Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions

Sandro Rüegge
ETH Zurich

Johannes Wikner
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Modern branch predictors prevent Spectre v2 attacks by associating predictions with the privilege domain they should be restricted to, or by providing barriers for invalidating predictions when switching contexts. Such branch predictors receive branch resolution and privilege domain feedback asynchronously, but it is unclear whether they always consider the correct order of events. In this paper, we introduce *Branch Predictor Race Conditions (BPRC)*, a class of vulnerabilities where asynchronous branch predictor operations violate hardware-enforced privilege and context separation mechanisms in all recent Intel CPUs. Our analysis reveals three variants, breaching the security boundaries between user and kernel, guest and hypervisor, and across indirect branch predictor barriers. Leveraging BPRC, we introduce *Branch Privilege Injection (BPI)*, a new Spectre v2 primitive that injects arbitrary branch predictions tagged with kernel privilege from user mode. Our end-to-end BPI exploit leaks arbitrary kernel memory from up-to-date Linux systems across six generations of Intel CPUs, at 5.6 KiB/s on Intel Raptor Cove.

1 Introduction

The most reliable countermeasures against Branch Target Injection (BTI) attacks (also known as Spectre v2) [8, 26, 28, 42, 43, 45] are enforced directly by processor microarchitecture or microcode (*i.e.*, hardware-enforced) [7, 18, 34]. These hardware-enforced countermeasures operate on highly asynchronous state inside the processor pipeline, resulting in a fertile breeding ground for race conditions. This paper introduces Branch Predictor Race Conditions (BPRC), a class of branch predictor vulnerabilities resulting from race conditions in recording asynchronous prediction feedback. BPRC variants evade hardware-enforced countermeasures on all recent Intel CPUs which we showcase with an end-to-end exploit that leaks arbitrary kernel memory on Intel Raptor Cove.

Hardware-enforced countermeasures. Hardware-enforced BTI defenses may either restrict or invalidate exploitable branch predictions. Indirect Branch Restricted Speculation (IBRS) is a collective name for BTI defenses that restrict newly learned indirect branch target predictions to the current privilege domain, mitigating cross-privilege BTI attacks. Legacy IBRS relies on slow, microcoded Model Specific Register (MSR) writes on privilege domain transitions, gravely impacting software performance [14]. Later processor generations provide enhanced/Automatic IBRS (eIBRS [18], AutoIBRS [34]) as the more efficient successor that remains always-active after a single MSR write at boot-time, making it the recommended option for operating systems and hypervisors. If the victim runs in the same privilege domain as the attacker, a software-issued Indirect Branch Prediction Barrier (IBPB) can instead invalidate predictions. While eIBRS and AutoIBRS have certain specification-related shortcomings [8, 38, 42, 45], they have so far been successful at enforcing their basic policy that indirect branch predictions learned in less privileged domains must be restricted from being used in more privileged domains.

Branch Predictor Race Conditions (BPRC). To study BPRC, we design experiments to determine *when* branch predictions are inserted in the branch predictor and *what* privilege domain restriction they take. We observe inconsistencies in the privilege domain restriction held by newly inserted indirect branch target predictions. In particular, we show that indirect branch target resolution feedback is managed asynchronously by the microarchitecture and is not synchronized by, for example, dispatch serializing instructions, like `lfence`. In fact, a delay occurs before feedback from a newly executed indirect branch is made available for future predictions of the same branch. Following this observation, we find that privilege domain transitions may occur before new predictions are inserted in the branch predictor and that such predictions may consequently associate with the wrong privilege domain on Intel CPUs. These results surface two concrete variants of BPRC on Intel eIBRS-enabled CPUs, which we call $\text{BPRC}_{U\to K}$ and $\text{BPRC}_{G\to H}$: injecting branch target predictions from user mode into the kernel and from the guest

into the hypervisor, respectively. We further investigate IBPB and find a third variant called BPRC$_{IBPB}$: injecting targets that remain valid after an IBPB. Our analysis also shows that BPRC is not newly introduced. In fact, the vulnerability is present across at least six different generations of Intel processor microarchitectures since Spectre v2 hardware mitigations were first introduced.

Branch Privilege Injection (BPI). To investigate the conditions under which BPRC$_{U\to K}$ becomes exploitable, we design further reverse engineering experiments that show the predictions that are inserted in the Branch Target Buffer (BTB) (as opposed to the Indirect Branch Predictor (IBP) [28]) are the ones that take the incorrect privilege. We use this insight to craft *Branch Privilege Injection (BPI)*, a new BTI primitive that allows an attacker to inject a branch prediction into a more privileged domain with high precision. Demonstrating BPI in a realistic setting, we construct an end-to-end exploit on Intel Raptor Cove that leaks arbitrary memory on Linux at 5.6 KiB/s with 99.8 % accuracy. To mitigate BPRC, we propose two directions: eliminating exploitable branches and preventing harmful indirect branch prediction via model-specific speculation controls. To eliminate exploitable indirect branches and returns, we suggest *Retpoline* [39] combined with disabling of alternate return target prediction in supervisor mode [20]. We measure an overhead of up to 3.1 % and 8.3 % with UnixBench and lmbench, respectively. For preventing harmful indirect branch prediction, we propose deactivating all indirect branch predictions in supervisor mode. While only supported on newer processors, this approach results in a lower overhead of up to 1.7 % in UnixBench and 6.4 % in lmbench. Furthermore, Intel provided us with a microcode patch to mitigate this issue which we also evaluate.

Contributions. In summary, our contributions are:

- We introduce BPRC, a new class of microarchitectural vulnerabilities where asynchronous branch predictor updates race against other operations, like privilege domain transitions and prediction invalidation.
- We demonstrate three concrete variants of BPRC on Intel CPUs with eIBRS protection where we bypass user-to-kernel (BPRC$_{U\to K}$), guest-to-hypervisor (BPRC$_{G\to H}$), and IBPB (BPRC$_{IBPB}$) security boundaries.
- Leveraging BPRC$_{U\to K}$, we introduce BPI, a new BTI primitive that can inject indirect branch target predictions tagged with a privileged domain from an unprivileged process. Our example end-to-end exploit based on BPI can leak hashes from `/etc/shadow` on Intel's recent Raptor Cove microarchitecture within 21 s.
- We evaluate two mitigation directions combining software defenses with model-specific speculation controls for addressing BPRC.

Responsible disclosure. We disclosed BPRC to Intel PSIRT in September 2024 and provided proof-of-concept code upon request. They confirmed that we are the first to report BPRC and assigned it CVE-2024-45332. Intel stated that a microcode update is required for full mitigation of the issue. To provide sufficient time to develop and test the microcode update, our findings were held under embargo until May 13, 2025. We also informed AMD and ARM who agree with our assessment that their processors appear not to be affected by BPRC. Additional information, including the source code for all the experiments, can be found at https://comsec.ethz.ch/bprc.

2 Background

We provide background on branch prediction, branch target injection attacks and relevant mitigations required to better understand this work.

2.1 Indirect branch prediction

In the ubiquitous x86 architecture, indirect branches are control-flow instructions that receive their *branch target* address from either a register or memory operand. The address of the branching instruction itself is the *branch source* address. Throughout program execution, as new branches and branching patterns are encountered, a Branch Prediction Unit (BPU) *learns* about their outcomes. Over time, the BPU gains increasingly refined understanding of the control flow of the program and can provide accurate predictions of future control flows, allowing better processor utilization. Processors employ both static and dynamic predictions for indirect branches. Static predictions are based on the Instruction Pointer (IP) of the branch source (*i.e.*, IP-based), whereas dynamic predictions are additionally based on the path-history preceding the branch source (*i.e.*, path-based) [44]. On Intel, as shown in Figure 1, IP-based predictions are served from the BTB, whereas path-based predictions are served from the IBP [28]. Branch-history is recorded in a Branch History Buffer (BHB). Similarly, AMD processors serve static predictions from a BTB and dynamic predictions from an Indirect Target Array (ITA) [3].

BTB. The Branch Target Buffer (BTB) is a set-associative structure, where the branch source maps to a particular BTB set and tag. On the recent Intel Golden and Raptor Cove microarchitectures, set and tag is derived from the lower 24 bits of the branch source [26, 28]. On AMD and Intel microarchitectures before Golden Cove, also higher bits are used to derive a BTB index [12, 45]. Because the majority of branch targets are near their respective branch sources, storing the full branch target address is redundant with 48-bit (even 57-bit on servers) virtual address spaces. In fact, the x86-64 ISA specifies at most a signed 32-bit displacement operand for

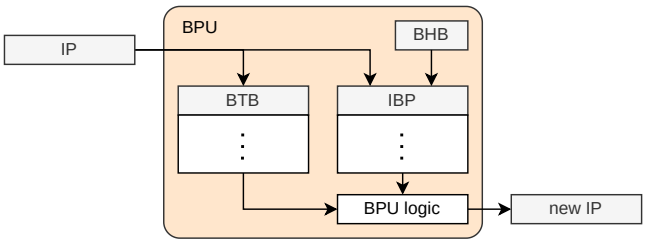

Figure 1: Branch prediction involving an IP-based (BTB) predictor and a path-based (IBP) predictor using branch history (BHB).

direct branch instructions. Hence, BTBs typically only store the lower portion of the branch target [48]. Besides the target, the BTB also stores other information, including *branch type* information [45]. Based on this type, the branch predictor decides whether an IBP-lookup should be performed for a dynamic prediction [28].

IBP. Unlike the BTB, the Indirect Branch Predictor (IBP) manages predictions for indirect branches with *multiple* targets. To predict the correct target, IBP uses branch history [28]. Intel CPUs records this branch history in the Branch History Buffer (BHB), and it is likely that AMD processors use a similar structure for their ITA-based predictions. The BHB is a shift-register that records a footprint of recently taken branches. This footprint consists of a few bits of information from each recorded branch source and branch target address. The IBP is an ITTAGE-like predictor, where predictions may correlate with an increasingly larger BHB footprint [37]. On Intel Golden Cove and Raptor Cove microarchitectures, a prediction may correlate with the last 34, 66 and 194 branches [28]. A hit in the IBP takes precedence over the BTB and provides a full 48-bit target address.

RSBA. Return target predictions are provided from a dedicated Return Stack Buffer (RSB). The RSB has a limited size and after sufficient return instructions it becomes empty. This condition is called *RSB underflow*, at which point return instructions may be predicted like indirect branches [43]. The Intel-specific behavior is known as Return Stack Buffer Alternate (RSBA). Although such behavior is not present on AMD processors [44], it is unclear whether ARM microarchitectures feature it. While new RSB predictions are added on call instructions, BTB and IBP predictions for return instructions need to be added on mispredicted return instructions.

2.2 BTI attacks

Branch Target Injection (BTI) is a class of transient execution attacks that aims to control the predicted target address of an indirect branch to leak information [26]. The attacker diverts the speculative control flow by controlling the predicted target from a *speculation gadget*, pointing it to a *disclosure gadget* that accesses secrets in the victim domain and transmits them to the attacker via a side channel. The speculation gadget, residing in the victim domain, contains a *victim branch* that the attacker influences through a *BTI primitive*. If the victim domain is different from the attacker's, the BTI procedure involves a *training branch* that resides in the domain of the attacker at an address that maps to the same BTB set and tag as the victim branch, imposing aliasing [13]. Moreover, BTI primitives sometimes involve priming the BHB into the same state as it will be in upon execution of the victim branch. Advanced BTI primitives include exploitation of Branch Type Confusion (BTC) [2,43], nested *phantom speculation* [38,45], and use of speculative *dispatch gadgets* [42].

2.3 Spectre defenses

Since the dawn of BTI attacks, a wide variety of hardware and software defenses have been introduced. We divide hardware defenses into three categories: (1) *restriction* of branch predictions to a certain privilege domain; (2) *sanitization* of parts of the branch prediction state; and (3) *supplementary speculation controls* to limit, disable, or reconfigure specific branch prediction features. If hardware defenses are not applicable or otherwise unfeasible, software defenses are employed.

Restriction. Indirect Branch Restricted Speculation (IBRS) is a BTI mitigation that restricts indirect branch target predictions to certain privilege domains. Branch target predictions learned in less privileged domains (*e.g.*, user, guest kernel) may not be used in more privileged domains (*e.g.*, supervisor, hypervisor). The original, now-legacy, IBRS mitigation requires software to write to a model-specific speculation control register on every privilege transition [24]. Modern Intel, AMD and ARM processors offer enhanced IBRS (eIBRS) [18], AutoIBRS [34], and CSV2 [7], respectively, to restrict predictions transparently, without such Model-Specific Register (MSR) writes. For eIBRS/AutoIBRS/CSV2 to restrict predictions correctly, knowledge of the current privilege domain is necessary — both when forwarding predictions and when learning about new ones. If branch predictors are updated asynchronously to the instruction stream, it is paramount that they track the privilege domain that these updates originate from.

Sanitization. Sanitization of branch prediction via Indirect Branch Prediction Barrier (IBPB) [1, 17] is recommended in scenarios where restrictions do not suffice, such as when distrusting contexts execute under the same privilege domain. Such contexts include distrusting user programs and distrusting guest virtual machines. IBPB invalidates all indirect branch target predictions in the BTB and IBP such that no instruction after the barrier will use predictions learned by instructions before the barrier. Because of its high performance penalty, IBPB is often used only as a last resort [28, 38, 44].

Supplementary speculation controls. In many cases, researchers have demonstrated attacks that exploit overlooked properties of branch prediction restriction [8,42,43] and isolation [38]. To mitigate many of these, vendors expose new speculation controls to tweak specific microarchitectural behaviors. For example, to mitigate Branch History Injection (BHI) attacks [8, 42], Intel introduced a new speculation control `BHI_DIS_S`. When toggled on, the mitigation guarantees that privileged domains will not use less privileged history for indirect branch predictions in supervisor mode [20]. Complementary speculation controls include `RRSBA_DIS_S` to disable RSBA speculation in supervisor mode and `IPRED_DIS_S` to disable indirect branch prediction in supervisor mode altogether.

Other vendors have introduced similar controls that limit, disable, or reconfigure microarchitectural features in response to advanced BTI attacks (*e.g.*, AMD [2,5]). Exactly what speculation controls, such as `BHI_DIS_S`, do at a microarchitectural level is undisclosed. To assess their security properties, we must carry out our own reverse engineering.

Software defenses. Software defenses provide an alternative if hardware mitigations are unavailable [2, 39] or too costly [14]. Speculation can be inhibited using the `lfence` instruction thanks to its instruction-stream serializing semantics, effectively forcing ongoing operations to complete before dispatching any following operations for execute (*i.e.*, *dispatch serializing*) [4, 19]. Predictors can also be sanitized with software constructs, for example RSB stuffing [14, 21, 22] and *untrain* procedures [2, 33]. Susceptible branches can be reduced or eliminated [2, 6, 16, 33, 39]. *Retpoline* [39] has been particularly effective against BTI attacks by eliminating indirect branches but incurs a performance overhead due to forced mispredictions. Additionally, defense in depth strategies like Supervisor-Mode Execution Prevention (SMEP) and Supervisor-Mode Access Prevention (SMAP) can be employed to restrict supervisor interaction with user pages [22].

3 Threat Model

We assume a local attacker with unprivileged code execution capabilities on a machine running an up-to-date Linux kernel on top of Ubuntu 24.04 with all the latest security updates and mitigations against CPU vulnerabilities (version 6.8.0-47-generic at the time of this writing). The aim of the attacker is to leak arbitrary privileged memory by hijacking the speculative control flow despite deployed hardware mitigations such as eIBRS, AutoIBRS and CSV2 on Intel, AMD and ARM CPUs. Similar to recent cross-privilege transient execution attacks [8, 38, 43], we assume that the attacker analyzes the target kernel in a preliminary offline stage to find suitable gadgets for their attack.

4 Overview

If supported by the processor, operating systems enable eIBRS or AutoIBRS to mitigate cross-privilege BTI attacks [19,34]. These mitigations need to keep track of the privilege domain of branch instructions to work correctly, which is non-trivial due to the highly complex and asynchronous nature of branch prediction. For example, previous work has shown that branch predictions are updated before branches retire, and in certain cases even before they are decoded [38]. Our first challenge revolves around analyzing the behavior of restricted branch prediction under race conditions.

> **Challenge (C1).** Exploring the behavior of branch predictors under race conditions.

In Section 5, we demonstrate that while branch prediction entries are sometimes created early [38], they are in many cases inserted in the branch predictor *after* the branches have retired. Upon further analysis, it turns out that the branch predictor also operates asynchronously despite synchronizing instructions. As such, branch predictions are added under asynchronous conditions, relying on state of past branch instructions, such as the privilege under which these branch instructions were executed. To investigate whether this state is correctly reflected under restricted branch speculation, we then trigger updates of branch predictions, specifically after privilege domain transitions. We find that when branch training involves a privilege switch, the injected target is occasionally associated with the wrong privilege domain in Intel CPUs. We refer to the class of effects caused by the asynchronous operation of the BPU as *Branch Predictor Race Conditions (BPRC)*. Our variant analysis of BPRC demonstrates three variants which bypass the user-to-kernel ($\text{BPRC}_{U\to K}$), guest-to-hypervisor ($\text{BPRC}_{G\to H}$) and IBPB (BPRC_{IBPB}) security boundaries, respectively.

$\text{BPRC}_{U\to K}$ and $\text{BPRC}_{G\to H}$ suggest that eIBRS is not capable of enforcing its specified security guarantees. Our next challenge involves understanding the exact conditions under which we can control the privilege of a branch prediction entry for creating an exploitation primitive.

> **Challenge (C2).** Understanding the conditions under which the observed security violations of eIBRS become exploitable.

In Section 6, we provide a new technique to discern BTB- and IBP-based branch predictions from one-another, eventually enabling us to pinpoint the BTB as the culprit of $\text{BPRC}_{U\to K}$. This allows us to create a new primitive, called *Branch Privilege Injection (BPI)*, which provides control over the associated privilege of indirect branch predictions. BPI resurrects the entire class of cross-privilege BTI attacks that were deemed dead since the inception of eIBRS.

Table 1: Evaluated Microarchitectures

Vendor	CPU	Year	Codename	Code	Microarch.	Microcode
Intel	Core i7-14700K	2023	Raptor Lake Refresh	RPL-R	Raptor Cove, Gracemont	0x129
	Xeon Silver 4510	2023	Sapphire Rapids	SPR	Golden Cove	0x2b000590
	Core i7-13700K	2022	Raptor Lake	RPL	Raptor Cove, Gracemont	0x129
	Core i7-12700K	2021	Alder Lake	ADL	Golden Cove, Gracemont	0x037
	Core i7-11700K	2021	Rocket Lake	RKL	Cypress Cove	0x59
	Core i7-10700K	2019	Comet Lake	CML	Skylake	0xfc
	Core i9-9900K	2018	Coffee Lake Refresh	CFL-R	Skylake	0x100
AMD	Ryzen 9 9900X	2024	Granite Ridge	–	Zen 5	0xb40401c
	Ryzen 7 7700X	2022	Raphael	–	Zen 4	0xa601203
ARM	Google Tensor	2021	Whitechapel	–	Cortex-X1	–
	Google Tensor	2021	Whitechapel	–	Cortex-A76	–

To demonstrate exploitation using BPI, we employ it against an indirect branch in kernel as an unprivileged attacker. However, our BPI primitive is ineffective if the victim branch is not served from the BTB. Hence, our next challenge is to extend our BPI primitive with an IBP-eviction primitive, to enable the repeated use of the vulnerable BTB.

Challenge (C3). Making our BPI primitive repeatable in the presence of current mitigations.

In Section 7, we investigate how MSR-provided speculation controls interact with BPI and find that Intel's new on-by-default BHI speculation control, `BHI_DIS_S`, *improves* the reliability of BPI. We find that this is because `BHI_DIS_S` disables the BHB-based IBP in the kernel entirely. We further demonstrate the impact of BPI by building an end-to-end exploit that leaks arbitrary memory from a recent Linux kernel despite all state-of-the-art defenses against transient execution attacks. In summary, we demonstrate how an ineffective hardware defense mechanism has devastating impact on the security of modern computer systems.

5 Branch Predictor Race Conditions

In this section, we study the behavior of indirect branches in relation to privilege domain transitions on a variety of microarchitectures with restricted speculation defenses. We first try to understand the asynchronous nature of branch predictor updates (Section 5.1) before experimenting with privilege transitions during asynchronous branch prediction updates (Section 5.2). In Section 5.3 we then present two additional variants of branch predictor race conditions.

Methodology. Table 1 lists all microarchitectures evaluated in this work. The recent Intel desktop processors feature two microarchitectures in a single CPU, one optimized for processing power (Golden Cove, Raptor Cove) and one providing low energy consumption (Gracemont). In our microarchitectural experiments we employ a kernel module to run arbitrary user-mode code with supervisor privileges. We further disable SMEP and SMAP which allows us to run the exact same code (history, branches) in different privilege domain. Experiments using the caches as side channel provide the branch target of tested control flow instructions from an uncached memory pointer. This lengthens the speculation window, resulting in clearer results. To test returns, we use an RSB underflow snippet as shown in Listing 1 to force RSBA predictions. The final attack in Section 7 does not require any modifications to the kernel and runs with all the default mitigations enabled.

```
1 %rep 32
2   lea    r8, qword ptr [rip + 0x3]
3   push   r8
4   ret
5 %endrep
```

Listing 1: Assembly to trigger an RSB underflow condition by repeatedly executing returns without a matching call.

5.1 Asynchronous branch predictor updates

There are two aspects that are interesting when considering branch prediction updates: (i) when the branch prediction is first *created* and (ii) when it is *inserted* in the branch predictor and consequently made available to future branch instructions. Previous work shows that branch predictions can be created *before* branch instructions retire, even before instruction decode is done [38]. Creating branch predictions early is particularly beneficial for complex branch predictors with long update latency, since the refined prediction can be used sooner. The second aspect, however, has not been previously studied.

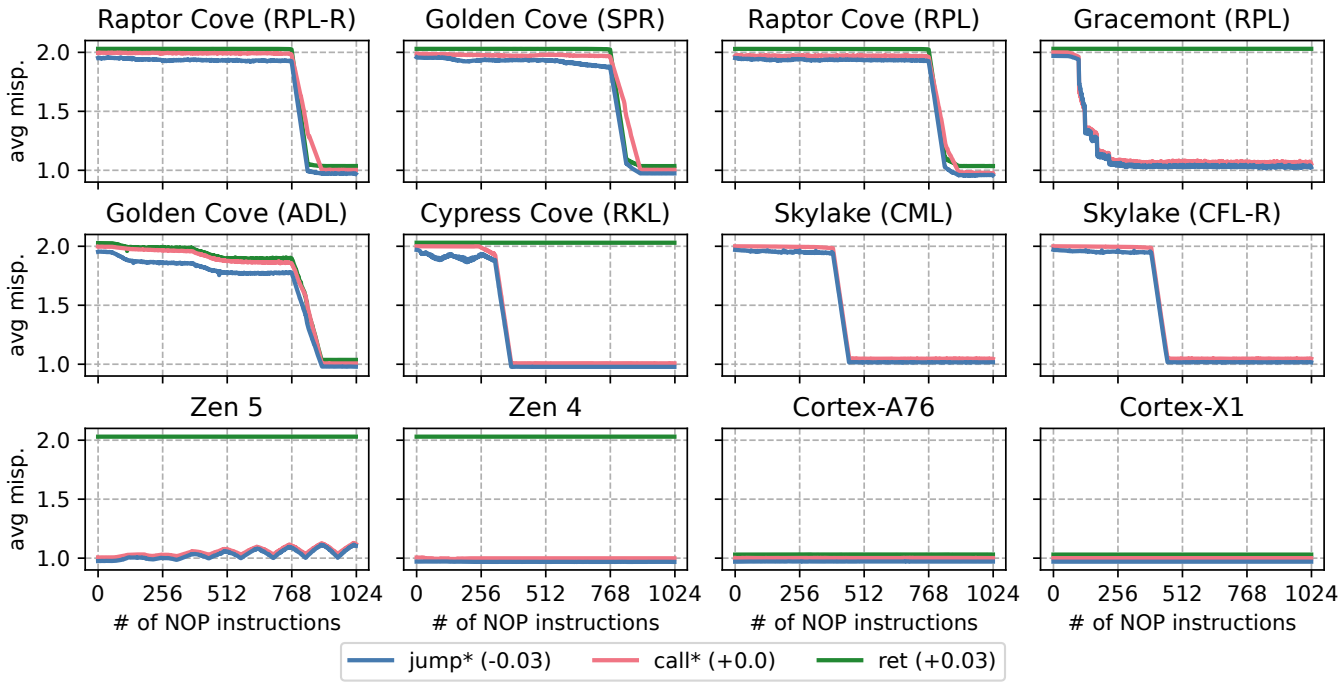

Figure 2: Average number of mispredictions (N=100 K) when executing the same branch twice with d `NOP`s of delay. The data points are shifted according to their label to improve visibility.

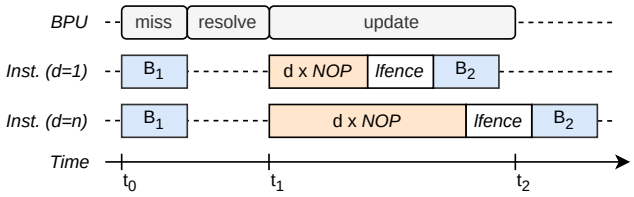

Figure 3: Branch prediction update timeline for the same branching instruction executed twice (B_1, B_2) with short ($d = 1$) to long ($d = n$) delay.

In particular, if a branch misprediction occurs, how long does it take before the correct prediction can be used for future occurrences of the same branch? Indeed, hardware designs of branch predictors report multiple cycles of update latency for new branch predictions [9].

As shown in Figure 3, we aim to detect the latency between the resolution of a branch instruction and the use of its prediction feedback in subsequent predictions. We execute the *same* branch instruction twice (B_1, B_2) with a variable delay d in-between, composed of `NOP` instructions and a dispatch-serializing `lfence`. By randomizing the branch source and target, the first execution of the branch (B_1) is set to mispredict. The execution is stalled until t_1, where the target of B_1 is resolved, resulting in the BPU adding a new prediction which becomes available at time t_2. The second execution of the branch (B_2) is delayed by d `NOP` instructions. If d is large enough (*i.e.* B_2 occurs after t_2), B_2 will be correctly predicted with the prediction learned from B_1.

We repeat the experiment 100 K times for each $d \in \{1,2,3,\ldots,1023\}$, for indirect jump (`jmp*`), indirect call (`call*`) and RSBA-predicted return (`ret`). We measure the average number of mispredictions by sampling performance counters for the respective instruction. We use *br_misp_retired.all_branches*, *br_misp_retired.indirect* and *br_misp_retired.ret* on Intel, *ex_ret_brn_ind_misp* and *ex_ret_near_ret_misp* on AMD, and *br_mis_pred_retired* on ARM. As an optional preliminary step, we warm up the instruction cache by executing the experiment code. While this improves the results on most Intel microarchitectures, we omit this step on Gracemont, ARM, and AMD microarchitectures, where this step negatively impacts the results.

In Figure 2, we plot the delay (d) against the average number of mispredictions for both branches (B_1 and B_2). We expect all modern processors to be able to predict B_2 given B_1, which would appear as an average of 1 misprediction in the graph as B_1 cannot be predicted correctly. However, without adding any delay between the two branches (*i.e.*, $d = 0$), we observe up to an average of 2 mispredictions from B_2 on Intel. With increasing delay, the number of mispredictions reduces drastically, suggesting that an update latency of branch target predictions exists. Returns are omitted in the results on Comet Lake and Coffee Lake Refresh, as they do not support *br_misp_retired.ret*. Gracemont, Cypress Cove, Zen 4 and

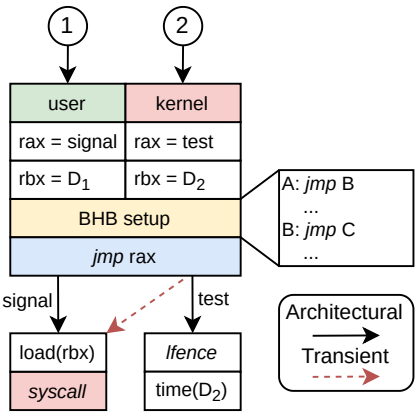

Figure 4: (1) Training of an indirect branch to a cache-signaling gadget, followed by a privilege domain change. (2) Executing the same branch again, but with a different target. D_1 and D_2 are distinct memory addresses.

Zen 5 have no RSBA predictor, resulting in 2 mispredictions on returns.

We emphasize that B_1 has retired before B_2 occurs, and that predictor updates delay until far after dispatch serializing instructions complete, by using d times `lfence` instructions instead of `NOP`s. This reduces the d, but still shows delayed updates. The same is true for other instructions that are known to flush the execution pipeline, like `cpuid`, or instructions that take hundreds of cycles like `pause` instructions. This means that B_1 is fully completed before B_2 executes and suggests branch prediction update latencies of hundreds of cycles. We call this Intel feature Asynchronous Branch Predictor Updates (ABPU).

Observation (O1). Branch predictions on Intel are inserted after respective branch instructions retire and are independent of dispatch serializing instructions.

Negative Results. On the evaluated AMD and ARM processors, we do not find observable ABPU. While Zen 5 does experience a distinct pattern of mispredictions, emerging with an increasing number of `NOP`s, there are no mispredictions for small d. For Zen 4, we need to take care of additional alignment to ensure correct BTB aliasing. In particular, we need to take the target and type of the preceding branch into account in addition to the source of a branch. Hence, we narrow down our scope to Intel parts, which is where we observe ABPU.

5.2 Restricted speculation analysis

As we have seen on Intel processors, branch predictor updates may occur after branch instructions have completed, and that these updates are unordered with respect to dispatch serializing instructions. This observation begs the question whether new predictions staged for insertion maintain all their respective state, or if supplementary state is gathered as the prediction is inserted. In particular, restricted speculation schemes (*i.e.*, eIBRS, AutoIBRS and CSV2) depend on the privilege domain that predictions should be restricted to. Hence, if the privilege domain at branch instruction *retirement time* does not match the privilege domain at prediction *insertion time*, and privilege domain state is gathered at insertion time, a possibility of inserting predictions with incorrect privilege domain emerges. In particular, since dispatch serializing instructions do not wait for predictor updates to complete, we hypothesize that such is not the case for privilege-domain changing instructions (e.g., `syscall`) either.

Table 2: Percentage of jump locations where a gadget hit was observed despite eIBRS.

Microarchitecture	`jump*`	`call*`	`ret`	`noise`
Raptor Cove (RPL-R)	81.1%	79.5%	90.7%	0.0%
Gracemont (RPL-R)	5.2%	7.5%	0.0%	0.0%
Golden Cove (SPR)	99.6%	98.0%	99.1%	0.0%
Raptor Cove (RPL)	79.9%	78.7%	87.6%	0.0%
Gracemont (RPL)	5.0%	9.0%	0.0%	0.0%
Golden Cove (ADL)	61.6%	63.2%	87.7%	0.0%
Gracemont (ADL)	27.8%	1.1%	[a]0.2%	0.2%
Cypress Cove (RKL)	3.1%	0.5%	0.0%	0.0%
Skylake (CML)	3.1%	4.0%	0.0%	0.0%
Skylake (CFL-R)	2.7%	3.3%	0.0%	0.0%

[a] indistinguishable from noise

Investigating this hypothesis, we propose an experiment where the same branch instruction is executed under different privilege domains, as illustrated in Figure 4. In (1), we execute a branch in user mode with a cache-signaling gadget as target, thereby training the branch predictor. This gadget emits an observable register-dependent cache signal if this target ever executes again transiently. After the branch, we trigger a privilege-domain transition to supervisor mode using the `syscall` instruction. From supervisor mode, the branch is executed again (2) but now with a different target that checks for the cache signal. Thanks to disabled SMEP, we can run the exact same instructions as during training. We use a BHB setup sequence to provide a matching history for both executions of the branch such that the BTB or IBP can provide the target. We run the experiment with the branch configured as an indirect jump (`jmp*`), indirect call (`call*`), and RSBA-predicted return (`ret`) [18]. For each configuration, we run the experiment 100 K times, each time randomizing the locations of the branch and the cache-signaling gadget. The primitive is repeated 8 times in succession during each run to warm up all involved microarchitectural components.

The results are presented in Table 2, where for each microarchitecture and tested branch instruction, we state the

percentage of runs with at least one cache hit from executing the branch in supervisor mode. We also report the baseline noise of our cache gadget as the number of hits on a memory location that was not the one accessed during speculation. On all Intel parts supporting eIBRS, we observe a cache signal above the noise for at least two of the three branch instructions. Because training in user mode should not result in a cache signal from executing the branch again in supervisor mode, these observed mispredictions are a clear *violation* of the eIBRS security guarantees.

Observation (O2). Intel eIBRS fails to correctly restrict branch target predictions originating right before a privilege domain change.

The observed security violations constitute an instance of the novel class of vulnerabilities which we term *Branch Predictor Race Conditions (BPRC)*.

5.3 BPRC variants

We refer to instances of BPRC by the boundary which they are violating, *i.e.* $\text{BPRC}_{U\to K}$ for the user-to-kernel violation from Section 5.2. We proceed to analyze additional instances of BPRC that violate security boundaries. In particular, we are considering the guest-to-hypervisor ($\text{BPRC}_{G\to H}$) and IBPB (BPRC_{IBPB}) boundaries in this section. We provide a summary of results here while leaving detailed results in Section 8.

$\text{BPRC}_{G\to H}$. To test for $\text{BPRC}_{G\to H}$, we port the experiment from Section 5.2 to the Linux KVM selftest framework where we have a lot of control over guest and host memory placement. The $\text{BPRC}_{G\to H}$ boundary, called VMExit, can occur explicitly due to guest instructions like `vmcall` or implicitly, for example due to hypervisor-intercepted instructions like `cpuid`, or guest physical page faults.

We replace the `syscall` with a VMExit using either a `vmcall` instruction, a `cpuid` instruction or by having the branch target guest physical page unmapped. By again looking for cache gadget hits, we find that $\text{BPRC}_{G\to H}$ affects all evaluated Intel microarchitectures, except for Gracemont.

BPRC_{IBPB}. We evaluate BPRC_{IBPB} again in a way similar to Section 5.2 by replacing `syscall` with an IBPB MSR write. As this MSR write is restricted to supervisor mode, the experiment has to run in the kernel using a kernel module. It turns out that even the IBPB, which is an explicit barrier, is not always synchronized with the BPU updates. While IBPB appears to be synchronized on the Gracemont, we can cause a race condition on all other Intel microarchitectures. Thus, IBPB, which is an expensive last resort mitigation, fails to protect against BPRC.

Observation (O3). In addition to $\text{BPRC}_{U\to K}$, Intel is also affected by at least $\text{BPRC}_{G\to H}$ and BPRC_{IBPB}.

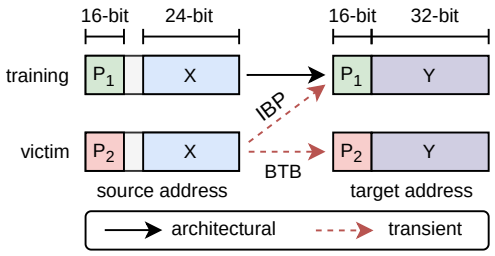

Figure 5: Discerning IBP from BTB predictions. With distinct upper 16 bits of the training and victim branch sources, the two predictors produce distinct predictions from the victim branch source.

Next, we find the conditions under which $\text{BPRC}_{U\to K}$ becomes exploitable. We leave the exploitation of $\text{BPRC}_{G\to H}$ and BPRC_{IBPB} to future work.

6 Branch Privilege Injection

In Section 5, we found a security violation in eIBRS related to privilege transitions. In this section, we construct a new primitive, called *Branch Privilege Injection (BPI)*, for exploiting $\text{BPRC}_{U\to K}$.

6.1 Discerning between predictors

To accurately inject privileged predictions, we need to understand which predictors are providing branch targets when they should not. Differences in the BTB and IBP can be exploited to trace back a prediction to its source predictor. The BTB only provides a partial target address, whereas the IBP provides a full target address. Figure 5 illustrates how we exploit this property to discern between BTB and IBP predictions on Intel CPUs:

1. The training source and training target addresses share the same upper 16 bits (P_1), thereby enabling creation of both a BTB entry and an IBP entry.

2. Assigning the victim branch source an address matching the lower 24 bits of the training branch source (X) causes aliasing in the BTB, and assuming same preceding branch histories, also in the IBP [28].

3. With non-matching upper 16 bits for the victim branch source (P_2) and training branch source (P_1), a BTB-based prediction at the victim branch produces a branch target different from the IBP-based target.

For the IBP, the target becomes the concatenation $P_1 \parallel Y$, and for the BTB, the target becomes $P_2 \parallel Y$. With this procedure, we can detect whether a mispredicted target used the BTB or the IBP by assigning the two possible branch targets ($P_1 \parallel Y$ and $P_2 \parallel Y$) with different cache-signaling gadgets. On Intel

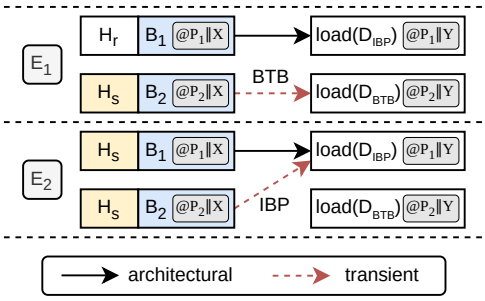

Figure 6: Two experiments to detect single-target branch insertion policy for the BTB and IBP on Intel CPUs. H_r and H_s are two different branch histories.

processors pre Golden Cove, BTB addressing can involve additional bits [12] but the approach still applies.

We run the two experiments (E_1 and E_2), illustrated in Figure 6, to determine when a prediction is inserted in the BTB and IBP. In both experiments, we first train a single-target indirect branch B_1 and then execute an aliasing indirect branch B_2. Using the technique introduced in this section, we can detect which predictor provided the target prediction for B_2. In E_1, we run B_1 and B_2 with different branch histories H_r and H_s. This results in a miss in the IBP and causes a fallback to the BTB. In E_2, B_1 and B_2 both use H_s which results in a prediction from the IBP. This means that both BTB and IBP get updated when there is a branch misprediction, even for single-target branches unlike what has been reported in [28].

Observation (O4). Upon misprediction, a new prediction is inserted into the Intel BTB *and* the IBP.

Based on this observation and the ability to discern the originating predictor, we can determine the culprit for BPRC. Hence, we repeat the experiments from Section 5.2 with different cache-signaling gadgets for BTB and IBP prediction. We find that the observed cache signals originate from BTB-based mispredictions.

Observation (O5). All observed eIBRS security violations originate from the BTB.

Following this observation, the next section focuses exclusively on the Intel BTB.

6.2 BPI breaking points

The further improve our understanding of BPI, we attempt to find breaking points in $\text{BPRC}_{U\to K}$ by inserting delays either during or after training the branch predictor. First, we want to test the hypothesis that there is a temporal proximity dependency between the privilege transition of the `syscall` and training branch, analogous to the delay derived in Section 5.1.

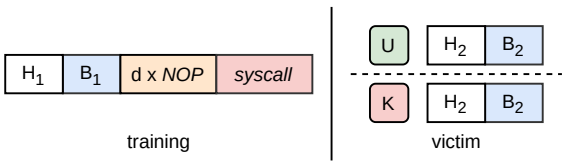

Figure 7: Experimental setup to determine in which privilege domain a branch target is available for a given delay d.

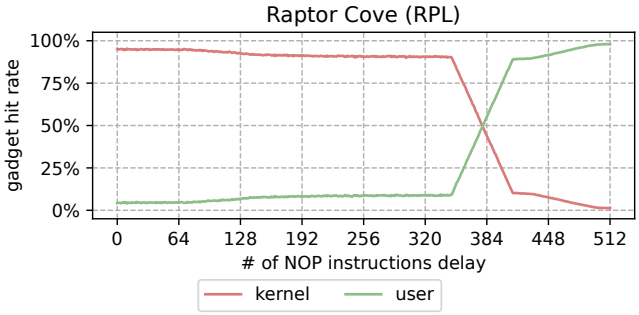

Figure 8: Evaluation of what privilege domain restriction a newly trained `jmp*` prediction gets associated with in relation to the delay of a following `syscall`. The `syscall` forms a key-component of the BPI primitive and needs to happen within a time-window given by 300–400 `NOP`s.

Second, while we hypothesize that BPI results in incorrect privilege domain associated with a BTB prediction, we want to ensure that the violation is not of transient nature and not only exploitable during a short time window.

We proceed with an in-depth analysis of the experiment from Section 5.2 that resulted in BPRC (Figure 4). We modify the experiment to match Figure 7, where training and victim branch executions are separated into two independent procedures. The training (B_1) and victim (B_2) branches now reside at two different addresses that alias in the BTB, with B_1 and B_2 in the address ranges of user mode and kernel mode, respectively. To avoid IBP hits, we randomize two different branch histories (H_1, H_2) before B_1 and B_2. To evaluate the timing dependency of the training procedure, between B_1 and the `syscall`, we use a controllable delay of d `NOP`s between them. For each $d \in \{0,1,2,\ldots,512\}$, we run the training procedure and then test the prediction of B_2 under user (U) or kernel (K) mode. After training, we always return to user mode before executing B_2. This experiment allows us to detect whether the prediction learned from B_1 gets restricted to user mode or kernel mode for a given d. To avoid cache misses and other factors that may impair the experiment, as a preliminary warm up step, we first run the experiment without taking any measurements.

We show the results for `jmp*` on Raptor Cove in Figure 8, but a similar effect emerges on all susceptible microarchitectures. The results expose a correlation between the delay of

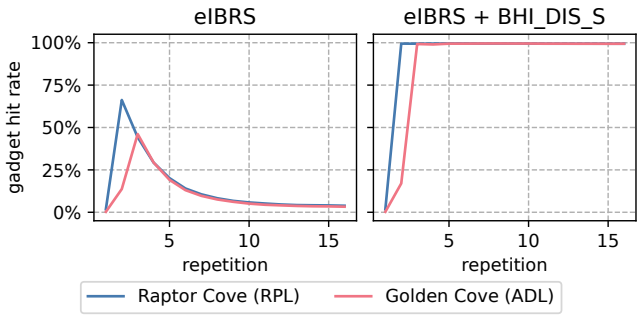

Figure 9: Leak primitive success rate over successive repetitions for `jmp*` instructions with and without the `BHI_DIS_S` mitigation.

the training `syscall` instruction and the privilege domain a prediction gets associated with. As such, the `syscall` following B_1 is a key-component of our BPI primitive, dependent on the temporal proximity between the two. This does not pose an issue for attackers since this delay is under their control.

Once the corrupted prediction is learned, no timing dependence between B_1 and B_2 exists. The prediction is persisted in the BTB until it is overwritten or evicted.

Observation (O6). The eIBRS violation is independent of the temporal proximity between the training and victim branches.

Evidently, we are observing inconsistent privilege domain restrictions associated with branch predictions that occur in temporal proximity to privilege transitions. These predictions are persisted in the BTB indefinitely, until they are evicted or overwritten. Hence, we have acquired an in-depth understanding of the constraints associated with BPI, which is a first-of-its-kind primitive to exploit $\text{BPRC}_{U\to K}$.

7 Practical Exploitation with BPI

To demonstrate the impact of the BPI primitive, introduced in Section 6, we build an arbitrary kernel memory leak exploit. The default Ubuntu 24.04 kernel version at the time of writing is 6.8.0-47-generic, and we leave all the default security mitigations enabled. To leak arbitrary memory from the kernel, we overcome the following four challenges.

1. Achieve a repeatable BPI primitive (Section 7.1).
2. Inject a kernel branch target (Section 7.2).
3. Find speculation and disclosure gadgets (Section 7.3).
4. Break KASLR (Section 7.4).

Similar types of user-to-kernel exploits have been described in prior work [8, 38, 42, 43].

Table 3: IBP misprediction percentage for the eighth execution of kernel branches with and without `BHI_DIS_S`. Gracemont does not support (RSBA) `ret`. Green signifies few mispredictions and red signifies only mispredictions.

Microarch.	No Mitigation			BHI_DIS_S		
	*jump**	*call**	*ret*	*jump**	*call**	*ret*
Raptor Cove (RPL-R)	0%	1%	1%	100%	100%	100%
Golden Cove (SPR)	0%	2%	1%	100%	100%	100%
Raptor Cove (RPL)	0%	0%	1%	100%	100%	100%
Gracemont (RPL)	1%	2%	-	100%	100%	-
Golden Cove (ADL)	0%	0%	1%	100%	100%	100%

7.1 Repeatable BPI

We need a repeatable and reliable injection primitive to achieve high bandwidth. During the analysis of $\text{BPRC}_{U\to K}$ in Section 6.2, we assumed full control of branch history at the victim branch. However, a speculation gadget in the kernel will have multiple branches preceding it that we do not control. In Figure 9-left, we show that with consistent branch history, the success rate of BPI decreases over successive repetitions. To understand why, we recall that the BTB only provides branch target predictions for indirect branches if there is no hit in the IBP. If the BTB predicts an indirect branch, and there is an IBP hit, the IBP-based prediction takes precedence [28], regardless of the outcome of the BPI.

Since kernel branch history can be manipulated and exploited from user mode [8], the newly introduced on-by-default `BHI_DIS_S` speculation control prevents user code from controlling path-based branch target predictions in kernel mode [20]. To avoid IBP hits in kernel mode, we reverse engineer branch prediction behavior when `BHI_DIS_S` is set. We execute the same branch with the same branch history eight times in kernel mode and measure the misprediction rate for the last execution. By placing the branch target > 4 GiB from the branch source, we ensure only the IBP can provide a prediction. Table 3 shows the misprediction percentage for `jmp*`, `call*` and (RSBA) `ret` with and without `BHI_DIS_S` set. Surprisingly, all branches mispredict with `BHI_DIS_S` set, suggesting it disables the IBP in kernel mode entirely, rather than only isolating or invalidating the BHB.

Observation (O7). `BHI_DIS_S` on Intel Golden Cove, Raptor Cove and Gracemont cores *disables* path-based indirect branch prediction (IBP) in kernel mode.

Figure 9-right presents the substantially improved repeatability of BPI provided by `BHI_DIS_S`. An unprivileged attacker cannot enable the mitigation themselves but Linux enables it by default to prevent BHI attacks. This serves as a reminder of how a mitigation against one attack can inadvertently improve the effectiveness of another.

7.2 Injecting a kernel branch target

Because of SMEP [23] (enabled by default), our cross-privilege BTI exploit must inject a kernel target rather than a user target into the BTB. Linux splits the address space between user and kernel using the highest address bit, making it non-trivial for an unprivileged attacker to inject a full kernel target into the branch predictor by branching to it. However, the BTB provides partial target addresses [28], so the attacker only needs to branch to an address where the lower portion matches the desired kernel target. The upper bits of the BTB target are provided by the victim branch source, which will be in the kernel address range. The technique follows the one we used in Section 6.1 and Figure 5.

7.3 Gadgets

BTI attacks use a *speculation gadget* with a victim branch that mis-speculates to a *disclosure gadget* to transmit secrets [43]. As BPI works for all types of indirect branches and RSBA-predicted returns, the speculation gadget may use any of these as victim branch. To transmit secrets with Flush+Reload [46], the disclosure gadget accesses a *reload buffer* with a secret-dependent offset. A reload buffer [40] is some arbitrary memory that is shared between the attacker and victim, acting as a side-channel transmission medium. In this case the victim (kernel) accesses the reload buffer through physmap which maps the attacker's user space reload buffer in kernel space.

Speculation gadget. The speculation gadget needs attacker-controllable secret and reload buffer pointers at the victim branch source. We can provide these from user mode as system call arguments. Many system call handlers accept arbitrary 64-bit values as arguments (although the system call might return an error code). Because returns are only vulnerable under RSB underflow, we look for speculation gadgets containing an indirect branch, where at least two registers are controllable via system call arguments.

```
static long __keyctl_read_key(struct key *key,
    char *buffer, size_t buflen)
{
  long ret;

  down_read(&key->sem);
  ret = key_validate(key);
  if (ret == 0)
    ret = key->type->read(key, buffer, buflen);
  up_read(&key->sem);
  return ret;
}
```

Listing 2: Linux snippet of *keyctl* calling *read*. The caller function gets inlined, resulting in registers *r13* (buffer) and *r12* (buflen) under attacker control.

To find such speculation gadgets, we use a regular expression matching function-pointer calls (typically compiled to indirect calls) in the source code. We then check if these calls are reachable from system call handlers. Such is the case for the *keyctl* system call, which leads to a function-pointer call (victim branch) to the function *read*, shown in Listing 2. Moreover, the *buffer* and *buflen* arguments at the victim branch are fully controllable via system call arguments and have been assigned to *r12* and *r13*. The chain of pointer dereferences, necessary for the function-pointer call, is likely to produce a long transient execution window of the disclosure gadget.

```
HUF_compress1X_usingCTable_internal_default:
;...
movzx  edx, byte ptr [r12]
mov    rbx, qword ptr [r13 + rdx*8]
;...
```

Listing 3: Disclosure gadget found in the Linux kernel in `HUF_compress1X_usingCTable_internal_default`.

Disclosure gadget. The disclosure gadget needs to use the two attacker-controlled registers to leak and transmit the secret via Flush+Reload. For Flush+Reload, we need to consider hardware prefetching, which may transparently fetch cache lines before we can measure their access times, inhibiting the side channel. An ideal disclosure gadget avoids prefetchers by transmitting the secret by multiplying it by 4096 before using it as the reload buffer offset, as most of prefetchers do not operate across page boundaries [35]. While we were unable to find an ideal gadget, Wikner and Razavi demonstrated that the secret can instead be deduced by offsetting the reload buffer pointer until the secret-dependent reload-buffer access lands in a consecutive page [43]. Considering this technique, we search for a disclosure gadget that dereferences one byte of the first attacker-controlled register (secret pointer), offsets a second attacker-controlled register (reload buffer) with this byte, and dereferences the resulting address. A suitable disclosure gadget is located in the Linux kernel ZSTD compression code, shown in Listing 3.

7.4 Breaking KASLR

Kernel Address Space Layout Randomization (KASLR) randomizes the base addresses of various kernel memory regions [27]. The kernel image region includes our gadgets and references to the other regions. With SMAP (enabled by default), the kernel can only access the reload buffer via the *physmap* memory region, which references all available physical memory. Hence, by *breaking KASLR*, the attacker derandomizes the kernel image and physmap regions and the physmap offset of the reload buffer.

Table 4: Summary of the observed primitives for all evaluated platforms.

Vendor	Microarch.	Year	Defense	Primitive					Exploitable
				ABPU[a]	BPRC$_{U\to K}$	BPRC$_{G\to H}$	BPRC$_{IBPB}$	Bypass Mitig.[b]	
Intel	Raptor Cove (RPL-R)	2023	eIBRS	✓	✓	✓	✓	✓	✓
	Gracemont (RPL-R)			✓	✓	✗	✗	✓[c]	
	Golden Cove (SPR)	2023	eIBRS	✓	✓	✓	✓	✓	✓
	Raptor Cove (RPL)	2022	eIBRS	✓	✓	✓	✓	✓	✓
	Gracemont (RPL)			✓	✓	✗	✗	✓[c]	
	Golden Cove (ADL)	2021	eIBRS	✓	✓	✓	✓	✓	✓
	Gracemont (ADL)			✓	✓	✗	✗	✓[c]	
	Cypress Cove (RKL)	2021	eIBRS	✓	✓	✓	✓	–[d]	✓
	Skylake (CML)	2019	eIBRS	✓	✓	✓	✓	–[d]	✓
	Skylake (CFL-R)	2018	eIBRS	✓	✓	✓	✓	–[d]	✓
AMD	Zen 5	2024	AutoIBRS	✗	–	–	–	–	✗
	Zen 4	2022	AutoIBRS	✗	–	–	–	–	✗
ARM	Cortex-X1	2021	CSV2	✗	–	–	–	–	✗
	Cortex-A76	2021	CSV2	✗	–	–	–	–	✗

[a] Asynchronous Branch Predictor Updates; [b] BPI exploitable despite `BHI_DIS_S`; [c] mitigation reduces success rate; [d] mitigation unavailable

Kernel image. The kernel image base is randomized with 9 bits of entropy (*i.e.*, bits $[29,21]$). As in previous work [12], we use BTB collisions to derandomize its location. However, recent Intel CPUs use insufficient BTB addressing bits (*i.e.*, bits $[23,0]$) to recover the location using their exact technique (Intel Haswell in [12] uses bits $[30,0]$). Instead, we recall that BTB targets are 32 bits wide, and that transiently executed branch instructions update the BTB [30, 38]. We can detect these BTB updates, as they may overwrite branch predictions from branches aliasing in the BTB (*i.e.*, matching lower 24 bits). Hence, for all possible kernel image locations, we use BPI on an indirect branch with a target that contains a second branch. Only if we guess the correct kernel base, the second branch executes and overwrites the prediction of an aliasing branch that we allocate in user space. We detect whether the prediction of our aliasing branch is still present in the BTB using a cache gadget at the trained target of the branch.

```
1 acpi_ns_check_sorted_list.part.0.isra.0:
2 ;...
3 mov  rax, qword ptr [r13 + 8]
4 mov  rdx, qword ptr [rax + r12]
5 ;...
```

Listing 4: Disclosure gadget for loading a 64-bit secret, useful to retrieve an address like the physmap base pointer and access it at an offset.

Reload buffer. We now need a kernel pointer to the reload buffer via physmap. The attacker maps the reload buffer to a Transparent Huge Page (THP) (enabled by default in Ubuntu), which reduces its possible locations by a factor of 512, since THP forces it to a 2 MiB-aligned physmap offset. We will first find the reload buffer physical address (*i.e.*, its physmap offset), then derandomize the physmap base address, as in [43].

Since we have recovered the kernel image base, we have a pointer to a pointer to physmap via the symbol *page_offset_base*. We pass the address of this symbol in (attacker-controllable) *r13* to the disclosure gadget in Listing 4. The gadget loads a full 64-bit value in the first instruction (unlike Listing 3), allowing us to load the physmap pointer from memory and combine it with another attacker-controlled offset in *r12*. By controlling *r12*, we scan through physmap for the physmap offset of the reload buffer. When *r12* matches this offset, it will emit a cache hit after executing the gadget. Finally, we derandomize the physmap base by testing all its possible locations (15 bits of entropy [27]), with the reload buffer's offset added to it, in *r13* until we observe a reload buffer cache hit.

8 Evaluation

We first evaluate the presence of BPRC variants and if they can be exploited by BPI across a variety of eIBRS/AutoIBRS/CSV2-enabled processors. Next, we evaluate the KASLR exploit and the arbitrary memory leak primitive. Finally, we evaluate our end-to-end exploit that leaks `/etc/shadow` to an unprivileged user, demonstrating the impact of BPI. We run the exploits on an Intel Raptor Lake processor with Ubuntu 24.04 and an up-to-date default kernel (version 6.8.0-47-generic at the time of writing).

Primitives. Table 4 summarizes the presence of BPRC variants for all evaluated systems. We found ABPU on *all* evaluated Intel microarchitectures. All processors with ABPU (*i.e.*,

since the introduction of eIBRS) were vulnerable to BPRC on the user-to-kernel, guest-to-hypervisor, and IBPB boundaries. Furthermore, the on-by-default `BHI_DIS_S` mitigation on supported Intel CPUs did not prevent BPI. ARM and AMD processors showed no indication of BPRC issues.

KASLR. We evaluated the KASLR derandomization median time and accuracy over 1000 executions. Before each execution, we rebooted the system to re-randomize KASLR. We derandomized the kernel image offset in 15ms with 98.4% accuracy, and we found the reload buffer and physmap locations after 21ms and 102ms, respectively, with 98.2% accuracy.

Leakage rate. For the arbitrary memory leak primitive, we measured the median bandwidth and byte-accuracy over 1000 executions. We leaked a 1 MiB kernel-allocated buffer of randomized data at 5.6 KiB/s with 99.8 % byte-accuracy.

End-to-end exploit. Our end-to-end exploit demonstrates the impact of BPI by leaking the root password hash from `/etc/shadow` as an unprivileged user. Because this file is normally read during boot, it often remains present in Linux's page cache until there is memory pressure. Otherwise, executing `passwd -S` brings it back into page cache. The exploit first breaks KASLR and then scans physmap for `/etc/shadow` using the arbitrary memory leak. We can identify the file, as it is page-aligned and usually starts with `root:$`. Over 1000 executions, our end-to-end exploit completed with a median of 21 s, leaking the full root hash 90.7% of the time. 97.9% of the time, there were at most 6 errors in the leaked hash.

9 Mitigation

BPRC compromises the security guarantees provided by eIBRS on modern Intel processors. We suggest different mitigation strategies for $\text{BPRC}_{U\to K}$ and $\text{BPRC}_{G\to H}$ and empirically evaluate their performance overhead.

According to our experiments, eIBRS cannot be disabled on the newest generation of Intel processors. As such, alternate mitigations will have redundant performance overhead from eIBRS. We measure this overhead for the relevant mitigations using the Unixbench [1] and lmbench [2] test suites, in line with previous work [15, 38, 43]. We choose the current defaults in Linux as baseline for the evaluation, and we accumulate the overhead of our mitigations as in previous work [38]. For each configuration, we run both test suites five times and select the median result for each of the benchmarks in the test suites. We calculate a performance score for each test suite as the geometric mean over the benchmarks and report the change in performance score as the overhead. Table 5 shows the overhead compared to the baseline of the mitigation strategies, which we discuss in detail next.

[1] https://github.com/kdlucas/byte-unixbench
[2] https://github.com/intel/lmbench

Table 5: Mitigation overhead in UnixBench / lmbench.

CPU	Mitigation	
	IPRED_DIS_S	*Retpoline*
Sapphire Rapids	1.7% / 6.4%	2.4% / 8.0%
Raptor Lake	1.1% / 6.3%	2.2% / 6.6%
Alder Lake	1.5% / 4.7%	2.4% / 8.2%
Rocket Lake	-	3.1% / 8.3%
Comet Lake	-	0.5% / 2.4%
Coffee Lake Refresh	-	0.5% / 1.6%

9.1 Eliminating indirect branches

Retpoline. Retpolines [39] replace indirect branches in an executable. However, since previous work has shown that return predictions are also vulnerable to BTI through RSBA predictions [25, 43], all returns in the program prone to RSB underflow need to be protected as well. To mitigate BHI in depth, Intel recommends using Retpolines in combination with the `RRSBA_DIS_S` speculation control to remove indirect branches and disable RSBA-predicted returns. The same approach appears applicable to mitigate BPI as well. While some processors in our evaluation (RKL, CML, CFL-R) do not support `RRSBA_DIS_S`, they also do not seem to be affected by $\text{BPRC}_{U\to K}$ through the RSBA predictor. This is either because they have no RSBA (RKL) or because the RSBA provides predictions through the IBP (CML, CFL-R) [44].

We empirically validated Retpolines-`RRSBA_DIS_S` combination by repeating the experiment from Section 5.2 on `ret`. The results showed no more cache gadget hits. With up to 3.1 %/8.3 % (UnixBench/lmbench) overhead, this mitigation is not as efficient as the isolation approach discussed in Section 9.2 but is still a viable option.

Alternatives. Alternatives have been proposed to avoid the forced return mispredictions caused by Retpolines. JumpSwitches [6] dynamically train a set of targets that can then be reached using direct jumps while falling back to Retpoline otherwise. Switchpoline [16] for ARM expands this concept by compiling statically known targets into a switch-like structure such that only dynamic targets need to be added with just-in-time compilation. Completely eliminating indirect branches is particularly interesting since it avoids all the complexity that arises from the many different microarchitectures, each of which requires or provides a different mitigation. While these mitigations seem complete, they assume that mispredicted direct branches never lead to side-channel observable effects that leak secrets. It remains to be seen whether this assumption is upheld in the future.

9.2 Isolation

IPRED_DIS_S. Newer Intel CPUs provide a supplementary speculation control to mitigate speculation attacks on indirect control flow instructions. `IPRED_DIS_S` prevents speculative execution at the predicted target of indirect jumps and indirect calls until the target is resolved. It further prevents the same for RSBA-predicted returns [20]. We verify the effectiveness of this mitigation empirically by repeating the experiment from Section 5.2 with `IPRED_DIS_S` enabled and we observe no more cache gadget hits on any supported microarchitecture. Even though this completely disables speculative execution of indirect branch targets, it incurs a lower overhead of up to 1.7 %/6.4 % (UnixBench/lmbench) compared with the Retpolines-`RRSBA_DIS_S` combination.

IBPB. Using IBPB on kernel/hypervisor entry, like many AMD parts do [38], is not a viable option. As we found in Section 5.3, even IBPB can be bypassed by BPRC_{IBPB}.

Microcode Update. According to the response by Intel PSIRT, a *microcode update* is required as an in-depth mitigation of BPRC. This is due to potential other variants of BPRC that may exist, and to fix BPRC_{IBPB}. Intel provided us with a pre-release microcode update for our Alder Lake processor in January 2025. We found that none of the three BPRC variants discussed in this paper show a signal anymore after the microcode update. The overhead turns out to be 1.4 %/2.7 % (UnixBench/lmbench), which is lower than completely disabling indirect branch prediction in kernel mode.

10 Related Work

Asynchronous microarchitectural flaws. Meltdown [29] exploits privilege checks asynchronous to data cache accesses, leading to transient use of memory in dependent operations, regardless of privilege domain restrictions. Foreshadow [10, 41] exploits present-bit checks asynchronous to data cache lookups, leading to a cascade of violations. A range of similar vulnerabilities were discovered under the collective name of MDS [11, 32, 36, 40], with the common denominator of transiently forwarding uninitialized data from various microarchitectural buffers during asynchronous checking of its presence. In this paper, we introduced BPRC, a class of vulnerabilities where asynchronous branch prediction updates cause race conditions, breaking multiple defenses against BTI attacks.

BTI attacks. BTI attacks were introduced as Spectre Variant 2 [26]. Existing hardware would either use *Retpolines* [39] or retrofitted hardware-enforced IBRS restrictions [18] against BTI, while newer Spectre-aware microarchitectures would use eIBRS, AutoIBRS or CSV2 [7, 18, 34]. *Retbleed* demonstrated vulnerabilities in return target prediction under RSB underflow on Intel parts without eIBRS, and an instance of BTC on AMD family 17h [43]. *Phantom* demonstrated a generalized view of BTC, including short-lived mispredictions detected and corrected during instruction decode [45]. Based on Phantom, the *Inception* attack injected return target predictions via a confused deputy on modern AMD processors [38]. Milburn et al., presented sibling-thread workloads, exposing vulnerabilities in AMD-style Retpolines and short-lived BTC mispredictions [31]. *BHI* attacks exploited branch history that is shared across privilege domains [8, 42], allowing limited attacker-control over indirect branch predictions across privilege domains. While particular dispatch gadgets allowed BHI to inject arbitrary branch target predictions via a confused deputy [42], our BPI primitive injects arbitrary branch target predictions across privilege domains. Wikner and Razavi showed that Intel's IBPB failed to invalidate IP-based RSBA predictions on Golden Cove and Raptor Cove cores and presented the first real-world cross-process BTI attack [44].

Yavarzadeh et al. [47] reverse engineered the BHB update function for conditional branches, which was shown to be identical in follow-up work on indirect branch prediction [28], enabling a synthetic ASLR derandomization attack. Bunnyhop [48] showed that the instruction prefetching on Intel processors is BTB-dependent. In contrast to BPRC, these attacks focus on the fetch and decode stages for exploitation and the eIBRS guarantees are upheld.

11 Conclusion

We introduced *Branch Predictor Race Conditions (BPRC)*, an event-misordering effect where asynchronous BPU operations work independent of the instruction stream. In particular, we demonstrated BPRC variants where the BPU operates on incorrect privilege domain state, and showed that it affects all Intel CPUs with eIBRS-enabled branch predictors. This insight allowed us to build *Branch Privilege Injection (BPI)*, a new BTI primitive for injecting predictions tagged with supervisor privilege from user mode, demonstrating how a broken hardware defense mechanism has devastating impact on the security of modern computer systems. Our end-to-end BPI exploit leaks arbitrary privileged memory from up-to-date Linux systems across six generations of eIBRS-enabled Intel CPUs, at 5.6 KiB/s on Intel Raptor Cove, respectively. Furthermore, we demonstrated two additional variants of BPRC which work accross the guest-to-hypervisor and IBPB security boundaries. We proposed and evaluated potential mitigations by combining software defenses with model-specific speculation controls.

Acknowledgments

We would like to thank the anonymous reviewers and shepherd for their constructive feedback. We would also like to thank the Intel PSIRT and security research teams for coordination and collaboration on this issue. In particular, we thank

Alyssa Milburn, Ke Sun, Thaís Moreira Hamasaki from Intel STORM and Stephen Haruna and Priya Iyer from Intel PSIRT. We are grateful to one of our anonymous reviewers and the Intel STORM team for pointing out BPRC$_{IBPB}$. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

Ethics considerations

The presented work concerns micrarchitectural vulnerability research. During this research no human subjects were involved at any point. Unfortunately, disclosure of security vulnerabilities always poses a risk to people using affected systems. To mitigate this risk, we engaged in responsible disclosure with Intel. In line with industry practice, the vulnerabilites were not publicly disclosed before the agreed upon embargo date (May 13, 2025). Furthermore, we have contacted AMD and ARM such that they can verify their security in light of BPRC.

We are confident that our research has not violated any legal standards, is in the interest of computer security around the world and adheres to the USENIX Security '25 Ethics Guidelines.

Open science

In this work, we present a variety of microarchitecture evaluation experiments which are based on specialized C code with hand-written assembly. We ran the experiments using Ansible to collect additional information about the environment. Based on the results, we generated all graphs automatically using Python. Furthermore, our work involves proof of concept code to exploit the presented vulnerabilities end-to-end on a real system. In line with USENIX Security '25 open science policy, we make these artifacts available to the public at https://github.com/comsec-group/bprc and https://doi.org/10.5281/zenodo.14636810. We are confident that these artifacts allow the security community to verify and reproduce the presented results.

References

[1] AMD. Indirect Branch Control Extension. 2018. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/111006-architecture-guidelines-update-amd64-technology-indirect-branch-control-extension.pdf.

[2] AMD. Technical guidance for mitigating branch type confusion, 2022. URL: https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf.

[3] AMD. Software optimization guide for the amd zen4 microarchitecture. 2023.

[4] AMD. Software techniques for managing speculation on amd processors. 2023.

[5] AMD. Technical update regarding speculative return stack overflow. 2024. URL: https://www.amd.com/content/dam/amd/en/documents/corporate/cr/speculative-return-stack-overflow-whitepaper.pdf.

[6] Nadav Amit, Fred Jacobs, and Michael Wei. Jump-Switches: Restoring the performance of indirect branches in the era of spectre. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 285–300, Renton, WA, July 2019. USENIX Association.

[7] ARM. The Armv8.5 architecture extension. 2024. URL: https://developer.arm.com/documentation/109697/2024_12/Feature-descriptions/The-Armv8-5-architecture-extension.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, Boston, MA, August 2022. USENIX Association.

[9] Bradley D. Hoyt, Glenn J. Hinton, David B. Papworth, Ashwani K. Gupta, Michael A. Fetterman, Subramanian Natarajan, Sunil Shenoy, and Reynold V. D'Sa. Method and apparatus for implementing a set-associative branch target buffer. *US Patent*, 1996.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *SEC*. USENIX, 2018.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*. ACM, 2019.

[12] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE.

[13] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 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*, ASPLOS '18, page 693–707, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3173162.3173204.

[14] Thomas Gleixner. LKML: [patch 00/38] x86/retbleed: Call depth tracking mitigation, 2022. URL: https://lore.kernel.org/lkml/f9fd86acac4f49bc8f90b403978e9df3@AcuMS.aculab.com/t/.

[15] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. Quarantine: Mitigating transient execution attacks with physical domain isolation. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, pages 207–221, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3607199.3607248.

[16] Lorenz Hetterich, Markus Bauer, Michael Schwarz, and Christian Rossow. Switchpoline: A software mitigation for spectre-btb and spectre-bhb on armv8. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 217–230, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Intel Corp. Indirect Branch Predictor Barrier. 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

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

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

[20] Intel Corp. Branch History Injection and Intra-mode Branch Target Injection / CVE-2022-0001, CVE-2022-0002 / INTEL-SA-00598, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[21] Intel Corp. Post-barrier Return Stack Buffer Predictions / CVE-2022-26373 / INTEL-SA-00706. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/post-barrier-return-stack-buffer-predictions.html.

[22] Intel Corp. Retpoline: A Branch Target Injection Mitigation. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[23] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer Manuals. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[24] Manikandan Jagatheesan. Performance Regression in Linux Kernel 5.19, 2022. URL: https://lore.kernel.org/lkml/PH0PR05MB8448A203A909959FAC754B7AAF439@PH0PR05MB8448.namprd05.prod.outlook.com/.

[25] Andi Kleen. LKML: Improve retpoline for Skylake, 2018. URL: https://lkml.org/lkml/2018/1/12/605.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[27] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *EuroS&P*. IEEE, 2020.

[28] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2137–2154, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[30] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *13th USENIX Workshop on Offensive*

Technologies (WOOT 19), Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/woot19/presentation/mambretti.

[31] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You cannot always win the race: Analyzing mitigations for branch target prediction attacks. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 671–686. IEEE.

[32] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7179–7193, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi.

[33] Borislav Petkov. x86/srso: Add a speculative ras overflow mitigation. 2024. URL: https://github.com/torvalds/linux/commit/fb3bd914b3ec28f5fb697ac55c4846ac2d542855.

[34] Kim Phillips. LKML: [PATCH 0/3] x86/speculation: Support Automatic IBRS, 2022. URL: https://lkml.org/lkml/2022/11/4/1199.

[35] Till Schlüter, Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. Fetchbench: Systematic identification and characterization of proprietary prefetchers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 975–989, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3623124.

[36] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[37] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:23, 2006.

[38] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, Anaheim, CA, August 2023. USENIX Association.

[39] Paul Turner. Retpoline: a software construct for preventing branch-target-injection, 2018. URL: https://support.google.com/faqs/answer/7625886.

[40] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[41] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[42] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 577–594, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/wiebing.

[43] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, Boston, MA, August 2022. USENIX Association.

[44] Johannes Wikner and Kaveh Razavi. Breaking the Barrier: Post-Barrier Spectre Attacks . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 89–89, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00089, doi:10.1109/SP61157.2025.00089.

[45] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 49–61. ACM.

[46] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[47] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237. IEEE.

[48] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security*, 2023.