clear when discussing defenses against these attacks in Section 5.

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

Attack \ Element	BTB	BHB	PHT	RSB	STLF
Spectre-PHT (Variant 1) [50]	○	◐	●	○	○
Spectre-PHT (Variant 1.1) [48]	○	◐	●	○	○
Spectre-BTB (Variant 2) [50]	●	◐	○	○	○
Spectre-STL (Variant 4) [29]	○	○	○	○	●
Spectre-RSB (ret2spec) [52, 60]	◐	○	○	●	○

3 Spectre

In this section, we provide an overview of all known variants of Spectre (cf. Figure 1). Spectre-type attacks exploit transient execution after a prediction or data dependency. We propose a categorization based on, first, the prediction mechanism exploited, and second, the mistraining mechanism.

Systematization of Spectre Variants. Table 1 shows all known Spectre-type attacks and the element they exploit. Branch prediction is a fundamental building block of several attacks. Modern processors have many different mechanisms for different types of branches. Intel [33] provides prediction mechanisms for all of them, distributed among different processor components, e.g., the Branch History Buffer (BHB) [11], the Branch Target Buffer (BTB) [54, 18], the Pattern History Table (PHT) [20], and the Return Stack Buffer (RSB) [20]. The corresponding microarchitectural elements are not shared among physical cores [21], but some are shared across logical cores [60]. We propose to combine all attacks that exploit the same microarchitectural element:

- Spectre-PHT: Variant 1 [50] and Variant 1.1 [48] both exploit the *Pattern History Table* (PHT)
- Spectre-BTB: Variant 2 [50] exploits the *Branch Target Buffer* (BTB)
- Spectre-STL: Variant 4 [29] exploits the CPUs memory disambiguation prediction, specifically store-to-load forwarding (STLF)
- Spectre-RSB: ret2spec [60] and Spectre-RSB [52] both primarily exploit the *Return Stack Buffer* (RSB)

NetSpectre [73], SGXSpectre [64], and SGXPectre [13], focus on the exploitation scenario of one or more of these variants, *i.e.*, they apply Spectre variants in specific attacks. Hence, they are not part of the classification of the variants.

Systematization of Mistraining Strategies. In current literature, several strategies for mistraining branch prediction for Spectre-type attacks have been overlooked. While they are not new variants of Spectre, they complete the field of Spectre-type attacks as they allow an attacker to mistrain branch prediction in previously unknown ways. These unknown ways are shown along the known ones in Figure 3.

There are 4 possibilities to mistrain the branch predictor:

1. within the same address space and the same branch location that is later on exploited (same-address-space in-place mistraining)

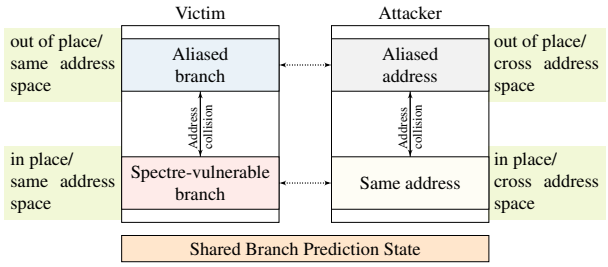

Figure 3: The 4 different ways for prediction mistraining: Either in the same process (same-address-space) or a different attacker-controlled process (cross-address-space). For both variants, mistraining is done with using the vulnerable branch (in-place), or a congruent address (out-of-place).

2. within the same address space with a different branch (same-address-space out-of-place)
3. within an attacker-controlled address space with a branch at the same address as the victim branch (cross-address-space in-place)
4. within an attacker-controlled address space at a congruent address to the victim branch (cross-address-space out-of-place)

3.1 Spectre-PHT (Bounds Check Bypass)

Bypass on Loads. All Spectre-PHT attacks presented so far, are same-address-space in-place attacks. Kocher et al. [50] first introduced Spectre Variant 1, an attack that exploits a conditional branch misprediction to load data transiently. On Intel, the primary microarchitectural element exploited here is the Pattern History Table (PHT). Depending on predictor mode, it is accessed based on combination of some bits from branch instruction virtual address and shift registers [20, 19] which contain the information about the last N branches, with N depending on the architecture [20]. The PHT contains a biasing bit indicating whether a branch is mostly taken or not. A code line "`if (x < len(array1)) { y = array2[array1[x] * 4096]; }`", can be "mistrained" supplying valid values of `x` so that the branch evaluates to true. The attacker can then supply a value for `x` that is out-of-bounds. The PHT then predicts that the branch evaluates to true and execution transiently follows along the mispredicted path. Following the branch, there is a code line acting as a covert channel, transmitting the value read from the out-of-bounds access.

SGXSpectre [64] mounted this attack on SGX enclaves.

Bypass on Stores. Kiriansky and Waldspurger [48] showed that transient writes are also possible by following the same principle. Using this technique, the attacker breaks both type and memory safety during transient execution. In an example code line "`if (x < len(array)) { array[x] = value; }`", we can see that, in the case where the attacker controlled input `x` is valid, some value gets written to the array at offset

x. Following the same approach and by manipulating the element as discussed in Section 3.1, the attacker transiently writes out-of-bounds. This creates a transient buffer overflow, allowing the attacker to execute arbitrary instruction sequences from the code already present within the victim domain, similar as in return-to-libc [76] and return-oriented programming (ROP) [74] attacks.

NetSpectre (Remote Spectre Attack). With NetSpectre, Schwarz et al. [73] proposed a new method that does not rely on an attacker executing its own code on a victim machine. A NetSpectre attack combines two different gadgets, namely a *leak* and *transmit* gadget, into one called a *NetSpectre gadget*. The leak gadget is responsible for accessing a bit stream at an index controlled by the attacker and based on that update the microarchitectural state. The transmit gadget then performs some operation exposing the microarchitectural state through a change in runtime, e.g., lower response time due to a cache hit. The attacker continuously mistrains the processor as in other previous Spectre-PHT attacks. However, as a remote attack, the attacker needs to extract the information using a remote cache covert channel instead of a simple Flush+Reload covert channel.

3.2 Spectre-BTB (Branch Target Injection)

In Spectre Variant 2 [50], the attacker poisons the branch target predictor. This poisoning can occur either in the indirect branch predictor or the generic Branch Target Buffer (BTB) via the fallback mechanism [50]. We refer to both types of poisoning as Spectre-BTB. The indirect branch predictor uses only the lowest 12 bits of the source instruction address in combination with the Branch History Buffer (BHB) to determine the jump target [30]. The BTB contains information about the target location of all recent jumps and uses only the lowest 31 bits [18] which are further folded together [50]. Due to collisions in target predictor structures, the attacker can exploit transient execution of the poisoned prediction to read arbitrary memory from another context, e.g., another process [30]. Contrary to Spectre-PHT, where the attacker bypasses a bounds check by mistraining with valid values, Spectre-BTB mistrains the branch predictor with malicious destinations. By doing so, the attacker can choose an arbitrary code location that gets transiently executed and exposes secret information. This arbitrary code location is called a *Spectre gadget* and is responsible for transferring the secret data into an attacker-controlled covert channel.

To mistrain the target predictor, the attacker must know the virtual address of the gadget in the victim's address space. Knowing this address, the attacker performs indirect branches to this address within its own address space. As the BTB is shared among logical cores, the attacker must co-locate the two processes on the same physical core.

The basic idea of Spectre-BTB is the execution of attacker-chosen gadgets (cf. ROP [74]). The difference to ROP is that the attacker does not rely on a vulnerability in the victim code, but instead exploits mistraining of the BTB and transiently executes the resulting code path.

With SGXPectre, Chen et al. [13] showed that by using branch target injection an attacker can extract critical information from an SGX enclave.

3.3 Spectre-STL (Speculative Store Bypass)

Modern processors improve performance by using a store buffer instead of writing updated values directly back to memory. This allows the processor to continue executing instructions. All data in the store buffer eventually gets written back to the main memory. One complication that arises is that a memory load reads stale memory as the store buffer has not yet written back the updated value to the main memory. To avoid this situation, the processor checks the store buffer for every load that it performs. As processors allow unaligned access, allowing overlaps as addresses might not precisely match, the search of the store buffer is complex and time-consuming. This process is referred to as "Memory Disambiguation". To increase the performance further, processors perform an optimization called "Speculative Store Buffer Bypass". The store buffer is only bypassed in speculative execution if the memory disambiguation predicts that the load does not conflict with any address in the store buffer. Horn [29] exploited the Speculative Store Buffer Bypass optimization combined with Flush+Reload, to leak earlier (e.g., unsanitized) values from memory locations. One restriction of Spectre-STL is that the attacker can only leak information from memory residing in the same privilege level.

3.4 Spectre-RSB (Return Stack Buffer)

Maisuradze and Rossow [60] and Koruyeh et al. [52] propose a new variant that neither exploit the BTB or the PHT, but instead exploit the Return Stack Buffer (RSB)). The RSB is a hardware stack, typically with 16 entries, that keeps track of the return addresses of previous *call* instructions. When a *ret* instruction is encountered, the top of the RSB is used to predict the return address. On hyperthreaded systems, RSBs are dedicated to a logical core. If the RSB is empty, Skylake processors use the BTB as a fallback [20].

Koruyeh et al. [52] show four scenarios where misspeculation occurs. In the first scenario, misspeculation occurs due to an overfill or underfill of the RSB. An overfill occurs when older entries are overwritten due to too many new return addresses being pushed onto the small RSB. After all the addresses that overwrote previous entries have been popped by *ret* instructions and the no longer available addresses are reached, an underfill is caused. An underfill is handled in different ways based on the CPU, e.g., Intel Skylake and newer generations use the BTB as a fallback[87], thus allowing attacks that exploit the BTB.

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

	Method		Spectre-PHT	Spectre-BTB	Spectre-RSB	Spectre-STL
Intel	same-address-space	in-place	● [50] ★		● [60]	● [29]
		out-of-place	★	★	● [60, 52]	○
	cross-address-space	in-place	★	● [50]	● [60, 52]	○
		out-of-place	★	● [50]	● [52]	○
ARM	same-address-space	in-place	●	★	● [6]	●
		out-of-place	★	☆	● [6]	○
	cross-address-space	in-place	★	● [6, 50]	☆	○
		out-of-place	★	☆	☆	○
AMD	same-address-space	in-place	●	★	★	●
		out-of-place	★	☆	★	○
	cross-address-space	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 not possible and previously unknown or not shown (☆). All tests performed with no defenses enabled. Empty fields still require testing.

In scenario two, an attacker can directly pollute the RSB by overwriting the return address on the software stack or deleting it. Then, the value in the RSB does not match the one on the software stack and the processor misspeculates.

Scenario three only speculatively pollutes the RSB. During speculation, calls push values on the stack which do not get removed when the processor discovers the misspeculation. This allows pushing addresses that are outside the accessible address space without dealing with exceptions.

The final scenario causes misspeculation based on the shared RSB, namely that values pushed from one executing thread are used by another thread upon a context switch. Using any method for causing misspeculation, an attacker tries to divert speculative execution to code leaking secret data via an observable covert channel, e.g., cache state.

3.5 Overlooked Mistraining Strategies

Spectre-PHT so far has only been described as a same-address-space in-place attack. We demonstrate that Spectre-PHT works in all four variants, showing that the field of possible attack scenarios is even larger than previously believed. We performed a vulnerability assessment for these new attack vectors on Intel, ARM, and AMD. For Intel, we tested our proofs-of-concept on a Skylake i5-6200U and a Haswell i7-4790. Our AMD test machines were a Ryzen 1950X and a Ryzen Threadripper 1920X. For experiments on ARM, a NVIDIA Jetson TX1 has been used. The results of this vulnerability assessment are shown in Table 2. We can see that, for Spectre-PHT, all vendors have processors that are vulnerable to all four variants of mistraining.

Spectre-PHT is not the only variant where mistraining strategies have been overlooked. Spectre-BTB has only been described in the cross-address-space in-place variant, although Kocher et al. [50] already mention that cross-address-space out-of-place might be possible if only a subset of the virtual address is used in the prediction. Our experiments have shown that we can perform mistraining in the same-address-space in-place variant on all three vendors, although the attack scenario is limited. On AMD and ARM, we were not able to show the out-of-place mistraining strategies. We assume that they are possible, but that they require a different set of bits that we were not able to determine.

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

	Exception Type				Permission Bit					
Attack	#GP	#NM	#BR	#PF	U/S	P	R/W	RSVD	XD	PK
Variant 3a [7]	●	○	○	○						
Lazy FP [77]	○	●	○	○						
Meltdown-BR	○	○	●	○						
Meltdown [56]	○	○	○	●	●	○	○	○	○	○
Foreshadow [84]	○	○	○	●	○	●	○	●	○	○
Foreshadow-NG [86]	○	○	○	●	○	●	○	●	○	○
Meltdown-RW [48]	○	○	○	●	○	○	●	○	○	○
Meltdown-PK	○	○	○	●	○	○	○	○	○	●

Spectre-RSB has been demonstrated in all four variants, but only on Intel [60, 52]. ARM mentions that same-address-space mistraining is possible, but no reference is given for cross-address-space [6]. While we expect them to work, we were not able to observe any leakage with any of our proofs-of-concept. We assume that it is a timing issue. On AMD, we were able to show all types for mistraining.

Spectre-STL exploits store-to-load forwarding, which does not incur any history-based prediction. Hence, mistraining is not possible, and thus, also cross-address-space mistraining strategies are not possible.

4 Meltdown-type Attacks

This section overviews known 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 out-of-order 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 processor's in-order instruction retirement mechanism takes care to discard any architectural effects of such computations, but as with the Spectre-type attacks above, secrets may leak through microarchitectural covert channels.

Systematization of Meltdown Variants. We introduce an extensible classification for Meltdown-type attacks in two dimensions. In the first level, we categorize attacks based on the exception that causes the transient execution. Second, for

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) [84, 86]	◐	●	○	✓
Meltdown-GP (Variant 3a) [7]	○	○	●	✓
Meltdown-NM (Lazy FP) [77]	○	○	●	✓
Meltdown-RW (Variant 1.2) [48]	●	●	○	✗
Meltdown-PK	☆	★	☆	✗
Meltdown-BR	★	★	☆	✗

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

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

4.1 Meltdown-US (Supervisor-only Bypass)

Modern processors commonly feature a "user/supervisor" page-table attribute to denote a virtual memory page as belonging to the operating system kernel. The original Meltdown attack [56] showed how to read kernel memory from user space on pipelined processors that do *not* transiently enforce the user/supervisor flag. The attack consists of 3 steps. In the first step, data inaccessible to the attacker is loaded into a register, which eventually causes a page fault. Before the fault becomes architecturally visible, however, the attacker executes a transient instruction sequence that accesses a cache line based on the privileged data in the register. In the final step, after the exception has been raised, the attacker uses Flush+Reload to determine which cache line was accessed and based on that recover the privileged data.

Meltdown attackers can choose to either handle or suppress page faults resulting from the unauthorized access. Lipp et al. [56] showed how to improve the attack's bandwidth by suppressing exceptions through transaction memory processor features such as Intel TSX [32]. 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. [84] presented Foreshadow, a Meltdown-type attack targeting Intel SGX technology [41]. Unauthorized accesses to enclave memory usually do not raise a `#PF` exception but are instead silently replaced with abort page dummy values (cf. Section 5.2). In the absence of a fault, there is no unauthorized transient computation, and 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 upon subsequent adversarial accesses to the unmapped enclave page. Analogous to the original Meltdown-US attack, the adversary now proceeds with a transient instruction sequence to encode and recover the secret (e.g., via a Flush+Reload covert channel).

Intel [36] 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. However, due to the tight connection of the L1 cache and address translation in modern microarchitectures, the physical address bits (*i.e.*, frame number) of a page-table entry suffering a terminal fault may still be passed on to the L1 cache. Hence, any access to a physical address that hits the L1 cache is passed on to the transient execution, regardless of access permissions. The original Foreshadow [84] attack also shows how to recover uncached enclave secrets by first abusing secure page swapping to prefetch arbitrary enclave pages into the L1 cache.

Foreshadow-NG. Foreshadow-NG [86] generalizes Foreshadow from the attack on SGX enclaves to bypass operating system of 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 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 [36].

4.3 Meltdown-GP (System Register Bypass)

Meltdown-GP (named initially Variant 3a) allows an attacker to read privileged system registers. It was first discovered and published by ARM [7] and subsequently Intel [37] determined that their processors 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, processors 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 [77] propose an attack targeting 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 processor 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 processor 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 [56, 84, 7, 77] 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]. Our systematization revealed, however, that the transient cause exploited above is clearly a `#PF` exception. Hence, this attack must be considered of Meltdown-type, but *not* a variant of Spectre.

4.6 Meltdown-PK (Protection Key Bypass)

Intel Skylake-SP server CPUs support memory-protection keys for user space (PKU) [35]. 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 [83, 28].

We present a novel Meltdown-PK attack to bypass both read and write isolation guarantees enforced through memory-protection keys. Thus, Meltdown-PK shows that PKU isolation can be bypassed if an attacker has code execution in the containing process, even if the attacker cannot execute the `wrpkru` instruction (e.g., due to blacklisting). Moreover, in contrast to cross-privilege level Meltdown attack variants, there is no software workaround. Intel can only fix Meltdown-PK in new hardware or possibly via a microcode update.

Experimental Results. We tested Meltdown-PK on an Amazon EC2 C5 instance running Ubuntu 18.04 with PKU support. For this purpose, 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 processors 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 processors ship with Memory Protection eXtensions (MPX) for efficient array bounds checking.

Our systematic evaluation revealed that Meltdown-type effects of the `#BR` exception have not been thoroughly investigated yet. Specifically, Intel's analysis [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. These observations do not shed light on the `#BR` exception as the root cause for the MPX bounds check bypass, wrongly classifying this attack as a Spectre variant, and do not consider IA32 `bound` protection at all.

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

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

Vendor \ Attack	Meltdown-US [56]	Meltdown-P [84, 86]	Meltdown-GP [7, 37]	Meltdown-NM [77]	Meltdown-RW [48]	Meltdown-PK	Meltdown-BR	Meltdown-DE	Meltdown-AC	Meltdown-UD	Meltdown-SS	Meltdown-XD	Meltdown-SM
Intel	●	●	●	●	●	★	★	☆	☆	☆	☆	☆	☆
ARM	●	○	●	_	●	_	_	☆	☆	☆	_	☆	☆
AMD	○	○	○	○	○	_	★	☆	☆	☆	☆	☆	☆

Symbols indicate whether at least one CPU model is vulnerable (●), no CPU is vulnerable (○), respectively (★ and ☆) if we are the first to show it, or not applicable (_). All tests performed without defenses enabled.

IA32 `bound` (Intel, AMD), or state-of-the-art MPX protection (Intel-only). For Intel, we ran the attacks on a Skylake i5-6200U CPU with MPX support, and for AMD we evaluated both a 2013 E2-2000 and a 2017 Ryzen Threadripper 1920X. In this, we are the first to practically showcase a Meltdown-type transient execution attack exploiting delayed exception handling on AMD processors [1, 56].

4.8 Residual Meltdown (Negative Results)

We systematically studied transient execution leakage for other, not yet tested exceptions. Following Intel's [32] classification of exceptions as *faults*, *traps*, or *aborts*, we observed that all known Meltdown variants so far have exploited faults, but not traps or aborts. We consistently found no traces of transient execution beyond traps or aborts, which leads us to the hypothesis that Meltdown is only possible with faults (as they can occur at any moment during instruction execution). Table 5 and Figure 1 summarize experimental results for fault types tested on Intel, ARM, and AMD.

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

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

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

Segmentation Faults. We consistently found that out-of-limit segment accesses never reach the transient execution.

Table 6: Categorization of Spectre defenses and systematic overview of their microarchitectural target.

Microarchitectural Element \ Defense	InvisiSpec [88]	SafeSpec [45]	DAWG [47]	Taint Tracking [50]	Timer Reduction [50]	RSB Stuffing [39]	Repoline [82]	SLH [12, 17]	YSNB [65]	IBRS [4, 40]	STIPB [4, 40]	IBPB [4, 40]	Serialization [1, 37]	Sloth [48]	SSBD/SSBB [3, 40, 6]	Poison Value [69]	Index Masking [69]	Site Isolation [81]
Cache	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
TLB	◐	●	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
BTB	○	○	○	○	○	○	●	○	○	●	●	●	○	○	○	○	○	○
BHB	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
PHT	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
RSB	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○
AVX	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FPU	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Execution Ports	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Category:			**C1**						**C2**						**C3**			

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

We suspect that, due to the simplistic IA32 segmentation design, segment limits are validated early-on, and immediately raise a `#GP` or `#SS` exception, without sending the offending instruction to the ROB. Thus, Meltdown-SS is not possible.

Instruction Fetch. To yield a complete picture, we finally investigated Meltdown-type effects during the instruction fetch and decode phases. On all of our test systems, we did not succeed in transiently executing instructions residing in non-executable memory (*i.e.*, Meltdown-XD), or following an invalid opcode exception (*i.e.*, Meltdown-UD). We suspect that exceptions during instruction fetch or decode are immediately handled by the processor, without first buffering the offending instruction in the ROB. Moreover, as invalid opcodes have an undefined length, the processor 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 Defenses

In this section, we discuss proposed defenses in software and hardware for the known Spectre and Meltdown variants. Furthermore, we propose a classification scheme for defenses based on their attempt to stop leakage, similar to the classification done by Miller [63].

For Spectre-type attacks, we categorize defenses into the following three categories:

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

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

C3: Ensuring that secret data cannot be reached.

Table 6 lists all proposed defenses against Spectre-type attacks and assigns them to the category they belong.

For Meltdown-type attacks, we categorize defenses into two categories:

D1: Ensuring that architecturally inaccessible data remains inaccessible on the microarchitectural level.
D2: Preventing the occurrence of faults.

5.1 Defenses for Spectre

C1: Mitigating or reducing accuracy of covert channels. Transient execution attacks use a covert channel to transfer the microarchitectural state change induced by the transient instruction sequence such that it can be observed on an architectural level. One approach in mitigating Spectre-type attacks is to reduce the accuracy of said covert channels or prevent them in the first place.

Hardware. One substantial enabler of transient execution attacks is that the transient execution sequence introduces a microarchitectural state change which is observed by the receiving end of the covert channel. In order to secure processors, SafeSpec [45] introduces shadow hardware structures that are used during the execution of transient instructions. Thereby, any microarchitectural state change can be squashed if the prediction of the processor was incorrect. While their prototype implementation protects only caches and, in extension, the TLB, other channels, e.g., using DRAM buffers [68], or execution unit congestion [56, 2], remain open.

Yan et al. [88] proposed InvisiSpec, a method designed 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 actual cache. In order to deal with memory consistency, InvisiSpec needs to compare 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 SafeSpec only protects the caching hierarchy of the CPU, an attacker can still exploit other covert channels.

Kiriansky et al. [47] propose a method to securely partition the cache across its ways. By introducing 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 processor tracks data loaded during transient execution and prevents it from being used in subsequent operations which might leak it.

Software. Many covert channels require an accurate timer in order to distinguish microarchitectural states, e.g., measuring the memory access latency to distinguish between a cache hit and cache miss. If the accuracy of timers is reduced such that an attacker can not differentiate between the microarchitectural stages any longer, the receiver of the covert channel cannot deduce the sent information. In order to mitigate browser-based attacks, many web browsers reduced the accuracy of timers in JavaScript by adding jitter [62, 69, 80, 85]. However, Schwarz et al. [72] demonstrated that timers can be constructed in many different ways and, thus, further mitigations are required [70]. 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 [75].

NetSpectre requires different strategies due to its remote nature. To mitigate the attack, Schwarz et al. [73] propose using DDoS detection mechanisms as the attacker has to send multiple thousands of identical packets to the victim. The second method they propose is to add artificial noise to the network latency. This increases the number of measurements an attacker has to perform to extract a single bit. At some point, the attack becomes infeasible in practice.

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

Since all Spectre-type attacks exploit different prediction mechanisms used for speculative execution, an effective approach would be to disable speculative execution entirely [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.

Software. Intel and AMD proposed to use serializing instructions like `lfence` on both outcomes of a branch [1, 37]. 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 [37]. Hence, Intel further proposed to use static analysis [37] to minimize the number of serializing instructions introduced. Microsoft uses the static analyzer of their C Compiler MSVC [67] to detect known-bad code patterns and insert `lfence` instructions automatically. Open Source Security Inc. [31] 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 [1]. This, in turn, reduces the size of the speculation window in case that misspeculation occurs.

While `lfence` instructions stop the speculative execution, Schwarz et al. [73] showed they do not stop speculative code fetches and other microarchitectural behaviors happening before execution. This includes powering up the AVX functional units, instruction cache fills, and iTLB fills which might still leak data.

Evtyushkin et al. [19] propose a similar method to serializing instructions. They propose that a software developer

can indicate branches capable of leaking sensitive information. When indicated, the processor should not predict the outcome of these branches. This also stops 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 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] called You Shall Not Bypass. They exploit that processors 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.

Google proposes a method called *retpoline* [82], a code sequence that replaces indirect branches with return instructions, to prevent branch poisoning. This method ensures that the return instruction predicts to a benign endless loop through the RSB to catch speculation. The actual target destination is pushed on the stack and returned to using the `ret` instruction. For retpoline, Intel [39] notes that in future processors that have Control-flow Enforcement Technology [34] (CET) capabilities to defend against ROP attacks, retpoline might trigger false positives in the CET defenses. To mitigate this possibility, future processors also implement hardware defenses for Spectre-BTB called *enhanced IBRS* [39]. This eliminates the need for retpoline.

To prevent the processor from speculating on the store buffer check, Intel provides a microcode update for disabling the mechanism called Speculative Store Bypass Disable (SSBD). AMD also supports SSBD [3]. 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].

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.

Hardware. One of the building blocks for some variants of Spectre is branch poisoning where an attacker mistrains a prediction mechanism (see Section 3). In order 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 processor 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 processors implement specific controls that allow to invalidate 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 [10], 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`) [10].

To prevent Spectre-PHT attacks, Kiriansky and Waldspurger [48] propose SLoth, a group of three microarchitectural defenses, to constrain store-to-load forwarding. The first, SLoth Bear, prevents store-to-load forwarding from either transient stores or to transient loads through a microcode update. The second, SLoth, relies on the compiler marking instructions as candidates for forwarding. The third, Arctic SLoth, uses dynamic detection of load and store pairs to determine candidates for forwarding.

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 [69]. 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 for Google Chrome called *site isolation* [81]. 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. With Chrome 67, Google has enabled site isolation by default [81].

Kiriansky and Waldspurger [48] propose to restrict access to sensitive data by using protection keys like Intel Memory Protection Key (MPK) technology [32]. 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 a `lfence` instruction in `wrpkru`, an instruction used to modify protection keys.

5.2 Defenses for Meltdown

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

In the case of Meltdown-type attacks, the fundamental problem is that the processor allows the transient instruction stream to compute on architecturally inaccessible values, and hence, leak them. By assuring that on the occurrence of a fault, the execution does not continue or respectively does not continue with the otherwise inaccessible value, such attacks can be mitigated in future silicon hardware designs. However, mitigations for existing microarchitectures are necessary, either through microcode updates, or operating system level software workarounds. All of 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 can also defend against Meltdown-US attacks by preventing kernel secrets from being mapped in user space. Besides its performance impact, KAISER has one practical limitation [56, 22]. Due to the design of x86, some privileged memory locations must always remain mapped in user space. KAISER has been implemented for the Linux kernel under the name kernel page-table isolation (KPTI) [59] and has also been backported to older versions. Microsoft provides a similar patch as of Windows 10 Build 17035 [43] and Mac OS X and iOS also have similar features available [42].

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 [38]. 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].

Directly addressing the access control race condition exploited by Foreshadow and Meltdown may not be feasible with microcode updates [84]. Intel therefore proposes a multi-stage approach to mitigate Foreshadow (L1TF) attacks on current processors [36, 86]. First, to maintain conventional process isolation, the operating system kernel should take care to sanitize the physical address field of unmapped page-table entries. The kernel could either clear the physical address field, or let it point to non-existent physical memory. In case of the former, Intel clarifies that 4 KB of dummy data should be placed at physical address `0x0`, and additionaly the PS bit should be cleared at the page directory and page directory pointer levels to prevent attackers from exploiting huge 2 MB or 1 GB pages.

For SGX enclaves or hypervisors, that 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. 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 [84] 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 [77] that is deployed since Linux 4.6 [58]. By replacing lazy switching with eager switching, the FPU is always available, and access to the FPU can never fault. Here, the countermeasure if effective, as there is no other way to provoke a fault when accessing the FPU.

5.3 Evaluation of Defenses

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

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

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

Site isolation still allows data leakage within the same process and is therefore not a full solution. With SLH, we were not able to observe any leakage, indicating that it successfully prevents Spectre-PHT-based attacks. This does not hold for YSNB as we were still able to observe leakage after introducing a data dependency.

IBRS, STIBP, and IBPB are heavily dependant on the underlying architecture and the used Linux version. As of Linux 4.19, systems supporting enhanced IBRS use this method instead of retpoline. If it is not available, the kernel is protected by retpoline if compiled correspondingly. IBRS is only activated for firmware calls as retpoline has a lower performance impact and the kernel does not contain any indirect branches. The IBPB support on Linux seems to be incomplete as the BTB is not flushed if a process is set to be dumpable. As the default behavior on Linux is to mark a process as dumpable, all processes that do not explicitly change that are still vulnerable to a Spectre-BTB attack. We were not able to verify IBPB, IBRS, and STIBP on AMD as our test machine does not support them.

Also, on current systems, STIBP is not enabled. There is a patch enabling it if three conditions are met [53]: The CPU has to be vulnerable to Spectre-BTB; hyperthreading must be supported and a sibling be online; and auto-selection of Spectre-BTB defenses must be enabled, *i.e.*, the default case. We verified whether a cross-address-space Spectre-BTB attack still works on a patched Linux system and did not observe any leakage, indicating that STIBP seems to work on Intel. In our tests, RSB stuffing only proved to be a reasonable approach against Spectre-RSB from different processes. Otherwise, we are able to circumvent it.

To use SSBD in user space, the process to be protected must issue a `prctl` system call to enable it. If the kernel has been compiled with `CONFIG_SECCOMP=y`, then SSBD is enabled for all processes using seccomp. Our tests showed that SSBD is a functional defense for Spectre-STL. We searched projects on GitHub to verify if any use SSBD via prctl. We found no project using this method except forks of Linux kernels. As the number of projects supporting seccomp is small, we conclude that SSBD is not commonly used. On ARM, we verfied that SSBB works but it has to be explicitly added by the developer.

In our experiments, we were not able to observe any leakage after a bounds check in the presence of a serializing instruction on Intel or AMD. For ARM, we were also not able to observe any leakage following a barrier instruction (`CSDB`) in combination with conditional selects or moves, but on some ARM implementations, we were able to leak data from a single memory access through the TLB after the `DSB SY+ISH` instructions. As a result, the static analysis approach of Microsoft and others is only a valid defense technique on ARM if a `CSDB` in combination with conditional selects or moves is emitted. As the observed leakage is only caused by one access and the common Spectre-PHT sequence consists of two loads, `DSB SY+ISH` still works in most cases. On AMD, `lfence` is not serializing by default. Instead, an MSR has to be set for the instruction to serialize [4].

Taint tracking [50] theoretically mitigates all forms of Spectre-type attacks as data that has been tainted cannot be used in a transient execution. Therefore, the data does not enter a covert channel and can subsequently not be leaked.

Reducing the accuracy of timers [50] is only a partial solution as Schwarz et al. [72] have shown that different methods can be used to generate a new, accurate timer. Additionally, it only makes it harder for an attacker to get the information, but that can be circumvented by taking more measurements.

While the Sloth [48] family of defenses was initially proposed to mitigate Spectre-PHT attacks, we argue that they should also be able to theoretically mitigate Spectre-STL.

Meltdown Defenses. We verified whether we can still execute our newly discovered Meltdown-type attacks on a fully-patched system. Even with all mitigations enabled, we were still able to execute Meltdown-BR, Meltdown-PK, and Meltdown-RW. These results indicate that current mitigations only prevent Meltdown-type attacks that do not cross the current privilege level.

6 Conclusion

Transient execution attacks leak otherwise inaccessible information via the CPU's microarchitectural state from instructions which are never committed. We presented a sound and extensible systematization of transient execution attacks. Our systematization uncovered 7 (new) transient execution attacks (Spectre and Meltdown variants) which have been overlooked and have not been investigated so far. We demonstrated *all* these variants in practical proof-of-concept attacks and evaluated their applicability to Intel, AMD, and ARM processors. We also systematically evaluated all defenses,

Table 7: Spectre-type attacks and whether a defense mitigates it.

Attack		InvisiSpec [88]	SafeSpec [45]	DAWG [47]	RSB Stuffing [39]	Retpoline [82]	Poison Value [69]	Index Masking [69]	Site Isolation [81]	SLH [12, 17]	YSNB [65]	IBRS [4, 40]	STIPB [4, 40]	IBPB [4, 40]	Serialization [1, 37]	Taint Tracking [50]	Timer Reduction [50]	Sloth [48]	SSBD/SSBB [3, 40, 6]
Intel	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	●	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	●	◐	◐	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
ARM	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●
AMD	Spectre-PHT	□	□	□	◇	◇	●	◐	◐	●	○	◇	◇	◇	◐	■	◐	◧	◇
	Spectre-BTB	□	□	□	◇	●	◇	◇	◐	◇	◇	■	◧	◧	◇	■	◐	◇	◇
	Spectre-RSB	□	□	□	◐	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	◇	◇
	Spectre-STL	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

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

discovering that some transient execution attacks are not successfully mitigated by the rolled out patches and others are not mitigated because they have been overlooked. Hence, we need to think about future defenses carefully and plan to mitigate attacks and variants that are yet unknown.

Acknowledgments

This work has been supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This research received funding from the Research Fund KU Leuven, and Jo Van Bulck is supported by the Research Foundation – Flanders (FWO). 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] ADVANCED MICRO DEVICES, INC. Software techniques for managing speculation on AMD processors, `https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf` 2018.

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

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

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

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

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

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

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

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

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

[11] BHATTACHARYA, S., MAURICE, C., BHASIN, S., AND MUKHOPADHYAY, D. Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls. Cryptology ePrint Archive, Report 2017/968, 2017.

[12] CARRUTH, C. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation, `https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html` 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] CHEN, G., WANG, W., CHEN, T., CHEN, S., ZHANG, Y., WANG, X., LAI, T.-H., AND LIN, D. Racing in hyperspace: closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)* (May 2018), IEEE.

[15] COSTAN, V., LEBEDEV, I. A., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium* (2016).

[16] DONG, X., SHEN, Z., CRISWELL, J., COX, A., AND DWARKADAS, S. Spectres, virtual ghosts, and hardware support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy* (2018), ACM, p. 5.

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

[18] EVTYUSHKIN, D., PONOMAREV, D., AND ABU-GHAZALEH, N. Jump over aslr: Attacking branch predictors to bypass aslr. In *International Symposium on Microarchitecture (MICRO)* (2016).

[19] 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).

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

[21] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* (2016).

[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., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[25] GRUSS, D., SCHUSTER, F., OHRIMENKO, O., HALLER, I., LETTNER, J., AND COSTA, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium* (2017).

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

[27] 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 (COSADE)* (2015).

[28] 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. *Technical report* (2018).

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

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

[31] INC, O. S. S. Respectre: The state of the art in spectre defenses, `https://www.grsecurity.net/respectre_announce.php` Oct. 2018.

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

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

[34] INTEL CORP. Control-flow Enforcement Technology Preview, `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf` June 2017.

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

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

[37] INTEL CORP. Intel analysis of speculative execution side channels, July 2018. `https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf` Rev 4.0.

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

[39] INTEL CORP. Retpoline: A Branch Target Injection Mitigation, `https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf` June 2018.

[40] INTEL CORP. Speculative Execution Side Channel Mitigations, `https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf` May 2018.

[41] INTEL CORPORATION. Intel Software Guard Extensions (Intel SGX), 2016. `https://software.intel.com/en-us/sgx` Retrieved on November 7, 2016.

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

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

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

[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 , `https://patchwork.kernel.org/patch/10427513/` 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. *Cryptology ePrint Archive: Report 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, `https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html` 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] KOSINA, JIRI. x86/speculation: Enable cross-hyperthread spectre v2 STIBP mitigation, `https://lore.kernel.org/patchwork/patch/983954/` Sept. 2018.

[54] LEE, S., SHIH, M., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium* (2017).

[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] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *S&P* (2015).

[58] LUTOMIRSKI, ANDY. x86/fpu: Hard-disable lazy FPU mode, `https://lkml.org/lkml/2018/6/14/509` June 2018.

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

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

[61] 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).

[62] MICROSOFT EDGE TEAM. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer, https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/ Jan. 2018.

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

[64] O'KEEFFE, DAN AND MUTHUKUMARAN, DIVYA AND AUBLIN, PIERRE-LOUIS AND KELBERT, FLORIAN AND PRIEBE, CHRISTIAN AND LIND, JOSH AND ZHU, HUANZHOU AND PIETZUCH, PETER. Enable SharedArrayBuffer by default on non-android, https://github.com/lsds/spectre-attack-sgx Jan. 2018.

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

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

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

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

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

[71] 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).

[72] 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).

[73] SCHWARZ, M., SCHWARZL, M., LIPP, M., AND GRUSS, D. Netspectre: Read arbitrary memory over network. *arXiv:1807.10535* (2018).

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

[75] SMITH, BEN. Enable SharedArrayBuffer by default on non-android, https://chromium.googlesource.com/chromium/src/+/4dbb4407b8a64dd9463ae34b1e9c19475acc1128 Aug. 2018.

[76] SOLAR DESIGNER. Getting around non-executable stack (and fix), http://seclists.org/bugtraq/1997/Aug/63 Aug. 1997.

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

[78] STRACKX, R., AND PIESSENS, F. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv:1712.08519* (Dec. 2017).

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

[80] THE CHROMIUM PROJECT. https://www.chromium.org/Home/chromium-security/ssca Actions required to mitigate Speculative Side-Channel Attack techniques.

[81] THE CHROMIUM PROJECTS. http://www.chromium.org/Home/chromium-security/site-isolation Site Isolation.

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

[83] 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).

[84] 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).

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

[86] 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. *Technical report* (2018).

[87] WOODHOUSE, DAVID. x86/retpoline: Avoid return buffer underflows on context switch, https://lore.kernel.org/patchwork/patch/871060/ Jan. 2018.

[88] YAN, M., CHOI, J., SKARLATOS, D., MORRISON, A., FLETCHER, C. W., AND TORRELLAS, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51th International Symposium on Microarchitecture (MICRO'18)* (2018).

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

A Consistency of the Universal and Unambiguous Naming Scheme

While our naming and classification scheme (cf. Figure 1) is based on the names of the microarchitectural elements and the exceptions found on modern x86 processors, this does not limit the generality or consistency of our systematization. Generally, microarchitectural elements which have equivalent functionality are equivalent in our classification scheme. Other microarchitectural elements with different functionality, e.g., other prediction mechanisms, can extend the given classification scheme. Exception names are typically specific to one architecture. However, ARM also has equivalent exceptions types, such as instruction aborts (formerly prefetch aborts) and data aborts which correspond to the class of page faults [8, 9]. Still, any exception which does not have a corresponding one in our classification scheme can be added in a consistent way by following the existing classification scheme up to the point where no alternative fits.