Phantom: Exploiting Decoder-detectable Mispredictions

Johannes Wikner[†], Daniël Trujillo[†], and Kaveh Razavi
ETH Zürich
[†] Equal contribution joint first authors

ABSTRACT

Violating the Von Neumann sequential processing principle at the microarchitectural level is commonplace to reach high performing CPU hardware — violations are safe as long as software executes correctly at the architectural interface. Speculative execution attacks exploit these violations and queue up secret-dependent memory accesses allowed by long speculation windows due to the late detection of these violations in the pipeline. In this paper, we show that recent AMD and Intel CPUs speculate very early in their pipeline, even before they decode the current instruction. This mechanism enables new sources of speculation to be triggered from almost any instruction, enabling a new class of attacks that we refer to as PHANTOM. Unlike Spectre, PHANTOM speculation windows are short since the violations are detected early. Nonetheless, PHANTOM allows for *transient fetch* and *transient decode* on all recent x86-based microarchitectures, and *transient execution* on AMD Zen 1 and 2. We build a number of exploits using these new PHANTOM primitives and discuss why mitigating them is difficult in practice.

1. INTRODUCTION

Security research at the intersection of software and hardware has surfaced a concerning amount of information leaks [9,11,12,13,21,36,39,45,46,58,59,61,62,66,67,73,74]. Spectre in particular forces the misprediction of branching instructions, leading to arbitrary information disclosure in many scenarios of interest [9, 11, 21, 36, 39, 46, 62, 73, 74]. While it is commonly assumed that the CPU speculates only after it decodes the instruction to be a branch, we show in this paper that all recent AMD and Intel CPUs speculate at much earlier stages of their pipeline. Our investigation into this speculation *before instruction decode* uncovers a new class of attacks that we refer to as PHANTOM speculation. We show the practical importance of PHANTOM speculation by building a number of exploits for the AMD Zen microarchitectures.

Speculation before instruction decode. In the first stage of a pipelined CPU architecture, the Instruction Fetch unit fetches blocks of instructions from the instruction cache. Instruction prefetchers try to predict future instruction cache lines and bring them into the cache before execution reaches those cache lines [78]. These predictions are made by learning the control flow of the instructions over time, and are not based on the instructions themselves. Depending on the control flow of the program, instructions that are prefetched may never enter the pipeline. We show in this paper that modern AMD and Intel CPUs do much more to improve performance: they predict and fetch the next block of instructions from the instruction cache into the pipeline immediately after the current fetch, *before* branch sources are decoded, in line with designs previously discussed in the microarchitecture community [8, 15].

The decision of whether the current instruction is a branch is made by the CPU's frontend before decoding the instruction by consulting the Branch Target Buffer (BTB). On top of predicting whether the current instruction is a branch, the BTB further provides the predicted branch target to the frontend. The branch target effectively becomes the frontend's next instruction fetch location.

Backend-issued resteers and Spectre. While speculation before decode improves performance, it may also result in bad speculation. This bad speculation can lead to information disclosure as shown by many variants of Spectre. The bad speculation that are caused by these Spectre variants is resolved in the CPU's backend where µops are executed. This is due to the fact that the instructions that cause speculation, such as indirect branches [9, 11, 21, 36, 48, 62] or returns [39, 46, 73, 74], have dependencies that can only be resolved at the execute stage. Upon detection of bad speculation, the backend issues a resteer to the frontend, so that it can restart instruction fetch from a corrected program counter. The time between misprediction and backend-issued resteer allows for speculative execution of several memory loads, which Spectre exploits.

Frontend-issued resteers and PHANTOM. The backend is not the only source of resteers. We find in this paper that in many cases of interest, the CPU frontend also issues resteers when the misprediction is detected by the decoder. We systematically explore the cases under which an attacker can force mispredictions that are resolved by the frontend. Frontend-issued resteers caused by bad speculation is the source of a new class of attacks that we call PHANTOM. Unlike Spectre, PHANTOM speculation windows are short, and it is unclear how far in the pipeline such speculations proceed. We build observation channels that can identify pipeline stages for various PHANTOM speculations on AMD and Intel CPUs. Insights from these observation channels provide attackers with exploitation primitives for *transient fetch* and *transient decode* on all recent AMD CPUs, and short *transient execution* on AMD Zen 1 and 2.

Exploitation with PHANTOM. Our analysis shows that PHANTOM speculation enables fetch, decode and a short execution window of the mispredicted target on AMD Zen 1 and 2, which is enough to trigger memory operations. On AMD Zen 3 and 4, PHANTOM speculation enables fetch and decode of the mispredicted target. We show the conditions for these PHANTOM speculations happen in real-world scenarios by building three exploits. First, we show PHANTOM speculation breaks KASLR of the kernel image on all AMD microarchitectures. Building this exploit required us to reverse engineer cross-privilege BTB functions on AMD Zen 3 and 4 for the first time. Second, on AMD Zen 1 and 2, we show that PHANTOM speculation can further break physmap KASLR, signifying full KASLR derandomization. Lastly, we show that PHANTOM speculation can leak arbitrary kernel memory with MDS gadgets on AMD Zen 1 and 2, which are not affected by MDS [13, 61, 67]. Our analysis shows that existing mitigations cannot stop PHANTOM attacks.

Contributions. In summary our contributions are as follows.

- We systematically analyze sources of misprediction before instruction decode on recent AMD and Intel microarchitectures. We further analyze how far in the pipeline target instructions proceed.

- We present PHANTOM, a new class of attacks based on decoder-detectable misprediction. PHANTOM speculation enables new primitives such *transient fetch* and *transient decode* on all recent AMD microarchitectures, and short *transient execution* on AMD Zen 1 and 2.

- We build three attacks based on PHANTOM speculation, derandomizing KASLR of the kernel image and physmap, as well as arbitrary kernel memory leak with an MDS gadget. The attacks on AMD Zen 3 and 4 required us to reverse engineer the cross-privilege BTB functions for the first time.

Disclosure efforts. We initially disclosed our findings to AMD in June 2022. AMD issued CVE-2022-23825 and released an advisory [5] in July 2022 that addresses both our findings and another issue [74]. However, our analysis shows that the published advisory only considers transient *execution*, but not transient fetch and decode. This means that existing mitigations are ineffective against our attacks, which we subsequently reported to AMD in May 2023. In response, AMD informed us that that they are not planning on releasing a new CVE. We have reported the existence of PHANTOM speculation to Intel in July 2023.

2. BACKGROUND

To understand the source of PHANTOM speculation and how to potentially defend against it, in this section we give an introduction to branch target prediction and how it interacts with the CPU pipeline. We then discuss how Spectre attacks use bad speculation to infer secrets, and the current approaches for mitigating them.

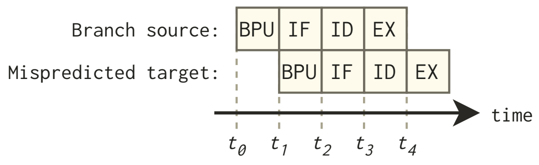

Fig. 1: BPU mispredicts the next branch target already while the branch source is in IF. How far the mispredicted target advances in the pipeline depends on the branch source dependencies, including its decoding stage. For example, at t_2 the decoder may discover that the branch source conflicts with the prediction at t_1.

2.1 Branch Target Prediction

The Branch Prediction Unit (BPU) provides the CPU pipeline with predictions of the upcoming control flow. To provide accurate predictions, branch predictors record control-flow history in registers and table-like buffers. These data structures are read from and written back to as instructions advance through the pipeline. We discuss some of these data structures next.

BHB and BTB. Branch History Buffers (BHBs) contain a footprint of recently encountered control-flow edges, and are used to index Branch Target Buffers (BTBs) [9, 36]. BTB entries contain branch targets predictions, which may serve indirect [36] or direct branches [71], as well as returns [74]. BTB entries can serve multiple targets depending on the size of each entry, which may depend on the distance between the branch source and target [78]. The BPU selects the target by matching a tag of the current BHB with the tag from one of the targets [9].

RSB. Return Stack Buffer (RSB), sometimes referred to as Return Address Stack (RAS), is another data structure that the CPU uses for predicting the target of return instructions [39, 46]. The RSB contains the *N* most recently encountered call sites so that return speculation can proceed without memory look-ups from the volatile stack pointer (*N* is usually 16 or 32).

2.2 Pipelining

Without pipelining, the slowest instruction determines the length of a CPU's clock cycle. With a long clock cycle, most components of the CPU remain underutilized. To improve utilization and increase instruction throughput, pipelining enables simultaneous processing of instructions by splitting their execution into stages. Pipeline stages include Instruction Fetch (IF), Instruction Decode (ID), and Out-of-Order Execute (EX), after which instructions *retire* and effects are committed to the architectural, visible state. While improving utilization, pipelining introduces concurrency issues. For example, if the location of the next IF depends on result of the currently executing instructions, awaiting their retirement would stall the pipeline. Therefore, the BPU provides IF a prediction of the next location. In case of a misprediction, an incorrect control flow will consequently be processed until a corrected fetch location is provided through a *resteer* signal. The time range between misprediction and resteer is known as the *speculation window* and can be several hundred clock

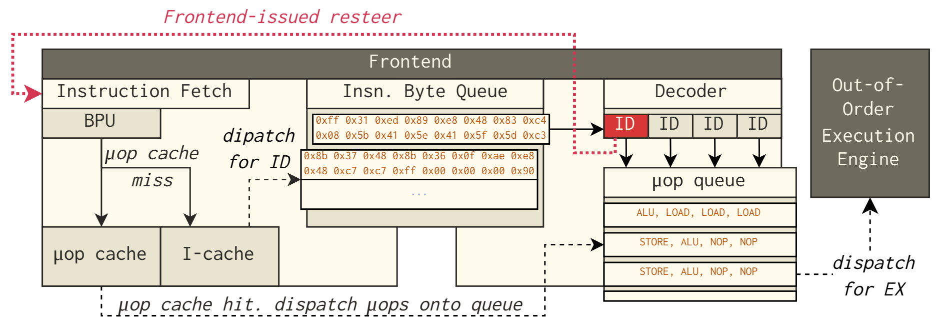
Frontend-issued resteer
Frontend
Instruction Fetch
BPU
µop cache miss
µop cache
I-cache
dipatch for ID
Insn. Byte Queue
0xff 0x31 0xed 0x89 0xe8 0x48 0x83 0xc4
0x08 0x5b 0x41 0x5e 0x41 0x5f 0x5d 0xc3
0x8b 0x37 0x48 0x8b 0x36 0x0f 0xae 0xe8
0x48 0xc7 0xc7 0xff 0x00 0x00 0x00 0x90
...
Decoder
ID
ID
ID
ID
µop queue
ALU, LOAD, LOAD, LOAD
STORE, ALU, NOP, NOP
STORE, ALU, NOP, NOP
µop cache hit. dispatch µops onto queue
Out-of-Order Execution Engine
dispatch for EX

Fig. 2: The different pipeline stages are decoupled, asynchronous modules that consume input queues and dispatch onto output queues. This example shows a decoupled *Instruction Fetch* unit that dispatches raw instruction bytes onto an *Instruction Byte Queue* (IBQ), which are decoded and dispatched onto a *µop-queue*. The *Instruction Fetch* unit continues to fetch instruction independently and push onto the IBQ.

cycles wide.

Figure 1 shows an example of a mispredicted control flow. It is important to consider that the control flow of the mispredicted branch target may start advancing through the pipeline while its branch source is still at an early stage — even before the branch source has reached the ID stage. Figure 2 illustrates how this can happen in finer detail. The IF and ID units are implemented as decoupled modules, communicating via the Instruction Byte Queue (IBQ). Blocks of raw instruction bytes are dispatched by IF onto the IBQ to be decoded by ID. These instructions bytes are then decoded into µops and dispatched onto a µop-queue. Another pipeline unit of the CPU backend can then process these µops for resource allocation and scheduling in the Out-of-Order execution engine. We imagine that the majority of the CPU pipeline components follow this asynchronous, event-driven behavior.

A problem with this architecture is the limited transactional support. Once work has been dispatched, it is difficult to keep track of the stages it reaches and what state needs to be rolled back on a resteer. In Figure 2, the BPU provides a mispredicted Program Counter (PC), from which instruction bytes will be fetched. At decode, the next PC can already be finalized, unless a branch source that is execute-dependent was decoded (e.g., conditional, indirect, or return branch). The decoder can therefore provide IF with this feedback, which could disagree with BPU-provided prediction. However, for the execute-dependent branch sources, the next PC can only be finalized during the execute stage. These execute-dependent cases are exploited in Spectre attacks.

2.3 Spectre

Spectre [36] is a subclass of transient execution attacks that abuses branch mispredictions. While the multitude of Spectre variants is ever-increasing, their common goal is to make the mispredicted control flow operate on architecturally inaccessible information, such that it can be transmitted (i.e., *leaked*) via a covert channel, commonly the CPU caches. We refer to the code snippets responsible for the transmission of information as the *disclosure gadgets*. By forcing a misprediction of an execute-dependent branch source, the attacker can hijack the speculative control flow to execute a disclosure gadget that loads secrets from memory and exfiltrates them using a covert channel.

Conventional Spectre attacks rely on hijacking execute-dependent branch sources that have speculation windows that are wide enough to queue up several secret-dependent memory loads. Our work explores a new Spectre class that considers speculation windows that can be resteered before the mispredicted instruction reaches the execute stage. The exploits we present work thanks to *speculation before instruction decode*, where not only the branch target but also the branch source can be mispredicted.

2.4 Mitigating Spectre

Modern systems use software and hardware-based defenses to block Spectre leaks, preventing transient access to secrets [1, 63, 75, 76], patching the branch sources susceptible to misprediction [4, 5, 31, 32, 65], and restricting the use of predictions [29, 30]. Early branch target prediction can impact the latter two, which we will briefly discuss.

Patching the source of mispredicted control flow. The *lfence* x86 instruction limits the mispredicted control flow by stalling execution until pending loads retire. Placing *lfence* where bad speculation may occur is often a recommended mitigation as it minimizes the speculation window [32]. *Retpolines* [4, 65] and *jmp2ret* [5] rewrite potential sources of attacker-controllable misprediction to prevent speculation altogether.

Restricted use of predictions. *RSB stuffing* [46], *post-barrier RSB stuffing* [64], *call-depth tracking* [20], and *untrain ret* [5] are software defenses that overwrite bad predictions with dummy targets. Hardware solutions strive to restrict branch targets instead of removing them. Indirect Branch Restricted Speculation (IBRS) [29] variants (AutoIBRS, eIBRS, Legacy IBRS) restrict branch predictions based on the privilege mode. Moreover, Single Thread Indirect Branch Predictors (STIBP) [30] restrict sibling threads' branch predictors from influencing each other.

The two types defenses have practical challenges. When patching the branch source, knowing which branch sources are vulnerable is not trivial in face of branch target prediction before instruction decode, as we show in this paper. Furthermore, for defenses that advertise restricted speculation, it is unclear whether they consider all pipeline stages to which bad speculation can advance.

3. THREAT MODEL

We consider a realistic threat model with an attacker that can execute unprivileged code on top of a recent Linux kernel. We assume a modern CPU that supports speculative and out-of-order execution. As we show in this paper, such CPUs may employ branch prediction before instruction decode. We further assume a default Ubuntu configuration including all state-of-the-art Spectre defenses, both in software and hardware. These include AutoIBRS [55], retpolines [65] and untrain ret [5]. The goal of the attacker is to infer secrets that are otherwise only available to privileged software by exploiting decoder-detectable mispredictions.

4. OVERVIEW

Previous research on the topic of branch target prediction exploitation almost exclusively focuses on cases where the training and victim branch sources have the same type of instruction, specifically indirect branches [9, 18, 36, 48]. Asymmetric combinations, where the training and victim branch sources are of different instruction types, are commonly not considered exploitable: instruction type mismatches can be discovered already at decode. Consequentially, mispredictions are detectable and can be resteered by the decoder before reaching execute. However, recent work has shown that the asymmetric case can also lead to long exploitable speculation windows [74]. As discussed in § 2.4, these cases can potentially compromise existing mitigations. Hence, the first question we ask in this paper is which asymmetric cases in modern CPUs can potentially lead to exploitable scenarios:

Research Question RQ1.

Which asymmetric combinations of branch types can trigger misprediction?

We hypothesize that the asymmetric combinations of branch types will likely lead to short mispredictions that the CPU can detect during decode due to mismatching instruction types. Consequently, our analysis could benefit from observation channels that allow us to infer how far in the pipeline a mispredicted control flow advances. For example, if we observe transient memory operations from the mispredicted target, we can infer that the mispredicted control flow reached Execute (EX) and advanced through the preceding stages, namely IF and ID. For when we cannot observe EX, we build the tools to observe *transient fetch* and *transient decode* of the (mispredicted) branch target. We then use these observation channels to detect branch target speculation for asymmetric combinations of training and victim branch sources in Section 5. Because the CPU frontend predicts a branch source instruction that may not match reality, or not even exist, we refer to these cases as PHANTOM speculation.

The established methods to observe branch misprediction rely on transient execution effects (e.g., data cache and port contention). However, because certain types of mispredictions may be invisible on certain systems, it is difficult to tell whether it is because the microarchitecture exhibits a different prediction scheme or because the resteer was issued before the mispredicted branch target could reach the EX stage and emit a signal. Being able to observe speculation windows without relying on EX, our second question is:

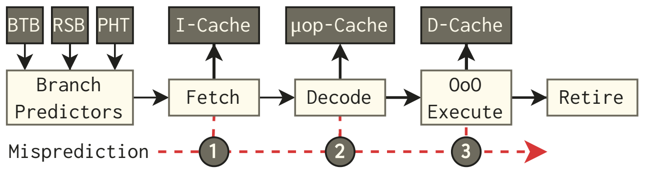

Fig. 3: We can discover a misprediction at any of the stages preceding retire. ❶ We can observe fetch by querying the I-Cache, ❷ decode by querying the µop-Cache, and ❸ using a load instruction in the mispredicted path, we can observe execution by querying the D-Cache.

Research Question RQ2.

What new exploitation primitives can we build using PHANTOM speculation?

In Section 6, we investigate what exploitation primitives we can build using PHANTOM speculation, even when predictions do not advance to the execute stage, that is *execution-free* speculation. Having built practical exploitation primitives, we investigate the following question:

Research Question RQ3.

What information can we leak with these exploitation primitives?

Short speculation windows are often considered harmless [4, 5], but this is unfortunately not always true [48]. In the same spirit, in Section 7 we build efficient end-to-end KASLR derandomization attacks on the latest generations of AMD CPUs using our exploitation primitives, despite mitigations that should block speculation. To do this, we reverse engineer BTB indexing schemes of the Zen 3 and Zen 4 microarchitectures. Finally, we demonstrate that PHANTOM speculation can be nested inside a conventional Spectre attack to increase its known attack surface. This nesting allows us to repurpose MDS-gadgets [34], known to be exploitable only on Intel CPUs, to also be exploitable on AMD Zen 1 and Zen 2 CPUs.

5. PHANTOM SPECULATION

In this section, we study methods for observing branch mispredictions, even if the branch target did not execute. We then use these methods to study some unconventional sources of branch mispredictions that we also discuss.

5.1 Observation Channels

Figure 3 provides an overview of the pipeline stages that we consider and their respective observation channels. Although performance counters can keep count of the number of mispredicted branches, they do not indicate how far mispredictions advance in the pipeline. Therefore, relying on performance counters that measure branch mispredictions is insufficient for this purpose.

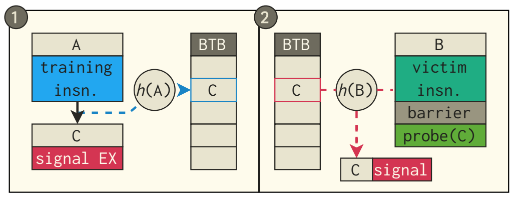

Fig. 4: In ❶, A creates a BTB entry to C, so that in ❷, the victim instruction of B may reuse that BTB entry. The instructions in C emit a transient execution signal. By fetching and decoding C, transient fetch and transient decode signals are already emitted.

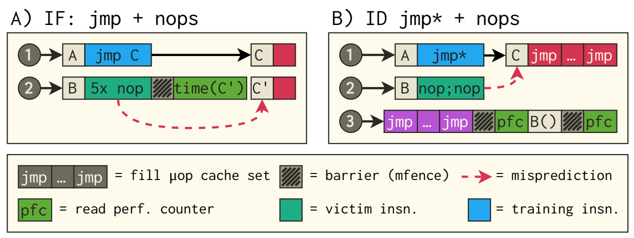

Fig. 5: A) Training *non branch* using (direct) *jmp*, measuring IF: C' is at an address at the same relative offset from B as C from A. B) Training *non branch* using (indirect) *jmp**, measuring ID: In ❸, the *jmp*-series will evict the *jmp*-series in C, resulting in µop-cache misses when executing B.

The generic procedure is as follows. We have two snippets of code A and B, where A ends with a training branch to C. In the example shown in Figure 4, C emits a signal to indicate transient execution, but before that, it will also signal transient fetch and transient decode. We want to observe that running B after A causes a misprediction to C that emits one or more of these signals. Hence, A is the *training source* and B is the *victim source*. To observe misprediction to C, after ❶, we moreover prime the microarchitectural state that we use as observation channel, for example flushing C from the instruction cache (I-Cache). In ❷, after the victim instruction, a *barrier* instruction, such as *lfence*, ensures that older operations have completed when we probe for a signal.

Previous work discusses how A and be B should be laid out in memory and executed to trigger BTB index aliasing (i.e., where $h(A) = h(B)$), resulting in branch target misprediction [9, 36, 74]. However, such BTB index aliasing has never previously been shown between user- and kernel-mode branches for newer microarchitectures such as AMD Zen 3 and Zen 4. To build a practical user-to-kernel exploit, we need BTB aliasing across privilege modes. Although user space BTB aliasing is sufficient for the purposes of building our observational channels, we will discuss our reverse engineering efforts to enable cross-privilege mode BTB aliasing on newer AMD microarchitectures in Section 6. We now discuss the construction of our observation channels.

Instruction Fetch (IF). Observing IF of a particular branch target can be done by observing the I-cache state using a timing side channel. Hence, our method to observe IF involves timing the execution time of C as shown in Figure 5 A. After ❶, we flush C from I-cache, and in ❷, we probe C by timing access of an instruction that resides in it.

Unfortunately, discerning IF from BPU-assisted I-cache prefetching [78] (e.g., due to spatial locality) is not possible using this method. I-cache prefetching ensures that instruction bytes are available before IF. To get a stronger indication that the prefetched instructions enter the pipeline, we construct an observation channel that lets us detect ID.

Instruction Decode (ID). Instructions decoded into µops are cached in the µop-cache. To detect decoded instructions, we therefore build an observation channel of the µop-cache using performance counters based on methods proposed in previous work [60]. We sample the performance counters *de_dis_uops_from_decoder.opcache_dispatched* on Zen 2, *op_cache_hit_miss.op_cache_hit* on Zen 3 and 4 and *idq.dsb_cycles* on Intel. By observing these events we find that these caches always have 64 8-way sets, selected by the lower 12 bits of the instruction's virtual address. Following the procedure illustrated in Figure 5 B, after ❶, we have primed a particular µop-cache set by executing a *jmp-series* of 7 direct forward branches separated by 4096 bytes. In ❷, we have allocated B so that it collides with the BTB entry created for A to C. In ❸, we select a *jmp-series* that maps to a µop-cache set that matches the set that C maps to. When we then execute B in ❸, if a misprediction to C happened, it results in eviction of one or more ways of the set, which is observable using performance counters. We can sample the counter before and after executing B, as the diagram shows. Alternatively, we can observe µop-cache misses after B by executing the purple-filled jmp-series again, sampling the counter before and after it. Measuring, µop-cache misses is less reliable on our Intel parts than on AMD. To accomplish reliable results, we use complementary negative testing using a training branch that does not alias with the victim branch. Only when we measure significantly more µop-cache misses compared to the negative test do we conclude that the mispredicted target advanced to ID.

Execute (EX). Previous work has presented several methods to observe transient *execution*, through caches [36] and port contention [3, 11]. While observing execution port contention is possible, the signal is less reliable than observing memory access. Typically, Spectre gadgets leak memory through transiently executing two memory accesses, where the second access depends on the result of the first. Hence, the mispredicted control flow not only reaches EX, but remains in this stage over several clock cycles, allowing a speculation window that is wide enough to execute a second load. With shorter speculation windows, resteer typically happens before any memory operation completes. However, dispatching a single memory operation consumes only a few clock cycles, and there is no mechanism to abort a dispatched memory request since merely fetching a cache line is considered harmless. We can hence rely on a data cache side channel to detect if a misprediction reaches the EX stage by using a memory access in the mispredicted path.

5.2 Triggering mispredictions

With the methods to observe short speculation windows, we want to explore the possible combinations of training and victim instructions. We consider the following instructions: indirect branch (*jmp**), direct branch (*jmp*), conditional

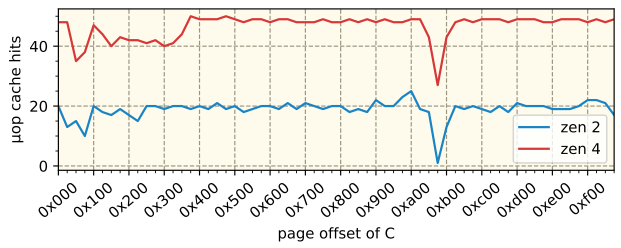

Fig. 6: Detecting speculative decode. Mispredictions caused by training *non branch* using *jmp** is observable in the µop-cache. Only when we place *C* at the page offset that matches the *jmp-series* in *B* (here *0xac0*), we see µop-cache misses.

Training	Victim instruction: *jmp**	*jmp*	*jcc*	*ret*	*non branch*
*jmp**	[a]	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕[b]	●●◕◕ ◕◕◕◕
jmp	●●◕◕ ○◕◔◔	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕
jcc	●●◕◕ ◕◕◔◔	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕
ret	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	—	●●◕◕ ◕◕◕◕
non branch[c]	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	●●◕◕ ◕◕◕◕	—

◔: IF ◕: ID ●: EX ◑: AMD Zen 1 ◑: Zen 2 ◑: Zen 3 ◑: Zen 4
◑: Intel 9th gen. ◑: 11th gen. ◑: 12th gen. (P core) ◑: 13th gen. (P core)
[a] Spectre-V2 [36] [b] Retbleed [74]. [c] Spectre-SLS [7, 72].

Tbl. 1: Various combination of training and victim instructions and how far they reach in the pipeline. The asymmetric combinations here, we refer to as PHANTOM speculation.

branch (*jcc*), return (*ret*), and nop-sled (*non branch*). The asymmetric combinations of these comprise 22 possible variants to evaluate on each microarchitecture. We consider training *jmp* and *jcc* branches with different displacement than the victim *jmp* or *jcc* as asymmetric as well. Some of these combinations are new, and some have been explored in previous work which we discuss next.

Training using *non branch* instructs the branch predictor that the victim instruction is not a branch, and results in the CPU executing the next instruction in sequence, even if the current instruction is actually a branch. This has been reported as Straight-Line-Speculation on some AMD microarchitectures [7, 72].

Training using *jmp* has, to the best of our knowledge, only been explored in previous work on Intel processors [78]. Concurrent work has studied *jmp* also on AMD processors [48]. Since the speculation may happen pre-decode, even though the PC-relative displacement is available as part of the branch instruction, the target is served from the BTB. Unlike indirect branch targets, the branch predictor serves direct branch targets as PC-relative. This means that for this combination, the signal does not become observable at *C* (based on Figure 5 A). Instead we create a copy of *C* to *C′*, which we allocate to an address that has the same relative distance from the victim instruction as *C* has from the training instruction.

Training using *jmp** has been explored in the original Spectre work [36]. In this paper, we are interested in the *jmp** instruction's impact on the other victim instructions (i.e., *jmp*, *jcc*, *ret*, and *non branch*).

Training using *ret* instructs the branch predictor that the victim instruction is a return, and the branch predictor will therefore predict a return from *B* when we execute it. The return target will not be to *C*, but to the most recent call site.

An important observation we made while designing these experiments is that *the training instruction always determines the prediction semantics of the victim instruction*. This could be because the victim instruction may not have been decoded when providing the prediction. In the next section, we will investigate how far in the pipeline these combinations reach and what exploitation primitives they enable.

6. EXPLOITATION PRIMITIVES

Table 1 shows the results of our experiments. We draw a number of interesting observations from these results. First, for all tested combinations, fetch and decode of the predicted target happens. This happens even in the absence of an architectural branch at that location (e.g., when the victim instructions are *nops*, used for the *non branch* case). We can thus conclude that the frontend fetches branch targets *before* it has even determined whether a branch exists. This leads to our first observation:

Observation O1.

On all tested CPUs, speculative branch targets are fetched before the branch source is decoded.

Moreover, our results show that instructions at speculative branch targets are decoded as well, even in the absence of any branch source. As an example, Figure 6 shows the results of the ID observation channel, when a *non branch* is confused with a *jmp**. Thus, our second observation is:

Observation O2.

On all tested CPUs, the speculative branch target fetches we are usually not only prefetched, but they enter the CPU pipeline.

We note that our results for some of our Intel parts do not indicate ID, and sometimes not even IF, in certain scenarios where the victim instruction is *jmp**. While this suggests that indirect branches are not subject to PHANTOM speculations, we cannot reject the hypothesis that this victim branch type has unknown effects on the accuracy of our measurements.

On AMD Zen 1 and Zen 2, instructions at the target even reach the execute stage. On these microarchitectures, we measure a D-cache hit on the address loaded from memory by the instructions at the speculative target. Our third observation therefore is:

Observation O3.

On AMD Zen 1 and Zen 2, decoder-detectable speculations yield windows long enough to execute code.

We occasionally observe transient execute after a taken conditional branch (*jcc*). This occurs when training it as *non branch*. This is likely due to the conditional branch sometimes being predicted non-taken, therefore unrelated to the training. All processor frontends will fetch and decode instructions following the victim branch in the current fetch block (typically 32 B) of instructions.

PHANTOM on Intel. Table 1 further shows the results of our experiments on a number of recent Intel processors. Most scenarios allow for transient fetch and decode. While these insights show similarity in designs from different vendors, Intel processors have eIBRS protection against cross-privilege attacks since the 9th generation. Moreover, the Intel processors we tested do not re-use a user-injected prediction in kernel mode, even while the mitigation is switched off. This suggests that these processors may address the BTB differently depending privilege mode, complicating exploitation. We therefore limit our focus to the AMD parts for exploitation.

6.1 Attacker primitives

Our results reveal two channels that can be used for exploitation. First, we can trigger speculation on arbitrary instructions that results in fetching and decoding the target. Second, on certain AMD microarchitectures, we can trigger PHANTOM speculations that fit a memory load, even on non-branch instruction.

We identify these channels to give rise to three adversarial exploitation primitives. This section describes these primitives in detail. In Section 7, we discuss how we build exploits using these primitives.

P1: Detecting mapped executable memory. An instruction fetch only populates the instruction cache if the target of the fetch was executable and backed by physical memory. Combining this insight with PHANTOM, we can detect whether a virtual address T is mapped and executable. The adversary would first ❶ train the BTB with branches to T, ❷ execute the victim and ❸ infer whether T was fetched. For the last step, the adversary can use Prime+Probe [52] on the instruction cache.

P2: Detecting mapped non-executable memory. If our target T is mapped but not executable, the fetch fails and would leave the state of the instruction cache unaffected. Using PHANTOM on Zen 1 and Zen 2, however, an adversary can trigger a data load of T. To detect mapped non-executable memory, the victim's address space needs to contain a disclosure gadget G that loads T from memory. Then, they would ❶ train the BTB with branches to G, ❷ execute the victim with T in a register and ❸ infer whether T was loaded from memory. To infer the data load, Prime+Probe can be used on the data cache. This primitive works on AMD Zen 1 and Zen 2 only.

P3: Leaking register values. Lastly, instead of detecting mapped memory, an adversary can use the short speculation window to leak the victim's register values on AMD Zen 1 and Zen 2. G filters out a single byte from the register and arranges it to reside in bits [13:6] (i.e., cache-line aligned), which it uses as offset into a mapped area in the victim's address space, and issues a load of the resulting address. An adversary would ❶ train the BTB with a branch to G, ❷ execute the victim and ❸ infer which address was loaded in the data cache using Prime+Probe.

Alternatively, if an adversary shares memory with the victim, they can use Flush+Reload [77] instead. G needs to shift the value to become a 64-byte aligned offset, which can be as big as the memory area which now must be shared with the adversary (e.g. physmap [35]). Thus, an adversary would ❶ train the BTB with a branch to G, ❷ execute the victim so that victim register-dependent memory ends up in the data cache, and ❸ infer which address was loaded in ❷ using Flush+Reload.

6.2 Collision with kernel addresses

Wikner and Razavi showed that triggering a misprediction on a kernel address can be achieved from user space by branching to a kernel address and catching the resulting page fault [74]. In order to collide with the desired kernel address, they reverse engineer BTB indexing functions. However, they did not discover cross-privilege functions on AMD Zen 3. Furthermore, more the recent AMD Zen 4 microarchitecture has not been studied on this topic in previous work. To evaluate our primitives and build exploits using them, we need to reverse engineer the cross-privilege BTB indexing functions on these newer microarchitectures.

We start on Zen 3 by allocating a kernel address K, using a kernel module which contains *nops* followed by a return instruction. By changing the Page Table Entry (PTE) attributes of address K, we make it accessible to user space.

Brute forcing. We first attempt to create collisions with K by brute forcing a pattern such that, when applied to the kernel address K, it yields a user space address that collides with K, as done in [74]. Using performance counters and timing results, we determine whether a collision was successful, However, this approach does not yield any results between user- and kernel addresses when flipping up to 6 bits. A possible reason of failing to find collisions could be that bit 47 is involved in multiple functions, requiring us to flip more bits. Since brute forcing all combinations with more than 6 bits takes an unreasonable amount of time, we consider an alternative approach.

SMT solver. Instead, we will generate random addresses to find collisions between user- and kernel addresses, and then observe patterns in the addresses that collide. For this, we use a Z3 SMT solver, as done in previous work [43]. For each kernel address K, we collect lists L_K of user space addresses that collide with the kernel address. To shrink the search space, we do not randomize the lower twelve bits of our user space addresses. Instead, we set them equal to K_{0-11}. We wish to find functions of address bits, such that they all yield the same value for K and all addresses in L_K. For this, we attempt to find coefficients for the equation system $(x_0 \times A_0) \oplus (x_1 \times A_1) \oplus \ldots \oplus (x_{46} \times A_{46}) \oplus (1 \times A_{47}) = y$ such that it yields the same value y for all addresses that collide. At the same time, we impose $x_0 + x_1 + \ldots + x_{46} + x_{47} \leq n$, where n is the maximum number of coefficients set to 1, which we gradually increase. This is to prevent solutions that combine different solutions to obtain the same output y.

Results. Our results are shown in Figure 7 and were found for $n = 4$. Specifically, we find that whenever b_{13} is toggled in the random-generated user space address with respect to K, b_{17} is toggled as well. Likewise, whenever b_{12} is toggled,

$$
\begin{aligned}
f_0 &= b_{47} \oplus b_{35} \oplus b_{23} & f_1 &= b_{47} \oplus b_{36} \oplus b_{24} \oplus b_{12} \\
f_2 &= b_{47} \oplus b_{37} \oplus b_{25} \oplus b_{13} & f_3 &= b_{47} \oplus b_{38} \oplus b_{26} \oplus b_{14} \\
f_4 &= b_{47} \oplus b_{39} \oplus b_{26} \oplus b_{13} & f_5 &= b_{47} \oplus b_{39} \oplus b_{27} \oplus b_{15} \\
f_6 &= b_{47} \oplus b_{40} \oplus b_{28} \oplus b_{16} & f_7 &= b_{47} \oplus b_{41} \oplus b_{29} \oplus b_{17} \\
f_8 &= b_{47} \oplus b_{42} \oplus b_{30} \oplus b_{18} & f_9 &= b_{47} \oplus b_{43} \oplus b_{31} \oplus b_{19} \\
f_{10} &= b_{47} \oplus b_{44} \oplus b_{32} \oplus b_{20} & f_{11} &= b_{47} \oplus b_{45} \oplus b_{33} \oplus b_{21}
\end{aligned}
$$

Fig. 7: Functions for creating cross-privilege collisions in the BTB found on Zen 3. Least significant 12 bits not considered.

b_{16} is flipped as well, and vice versa. In essence, that means that these bits are used in multiple, partially overlapping functions. Therefore, we erroneously obtained functions almost identical to the ones presented.

Comparing our results with those in [74], we see that all functions additionally include bit 47. However, we did not find some of the functions, potentially because they do not involve bit 47. We also find some functions that were previously not discovered.

Overlapping functions. While trying to create collisions with kernel addresses by flipping multiple bits, we discovered that using other lower bits shown in Figure 7 does not yield colliding addresses. We suspect that this is due to overlapping functions, just as b_{12}, b_{13}, b_{16} and b_{17} are used in multiple functions. These functions may not involve bit 47, or use address bits we did not consider. One reason for overlapping functions could be that some functions are used for tag generation, while others are used for set selection. Therefore, to create collisions, we use the higher bits (i.e., the first three bits of each function). As an example, for a kernel address K, we can obtain a user-colliding address by computing $K \oplus$ `0xffffbff800000000` or $K \oplus$ `0xffff8003ff800000`. We confirm both of these patterns to work on AMD Zen 4 as well.

6.3 SuppressBPOnNonBr and AutoIBRS

In response to our report, AMD disclosed a configuration of Zen 2 CPUs that should prevent speculation from non-branch instructions. By setting bit in MSR `0xC00110E3` named SuppressBPOnNonBr, PHANTOM speculations should be prevented. In addition, Zen 4 CPUs support AutoIBRS, which restrict speculation to be influenced across privilege levels. In this section, we discuss the implication of setting the SuppressBPOnNonBr bit and enabling AutoIBRS on our results.

SuppressBPOnNonBr. We first measure the overhead of setting this bit using UnixBench[1]. We run each benchmark 5 times and compute the geometric mean across all tests. Our UnixBench results indicate an overhead of 0.69% (single-core) and 0.42% (multi-core).

To understand the impact of setting this bit on our results, we repeat the experiments as described in Section 5.1 with this bit enabled. As expected, our results show that whenever the victim instruction is of type non-branch, we do not observe execution at the predicted target anymore. However, we find that this bit *does not* prevent IF or ID when the victim instruction is not a branch.

[1]`https://github.com/kdlucas/byte-unixbench`

µarch	Model	Accuracy	Rate
Zen	AMD Ryzen 5 1600X	96.30%	204 bits/s
Zen 2	AMD EPYC 7252	93.04%	215 bits/s
Zen 3	Ryzen 5 5600G	100%	256 bits/s
Zen 4	Ryzen 7 7700X	90.67%	341 bits/s
Zen	AMD Ryzen 5 1600X	100%	256 bits/s
Zen 2	AMD EPYC 7252	99.28%	292 bits/s

Tbl. 2: Accuracy and leakage rate of **P1** (top) and **P2** (bottom) when leaking 4096 bits (median of 10 runs).

Observation **O4.**

SuppressBPOnNonBr does not prevent IF or ID caused by PhantomJMPs.

AutoIBRS. We repeat the experiments in Section 5.1 on Zen 4. However, we train in user space while we try to trigger a speculative branch in kernel space. Interestingly, our results show that the IF is still triggered, despite AutoIBRS.

Observation **O5.**

AMD AutoIBRS does not prevent IF of cross privilege mode branch targets.

Our previously described primitive **P1** is thus unaffected on all AMD Zen microarchitectures. Primitives **P2** and **P3**, however, are now restricted to speculation on branch instructions on AMD Zen 2, thanks to the `SuppressBPOnNonBr` mitigation. However, given that branches are common in software, the impact of this mitigation is negligible. In addition, **P2** and **P3** still work unrestricted on Zen 1.

6.4 Covert Channel

Our primitives **P1** and **P2** enable an adversary to trigger a fetch and, on some microarchitectures, even a data load (i.e., execute) from a branch instruction, even if its in a higher privilege mode. In this section, we investigate the accuracy and leakage rate of these primitives. For this, we build a kernel module that performs a number of direct branches. We aim to hijack one of these by injecting a prediction from user mode, that triggers when executing kernel module branches.

Fetch. We randomly generate 4096 bits. In the kernel address space, T_1 is mapped in memory while T_0 is not. For each random bit b, ❶ we prime a chosen instruction cache set S, ❷ inject a prediction to T_b which maps to cache set S, ❸ invoke the kernel module and ❹ probe cache set S. If our probe step indicates a higher latency after injecting a prediction to T_1 than to T_0, we deduce that b was 1, otherwise 0. To improve accuracy, we inject the speculative branch so such that it straddles a page boundary and furthermore stress the sibling thread [2]. The results can be seen in Table 2-top.

Execute. Again, we randomly generate 4096 bits. However, an additional address T is mapped executable in kernel mode, containing a memory load of the address in register **R**. For each random bit b, ❶ we prime a chosen data cache set S, ❷ inject a prediction to target T, ❸ invoke the kernel module

[2]We use `stess -c 10`. Other workloads work too

μarch	Model	Accuracy	Median time
Zen 2	AMD EPYC 7252	97%	4.09 s
Zen 3	Ryzen 5 5600G	100%	1.38 s
Zen 4	Ryzen 7 7700X	95%	1.23 s

Tbl. 3: Accuracy and median time needed to derandomize kernel image location on AMD Zen microarchitectures using **P1**, over 100 runs.

while **R** is assigned a pointer to T_b, and ❹ probe data cache set S. If we observe a slowdown when probing S, we deduce that b was 1, otherwise 0. Additional sibling thread workloads were unnecessary for the tested parts. Table 2-bottom shows our results.

7. EXPLOITATION

We build three exploits using the primitives we discussed in Section 6 to break kernel image KASLR, physmap KASLR and consequently leak kernel data. Recent KASLR exploits on AMD microarchitectures do not fully derandomize the address [43], do not consider the newer Zen 3 and Zen 4 CPUs [42, 43], only work in the absence of KPTI [42] or focus exclusively on kernel image KASLR [43]. KASLR is an important defense against memory corruption attacks in the kernel, since it prevents the knowledge of specific gadget addresses [14].

We use **P1** to leak the kernel image location in Section 7.1, and **P2** to leak kernel's physmap location in Section 7.2. These two attacks rely on Prime+Probe [52] which turns out to be noisy. We discuss how we can overcome this in Section 7.3. Finally, in Section 7.4 we show how **P3** extends the attack surface of Spectre with new gadgets.

7.1 Breaking kernel image KASLR

We show how we can derandomize kernel image KASLR on AMD microarchitectures with PHANTOM speculation. We run Linux kernel 5.19 with the latest patches.

```
1 nop     DWORD PTR [rax+rax*1+0x0]
2 push    rbp
3 mov     rbp,rsp
```

Lst. 1: We trigger speculation at the nop instruction in `__task_pid_nr_ns()`. Found at kernel image offset 0xf6520.

We can break kernel image KASLR with **P1**. KASLR places the kernel image in one of 488 possible locations [40]. For each possible location, we inject a *jmp** prediction to a branch target that maps to a specific instruction cache set that we choose. The `getpid()` system call will execute the code shown in Listing 1, where we inject a prediction at the *nop* instruction on Line 1. Hence, for each possible location, ❶ we prime the chosen cache set, ❷ inject the *jmp** prediction, ❸ issue `getpid()`, and ❹ probe the cache set. If the prediction was used and the branch target was mapped in memory, we can observe a cache signal from the chosen cache set through Prime+Probe. This happens when testing with the correct kernel image location.

Results. We run our KASLR exploit 100 times on our AMD Zen machines, each time rebooting the machine to refresh KASLR. Table 3 presents the success rate and median time needed to derandomize the kernel image location.

μarch	Model	Accuracy	Median time
Zen	AMD Ryzen 5 1600X	100%	101 s
Zen 2	AMD EPYC 7252	90%	106.5 s

Tbl. 4: Accuracy and median time needed to find physmap on a AMD Zen 2 microarchitecture using **P2**, over 10 runs.

μarch	Model	Memory	Accuracy	Median time
Zen	AMD Ryzen 5 1600X	8 GB	99%	1 s
Zen 2	AMD EPYC 7252	64 GB	100%	16 s

Tbl. 5: Accuracy and median time needed to find a physical address on AMD Zen 1/2 microarchitectures, over 100 runs.

7.2 Breaking physmap KASLR

Now that we have found the kernel image location, we can further derandomize physmap KASLR on AMD Zen 1 and Zen 2. Physmap is the direct mapping of physical memory in the kernel address space, and has, depending on configuration, 25 600 possible locations [40]. Randomizing its location helps preventing certain memory corruption attacks [35].

As mentioned in Section 6.1, we can only detect mapped memory using the speculative instruction fetch if the target is executable. However, physmap is marked non-executable. To derandomize physmap, we use **P2**, which detects mapped non-executable memory by detecting a transient load in the PHANTOM speculation window using Prime+Probe on the L2 data cache. For Prime+Probe on L2, we use 2 MiB physically contiguous transparent huge pages.

```
nop     DWORD PTR [rax+rax*1+0x0]
push    rbp
mov     esi,0x4000
mov     rbp,rsp
sub     rsp,0x8
call    0x9341c7b0
```

Lst. 2: We trigger speculation at the call instruction, upon entering `__fdget_pos()`. Found at kernel image offset 0x41db60.

```
mov     r12,QWORD PTR [r12+0xbe0]
```

Lst. 3: Our disclosure gadget to leak the physmap location. Found at kernel image offset 0x41da52.

By using the tooling from previous work [74], we find that upon executing the `readv()` system call, we control the value of **R12** using the second argument of the system call (i.e., **RSI**) when `__fdget_pos()` is called. We trigger speculation by confusing the call instruction shown in Listing 2 with a *jmp** prediction to the disclosure gadget shown in Listing 3. The steps of the attack are very similar to our kernel image KASLR derandomization exploit, except that the EX step of **P2** allows us to issue a load that we can detect with Prime+Probe.

Results. We run our physmap derandomization exploit 10 times on our vulnerable AMD machines, each time rebooting the system. Table 4 shows the success rate and median time needed to derandomize the physmap location.

7.3 Overcoming Noise

Prime+Probe proves to be very noisy, especially on the L1 instruction cache. This may be due to the cache replacement policy or because the system call thrashes the chosen cache set before we can probe it. Instruction prefetching may contribute to this as well.

To improve the results, we repeat our exploit for multiple cache sets. For each set S, in addition to measuring the probing time when the injected target maps to the given set (T_S), we also measure the time when it maps to some unrelated set. This gives us a baseline time for the monitored set, B_S. For the kernel image derandomization exploit, we score each possible location by using a bounded relative timing difference between speculatively branching to addresses mapping to the primed cache set and the baseline, accumulated for all 64 sets. That is, $score_{guess} = \sum_{S=0}^{S \leq 64} min(max(T_S - B_S, -10), 10)$. To amplify the difference, we trigger another speculative branch along the execution path of the system call to an additional target mapped to S.

7.4 Leaking kernel memory

We now discuss how **P3** extends the attack surface of Spectre with new gadgets. First, to leak kernel memory, we need to find the location of a Flush+Reload buffer (reload buffer) in physmap.

Enabling Flush+Reload. We use the attacks in Section 7.1 and Section 7.2 to leak the kernel image and physmap locations. To enable Flush+Reload, we make a guess, P_g of the physical address of a virtual address A in our user mode program. We use the same setup as described in Section 7.2, meaning we trigger speculation during the `readv()` system call. We pass physmap + P_g in **RSI** to the system call. We can verify if P_g is correct using Flush+Reload on address A. To reduce entropy, we allocate A as a 2 MiB transparent huge page. Our results show that we can successfully determine the physical address of a virtual address in our program.

We attempt to determine the physical address of A 100 times. To re-randomize the physical address of A in each attempt, we allocate a random number (0–99) of huge pages before allocating A. Table 5 presents the accuracy and median time observed.

Leaking memory with MDS gadgets. In this section, we discuss how limited Spectre gadgets, referred to as MDS gadgets in previous literature [34], can be combined with our **P3** to leak arbitrary kernel memory. A conventional Spectre gadget performs two loads: one that fetches the secret from memory and one which encodes it in a reload buffer with a secret-dependent offset. With **P3**, however, we are able to trigger the secret-dependent load elsewhere. A gadget that only performs one out-of-bound load would thus be enough to enable arbitrary read capabilities.

```
1 void read_data(uint64_t user_index) {
2     if (user_index < *array_length) {
3         uint8_t data = array[user_index]
4         parse_data(data);
5     }
6 }
```

Lst. 4: A sample MDS gadget.

To prove the feasibility of such an exploit, we build a kernel module that contains an MDS gadget. Listing 4 shows what such a gadget might look like. When the user provides an out-of-bounds value to `read_data()`, the conditional branch may be incorrectly predicted as taken, causing a user-controlled address to be fetched from memory. A conventional Spectre attack would not succeed, however, since there is no `data`-dependent load. Our goal is to induce this secret-dependent load using **P3**.

We assume we know where our MDS gadget resides in the kernel address space. We also assume we know the start of physmap, the physical address of our reload buffer, and the kernel address of a disclosure gadget that performs the secret-dependent load. All this information can be leaked with our previous steps. We also assume that the address of `array` (Line 3 of Listing 4) is known. The user provides the kernel module with `user_index` and the location of our reload buffer in the kernel's virtual address space.

Relying on BTB aliasing, we train the conditional branch to predict taken. Additionally, we train the BTB with a branch to the disclosure gadget at the location of the (direct) call to `parse_data()`. Our disclosure gadget indexes into our reload buffer using the (shifted) value of `data`.

Results. We run our proof-of-concept on an AMD Zen 2 EPYC 7252. Our results show that we can reliably leak 4096 bytes of randomized data from the kernel using an MDS gadget. We repeat our experiment 10 times, each time after a reboot. In 8 of these attempts, we measure a median bandwidth of 84 bytes/s, achieving a perfect accuracy of 100%. In the remaining 2 attempts, no signal was observed. One possible explanation could be an undesired BTB aliasing.

Finding MDS gadgets. This work focuses on the analysis of frontend speculation and not the discovery of gadgets. Previous work shows how one can find MDS-like gadgets in the kernel [34]. Furthermore, new gadgets are continuously discovered and patched as shown recently by Google [79].

8. MITIGATION

We discuss the hardware- and software-based mitigation strategies for the type the exploits that we built, including mitigations that AMD proposed. The mitigations therefore mainly concern attacks against the OS kernel by an unprivileged user, described in finer detail in Section 3.

8.1 Hardware mitigations

SuppressBPOnNonBranch. AMD introduced an MSR bit *SuppressBPOnNonBranch* that, when set, limits branch prediction to control-flow edges [5]. This means that *SuppressBPOnNonBranch* should not allow the *nops* column in Table 1 to proceed in the pipeline. Our evaluation, however, paints to two problems: ① *SuppressBPOnNonBranch* is not supported on AMD Zen (+) and ② on AMD Zen 2, the *SuppressBPOnNonBranch* does *not* stop branch *prediction*. As we showed in Section 6, the invalid branch target still advances through IF and ID. However, we confirm that it stops the transient *execution*. Only stopping transient execution at non-branches means primitives **P1** is unaffected, and **P2** and **P3** still work if targeting a victim instruction that is a control-flow edge.

AutoIBRS. Our evaluation suggests that AutoIBRS has the same issue as the *SuppressBPOnNonBranch* mitigation in the sense that the misprediction is only prevented after ID, which again means that primitive **P1** is unaffected. This would be necessary if AutoIBRS follows the original IBRS specification, which, as the name suggests, concerns only *indirect branch* speculation. A possible enhancement for AutoIBRS could be that the branch predictor refuses to serve any prediction where the current privilege mode mismatches the privilege mode specified by the prediction.

An in-depth mitigation in this direction should stop predictions until the decoding of the branch source has finished, thereby preventing all branch type confusions. However, we presume that such an approach would require fundamental design changes and performance impact, rendering it unfeasible in practice.

8.2 Software mitigations

lfence. AMD has also continued to recommend using *lfence* at sources of bad speculation, however as we discussed earlier in Section 2.4, finding all possible sources of bad speculation is not trivial. Because proactively placing speculation barriers behind every conditional branch has an average performance impact of 5× [69], determining the branches where speculation barriers are necessary is in practice often a semi-manual process. Although automatic tools have been proposed [24, 34, 50, 69, 79], most of them do not achieve completeness. Furthermore, short speculation expands the set of possible gadgets that future tools of this type should consider.

IBPB. Indirect Branch Prediction Barrier (IBPB) is a mechanism supported by all recent x86 CPUs that is used to flush the BTB state when switching between distrusting execution contexts. On some microarchitectures, IBPB flushes more than just indirect branch predictions. As such, if IBPB flushes all types of predictions, it is possible to use it when switching from user mode to kernel mode to ensure that user mode cannot cause PHANTOM speculation in the kernel context. The problem with IBPB is its large performance penalty. Assuming that IBPB can flush all types of predictions, it mitigates all our exploitation primitives **P1**, **P2**, and **P3**.

9. RELATED WORK

We discuss related work on microarchitectural side channels and transient execution attacks.

9.1 Microarchitectural info leaks

Kocher et al. [37] discuss that microarchitectural optimizations, like branch prediction, caches, variable-time instructions (e.g., division and multiplication) are workload dependent. Using timing side channels on these components, secret bits of information about the workload can be inferred. A recent study by Ramhöj et al. systemizes various microarchitectural attacks over the past few decades [26].

Shared cache leaks. Caches are the most studied component of microarchitectural side channels. Percival [54] showed how co-located sibling threads could infer each other's memory interactions. Osvik et al. [52] introduced Prime+Probe and Evict+Time naming, but the similar techniques were in previous cache attacks as well [10]. Prime+Probe was adapted to web browser settings by Oren et al. [51] and Gras et al. [22] showed Evict+Time to leak pointers from within the browser sandbox. Yarom et al. [77] introduced the more noise-resistant Flush+Reload technique in shared memory scenarios. Cache attacks continue to advance along with modern CPU caches [17,23,43,56]. Advanced cache attacks also considers ring and mesh interconnects [16,53,68].

Branch Predictor side channels. Aciiçmez et al. [2] showed leaking secrets through Simple Branch Prediction Analysis to exfiltrate cryptographic keys. Evtyushkin et al. [18, 19] showed branch prediction can be abused to break KASLR and infer control flow inside SGX. Lee et al. [41] used branch shadowing to infer control flow inside SGX.

Beyond cache and branch predictor side channels, leakage through variable instruction timing [6, 38, 62], port contention [3, 11], frequency [42, 70], and power use [44] have been studied in the past.

9.2 Transient execution attacks

Spectre [36] and Meltdown [45] combined microarchitectural side-channels with invalid speculative and Out-of-Order (OoO) execution, creating the new class of transient execution attacks. Meltdown triggers invalid OoO execution through a faulting memory load. Other fault-based transient execution attacks include [12, 13, 58, 59, 66, 67].

Spectre enables info leaks through invalid speculative execution triggered through the manipulation of branch target predictors [9, 33, 36, 48, 74], return target predictors [39,46,73,74], branch condition predictors [21,33,36,49], and the memory disambiguator [27].

Wikner and Razavi [74] showed branch target misprediction of return instructions by training them using indirect branches. Wieczorkiewicz [71, 72] reverse engineered AMD branch predictors showing branch condition and straight-line misprediction vulnerabilities. We did a systematic analysis of branch mispredictions due to early speculation and discovered new sources of information leakage.

Milburn et al. [48] showed that particular SMT workloads can extend the speculation window which bypassed the AMD's original, *lfence*-style retpoline [4]. Ren et al. [60] introduced techniques to reverse engineer the µop-cache and potential methods to leak secrets across *lfence* barrier.

Zhiyuan et al. [78] manipulates the BTBs to trigger instruction prefetching on Intel CPUs. Unlike this concurrent work, the mispredictions we trigger lead to the fetched instructions to enter the pipeline with intent to execute as we showed using a µop-cache observation channel.

9.3 Mitigations

Mitigation of potentially exploitable branch sources by inserting speculation barriers has been explored in previous work [28, 47]. It is possible to leverage binary or source analysis for discovering exploitable Spectre gadgets [24, 25, 34, 50, 57, 69, 74]. In particular, Johannesmeyer et al. [34] explored dynamic analysis through fuzzing combined with speculative emulation to automatically discover Spectre and MDS leaks in the Linux kernel. In their work, conventional Spectre V1-gadgets dereference an attacker-controllable pointer and subsequently encode the result in the cache (e.g., two dependent loads). As we showed in this paper, PHANTOM can leak arbitrary data using only a single load (similar to MDS-gadgets), by triggering short speculation to a separate gadget that dispatches the second load. Based on the results presented in Kasper [34], PHANTOM increases possible Spectre gadgets by about 4 times (from 183 to 722).

10. CONCLUSION

We introduced PHANTOM, a new class of attacks that arise from early speculation in recent AMD and Intel microarchitectures. PHANTOM allows for triggering speculation on arbitrary instructions with speculation windows that are short, but still allowing for *transient fetch* and *transient decode* on almost all tested microarchitectures, and *transient execution* of a single memory operation on AMD Zen 1 and 2. We used these new PHANTOM primitives in the construction of three attacks: leaking code and data KASLR on all AMD Zen microarchitectures, as well as arbitrary kernel memory with MDS gadgets on AMD Zen 1 and 2. The attacks on AMD Zen 3 and 4 required us to reverse engineer the cross-privilege BTB functions for the first time. Our analysis of existing hardware and software mitigations shows that mitigating PHANTOM is going to be difficult in practice.

Acknowledgments

We thank the anonymous MICRO reviewers, David Kaplan of AMD, and Brad Spengler of Open Source Security Inc., for their feedback. We furthermore thank the members of the Linux security response group of Retbleed for insights. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

REFERENCES

[1] "The Chromium Projects: Site Isolation," accessed on 29.1.2022. [Online]. Available: https://www.chromium.org/Home/chromium-security/site-isolation/

[2] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 312–320.

[3] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[4] AMD, "Amd64 technology indirect branch control extension," 2018. [Online]. Available: https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf

[5] AMD, "Technical guidance for mitigating branchtype confusion," 2022, accessed on 1.8.2022. [Online]. Available: https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf

[6] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 623–639.

[7] ARM, "Straight-line speculation," 2020, accessed on 8.2.2023. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Security%20Update%2008%20June%202020/Straight-line_Speculation-v1.0.pdf

[8] T. Asheim, B. Grot, and R. Kumar, "A storage-effective btb organization for servers," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1153–1167.

[9] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks," in *USENIX Security*, 2022.

[10] D. J. Bernstein, "Cache-timing attacks on AES," The University of Illinois at Chicago, Tech. Rep., 2005.

[11] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS. Association for Computing Machinery, 2019, https://doi.org/10.1145/3319535.3363194.

[12] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *SEC*, 2018.

[13] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[14] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 481–493.

[15] Y.-J. Chang, "Lazy btb: reduce btb energy consumption using dynamic profiling," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006, pp. 917–922.

[16] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around:{Side-Channel} attacks and mitigations on mesh interconnects," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2857–2874.

[17] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+abort: A timer-free high-precision l3 cache attack using intel {TSX}," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.

[18] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–13.

[19] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 693–707.

[20] T. Gleixner, "LKML: [patch 00/38] x86/retbleed: Call depth tracking mitigation," 2022. [Online]. Available: https://lore.kernel.org/lkml/f9fd86acac4f49bc8f90b403978e9df3@AcuMS.aculab.com/t/

[21] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1871–1885.

[22] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu." in *NDSS*, vol. 17, 2017, p. 26.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Springer, 2016, pp. 279–299.

[24] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[25] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "Specusym: Speculative symbolic execution for cache timing leak detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1235–1247.

[26] N. R. Holtryd, M. Manivannan, and P. Stenström, "Sok: Analysis of root causes and defense strategies for attacks on microarchitectural optimizations," *arXiv preprint arXiv:2212.10221*, 2022.

[27] J. Horn, "Issue 1528: Speculative execution, variant 4: Speculative store bypass," 2018. [Online]. Available: https://bugs.chromium.org/p/project-zerot/issues/detail?id=1528

[28] O. S. S. Inc., "Respectre: The state of the art in spectre defenses," 2018. [Online]. Available: https://grsecurity.net/respectre_announce

[29] Intel Corp., "Indirect Branch Restricted Speculation," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

[30] Intel Corp., "Speculative Execution Side Channel Mitigations," 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html

[31] Intel Corp, "Retpoline: A branch target injection mitigation," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html

[32] Intel Corp, "Speculative Execution Side Channel Mitigations," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html

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

[34] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel," in *NDSS*, Feb. 2022. [Online]. Available: https://download.vusec.net/papers/kasper_ndss22.pdf

[35] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.

[36] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[37] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[38] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels." in *USENIX Security Symposium*, 2017, pp. 69–81.

[39] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, 2018, https://www.usenix.org/conference/woot18/presentation/koruyeh.

[40] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 309–321.

[41] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing." in *USENIX Security Symposium*, vol. 19, 2017, pp. 16–18.

[42] M. Lipp, D. Gruss, and M. Schwarz, "{AMD} prefetch attacks through power and time," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 643–660.

[43] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 813–825.

[44] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 355–371.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[46] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2018.

[47] Microsoft, "Spectre mitigations in msvc." [Online]. Available: https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/

[48] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection," *arXiv preprint arXiv:2203.04277*, 2022.

[49] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, "Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing," *arXiv preprint arXiv:2301.07642*, 2023.

[50] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1481–1498.

[51] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1406–1418.

[52] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*, 2006, pp. 1–20.

[53] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical." in *USENIX Security Symposium*, 2021, pp. 645–662.

[54] C. Percival, "Cache missing for fun and profit," 2005.

[55] K. Phillips, "LKML: [PATCH 0/3] x86/speculation: Support Automatic IBRS," 2022. [Online]. Available: https://lkml.org/lkml/2022/11/4/1199

[56] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2906–2920.

[57] Z. Qi, Q. Feng, Y. Cheng, M. Yan, P. Li, H. Yin, and T. Wei, "Spectaint: Speculative taint analysis for discovering spectre gadgets," in *Annu. Network and Distributed System Security Symp.(NDSS)*, 2021.

[58] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1451–1468.

[59] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CrossTalk: Speculative Data Leaks Across Cores Are Real," in *S&P*, 2021.

[60] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 361–374.

[61] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[62] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 279–299.

[63] J. Shahid and O. Weisse, "https://lwn.net/articles/909469/," 2022, accessed on 02.02.2023. [Online]. Available: https://lwn.net/ml/linux-kernel/20220223052223.1202152-1-junaids@google.com/

[64] D. Sneddon, "[PATCH 5.4 14/15] x86/speculation: Add RSB VM Exit protections," 2022. [Online]. Available: https://lkml.org/lkml/2022/8/9/728

[65] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[66] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020, pp. 1399–1417.

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

[68] J. Wan, Y. Bi, Z. Zhou, and Z. Li, "Meshup: Stateless cache side-channel attack on cpu mesh," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1506–1524.

[69] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2019.

[70] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 679–697.

[71] P. Wieczorkiewicz, "The amd branch (mis)predictor: Just set it and forget it!" 2022, accessed on 8.2.2023. [Online]. Available: https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it

[72] P. Wieczorkiewicz, "The amd branch (mis)predictor part 2: Where no cpu has gone before (cve-2021-26341)," 2022, accessed on 8.2.2023. [Online]. Available: https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before

[73] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, "Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks," in *16th IEEE Workshop on Offensive Technologies (WOOT'22)*. IEEE, May 2022, https://comsec.ethz.ch/wp-content/files/spring_woot22.pdf.

[74] J. Wikner and K. Razavi, "Retbleed: Arbitrary Speculative Code Execution with Return Instructions," in *USENIX Security*, 2022. [Online]. Available: https://comsec.ethz.ch/wp-content/files/retbleed_sec22.pdf

[75] D. Williams, "LKML: [PATCH v6 02/13] array_index_nospec: sanitize speculative array de-references," 2018, https://lore.kernel.org/lkml/151727414808.33451.1873237130672785331.stgit@dwillia2-desk3.amr.corp.intel.com/.

[76] H. Xia, D. Zhang, W. Liu, I. Haller, B. Sherwin, and D. Chisnall, "A secret-free hypervisor: Rethinking isolation in the age of speculative vulnerabilities," in *IEEE S&P '22*. IEEE, 2022, pp. 370–385.

[77] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[78] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "Bunnyhop: Exploiting the instruction prefetcher," 2023.

[79] J. Zomer and A. Sandulescu, "Linux kernel: Spectre-v1 gadgets," 2023. [Online]. Available: https://github.com/google/security-research/security/advisories/GHSA-m7j5-797w-vmrh