Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks

Enrico Barberis[†] Pietro Frigo[†] Marius Muench Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
{e.barberis, p.frigo, m.muench}@vu.nl
{herbertb, giuffrida}@cs.vu.nl

[†] Equal contribution joint first authors

Abstract

Branch Target Injection (BTI or Spectre v2) is one of the most dangerous transient execution vulnerabilities, as it allows an attacker to abuse indirect branch mispredictions to leak sensitive information. Unfortunately, it also has proven difficult to mitigate, with vendors originally resorting to inefficient software mitigations like retpoline. Recently, efficient hardware mitigations such as Intel eIBRS and Arm CSV2 have been deployed as a replacement in production, isolating the branch target state across privilege domains. The assumption is that this is sufficient to deter practical BTI exploitation. In this paper, we challenge this belief and disclose fundamental design flaws in both Intel and Arm solutions.

We introduce *Branch History Injection* (BHI or Spectre-BHB), a new primitive to build cross-privilege BTI attacks on systems deploying isolation-based hardware defenses. BHI builds on the observation that, while the branch target state is now isolated across privilege domains, such isolation is not extended to other branch predictor elements tracking the branch history state—ultimately re-enabling cross-privilege attacks. We further analyze the guarantees of a hypothetical isolation-based mitigation which also isolates the branch history and show that, barring a collision-free design, practical same-predictor-mode attacks are still possible. To instantiate our approach, we present end-to-end exploits leaking kernel memory from userland on Intel systems at 160 bytes/s, in spite of existing or hypothetical isolation-based mitigations. We conclude software defenses such as retpoline remain the only practical BTI mitigations in the foreseeable future and the pursuit for efficient hardware mitigations must continue.

1 Introduction

Of all the transient execution vulnerabilities [8, 20, 34, 35, 37, 39, 40, 43–45, 49, 55–58, 63] discovered since the first disclosure of such issues in 2018 [62], one of the most worrisome was Branch Target Injection (BTI or Spectre v2 [34]). As the name suggests, it allows attackers to *inject* speculative *branch targets* into a victim's context—while also ignoring the architectural privilege boundaries [9, 34]. That is, an unprivileged attacker can transiently control the execution of a more privileged victim (e.g., the kernel or hypervisor).

Given the severity of the issue, vendors have devised a kaleidoscope of security mitigations. After a first generation of heavyweight hardware mitigations [4, 27, 51]—deemed exceedingly inefficient by the community [54]—the software-based retpoline mitigation reached widespread adoption [21, 28]. While retpoline can eradicate BTI attacks by replacing indirect branches with return instructions, it also cripples indirect branch prediction altogether (other than reducing the efficiency of return address prediction) and still incurs non-trivial performance overhead [1, 17]. As a replacement, hardware vendors have recently deployed more efficient hardware mitigations such as Intel eIBRS [26] and Arm CSV2 [4], which *isolate* the branch-target-buffer entries across privilege domains—effectively re-enforcing privilege boundaries in the transient realm. The current assumption is that this mitigation strategy is sufficient to deter practical BTI exploitation without sacrificing indirect prediction benefits.

In this paper, we challenge this belief by analyzing the residual attack surface of the most recent CPU generations. More specifically, we investigate the security guarantees of Intel, AMD, and Arm solutions and identify fundamental design flaws in Intel eIBRS and Arm CSV2 defenses that allow for cross-privilege BTI attacks. We disclose *Branch History Injection* (BHI or Spectre-BHB): a new primitive that allows attackers to build practical BTI attacks despite the new isolation-based hardware defenses. At the core of this conclusion lies the observation that these isolation guarantees are limited to specific elements of the branch predictor (i.e., the branch target buffer), while others, such as those tracking the previous branches' history, are neglected. Hence, unprivileged attackers can re-enable cross-privilege BTI attacks via BHI even on the latest Intel and Arm CPUs, as they can still control the speculative target of higher-privilege (i.e., kernel's or hypervisor's) indirect branches.

To demonstrate the practicality of this primitive, we focused on Intel systems and implemented an end-to-end userland exploit leaking arbitrary memory from a fully protected Linux kernel at 160 bytes/s. To the best of our knowledge, this is the first cross-privilege BTI attack against an unmodified system with hardware isolation-based mitigations such as eIBRS in place. We further show that even hypothetical refinements of such isolation-based mitigations (i.e., which also isolate the branch history across privilege domains) are still insufficient. In other words, we show that generic intra-mode BTI attacks beyond BHI—previously considered unfeasible—are a realistic threat. As a concrete demonstration, we expand on the previous exploit to mount an end-to-end confused-deputy BTI attack where *both* the injection and the transient execution of the disclosure gadget occur entirely during kernel execution.

After exposing the limitations of hardware-based isolation against BTI attacks, we discuss mitigations and formulate recommendations for existing production systems. In particular, we suggest reverting to software defenses such as retpoline, even on the last-generation systems, and further restricting the transient execution attack surface by disabling exploitation-friendly features such as unprivileged eBPF.

Contributions. We make the following contributions.

- We characterize modern mitigations against cross-privilege BTI attacks and perform a security analysis of Intel eIBRS and Arm CSV2.
- We disclose issues with Intel and Arm mitigations and introduce Branch History Injection: a new technique to enable cross-privilege BTI exploits even on modern systems (CVE-2022-0001, CVE-2022-23960).
- We showcase the first end-to-end userland exploit against the kernel leaking arbitrary data from an unprivileged user at 160 bytes/s on the latest Intel eIBRS-equipped systems and discuss mitigations.
- We demonstrate that intra-mode BTI exploits beyond BHI are a realistic threat, evidencing fundamental limitations of isolation-based defenses (CVE-2022-0002).

Additional information about BHI, including our proof-of-concept exploits and analysis code, is available online at https://vusec.net/projects/bhi-spectre-bhb.

2 Background

In this section we provide the necessary background on modern branch prediction and BTI attacks.

2.1 Branch Prediction

The Branch Prediction Unit (BPU) is a fundamental component of a modern CPU's front-end. This unit *predicts* the target of upcoming branches based on previous behavior in order to fetch and speculatively execute the upcoming instructions. It typically deploys separate logic for the prediction of *direct* and *indirect* branches. While information on the branch source address and whether the branch was taken in the past are usually sufficient for efficient direct branch prediction, indirect branch prediction is by far more complex.

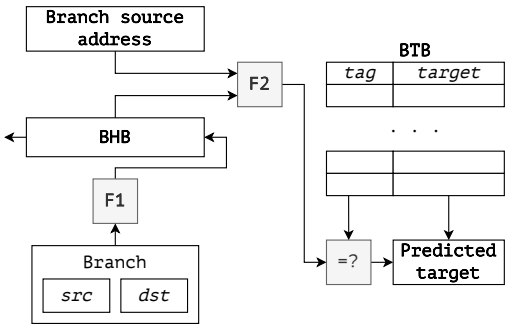

Figure 1: A simplified diagram of indirect branch prediction logic.

According to the literature [10, 15, 16, 32, 33], to accurately predict indirect branches with multiple targets, it is necessary to store some *context* associated with a given branch. The idea is that recent execution will likely correlate with the branch target since it is selected dynamically depending some conditions (e.g. a `switch-case` to compute the indirect branch target). Previous work [18, 19, 34, 62, 64] partially reverse engineered the branch prediction logic for several systems, and, despite the major microarchitectural differences, they observed macro-level components common to most modern BPUs. In Figure 1, we show a simplified diagram of such an indirect prediction logic. This incorporates a Branch Target Buffer (BTB) and a Branch History Buffer (BHB).

The BTB is a cache that stores the most recent target addresses for different branches and branch contexts. This cache is addressed with a tag computed from the branch source address and the branch context (F2). To create the context, the branch predictor takes both the source address and the *history* of previously executed branches into account, stored as a hash inside the BHB [15, 34, 64]. This hash is constructed over the source and destination address of previous branches, as well as the current state of the BHB. The number of branches tracked this way and the hash function F1 applied to the addresses depend on the microarchitecture. Prior reverse engineering efforts revealed that for older Intel microarchitectures, the BHB is implemented as a shift register which gets updated by XORing its rightmost bits with the folded source and destination address of a taken branch [34, 64].

Even if the indirect branch predictor internals and naming widely change among different microarchitectures, the majority of them use a context-based approach [10, 15, 23, 34, 64]. For simplicity, we refer to BHB as a generic predictor structure that holds such context.

2.2 Branch Target Injection

With the disclosure of Spectre in 2018 [34, 62], researchers discovered the possibility of exploiting the branch predictor in numerous ways to read data outside of sandboxed environments [34, 43], across privilege boundaries [34, 37, 55], and even outside of secure enclaves [11, 55]. Among all the different variants, our work focuses on *Branch Target Injection* (BTI)—also known as *Spectre v2* or *Spectre-BTB* [9].

BTI is an intrinsic vulnerability of the indirect branch predictor described in Section 2.1. Since the target address of indirect branches is available only at runtime, modern processors try to predict the branch target and speculatively fetch and execute the instructions at the predicted location. In BTI, an attacker abuses this behavior to *mistrain* a victim indirect branch and predict a target address containing a gadget to leak sensitive data [34, 61, 62]. While for conditional branches only the taken or not-taken paths can be transiently executed, BTI has the flexibility to transiently hijack the control flow and execute arbitrary code in the victim's context.

One can derive different BTI variants [9], depending on whether the speculatively executed gadget resides at the same privilege level of the victim branch (*intra-mode*) or at a different privilege level (*inter-mode*), and whether the mistraining and the misprediction happens at the same branch (*in-place*) or using a different aliased branch (*out-of-place*).

3 Overview

Since the introduction of mitigations meant to reinforce hardware privilege boundaries against BTI [34], the common assumption is that such *transient privilege escalation* attacks are impractical, if not infeasible. In this paper, we challenge this assumption by answering the following questions:

1. *What is the current status of Spectre-v2 defenses?* In Section 4, we outline the fragmented landscape of kernel defenses against BTI. We examine the guarantees of these solutions, their use on current systems, and the challenges they lay out for an attacker.

2. *Can we still poison the target of kernel indirect branches from userland?* After characterizing the defenses, we analyze the state-of-the-art hardware implementations (i.e., Intel eIBRS and Arm CSV2) in detail in Section 5 and disclose Branch History Injection (BHI): a new primitive that allows an unprivileged attacker to control mispeculation of higher-privilege (i.e., kernel's or hypervisor's) indirect branches by manipulating non-isolated parts of the branch context.

3. *Are such attacks still practical?* To demonstrate the severity of our findings, Section 6.2 leverages BHI to build an end-to-end BTI exploit against an unmodified kernel running on an eIBRS-enabled system. Furthermore, in Section 6.3, we showcase an *intra-mode* (i.e., kernel-to-kernel) exploit to demonstrate that isolation-based defenses, even when they consider the entirety of the indirect branch prediction state, still allow for practical same-predictor-mode attacks.

4 Spectre-v2 Defenses

In this section, we describe all the software and hardware Spectre-v2 defenses that have been deployed across different generations of Intel, AMD, and Arm processors [4, 27, 28, 51]. We focus on the newer in-silicon solutions protecting against cross-privilege attacks to better understand the residual attack surface available after their adoption.

4.1 Software Defenses

Among software Spectre-v2 defenses we find the following:

- **Generic retpoline.** Grown into the standard software defense against BTI, retpoline is a special code sequence that converts an indirect branch to a `ret` instruction to ensure that the *Return Stack Buffer* (RSB) is used rather then the BTB. Retpoline "traps" into an infinite loop any possible mispeculation until the actual address is resolved and popped from the stack [21]. As a result, no indirect branch prediction is performed. While retpoline is documented to work for most Intel and AMD processors [27, 52], it is not effective on Arm [5].

- **AMD retpoline.** AMD proposed an alternative retpoline defense [52], where every indirect branch is turned into a `lfence/jmp` sequence. The `lfence` ensures that the load for the indirect branch target is retired before the jump, making the residual transient window small enough to hinder exploitation [52]. This solution is tailored to AMD processors [60] and performs better than generic retpoline [41].

- **Arm defenses.** For older microarchitectures not supporting hardware mitigations, Arm suggests the invalidation of the branch predictor entries on mode switch by either executing the `BPIALL` instruction, or by disabling and re-enabling the MMU [12]. Arm implemented a Secure Monitor Call named `SMCCC_ARCH_WORKAROUND_1` to execute the appropriate defense depending on the affected processor.

4.2 Hardware Defenses

In this section, we discuss the hardware defenses proposed by Intel, AMD, and Arm. We expand on the more relevant ones for cross-privilege BTI attacks.

- **Intel/AMD - IBPB.** *Indirect Branch Prediction Barrier* is a strong barrier to ensure that the execution of previous indirect branches cannot influence the prediction of subsequent branches executed after the barrier [27, 51].

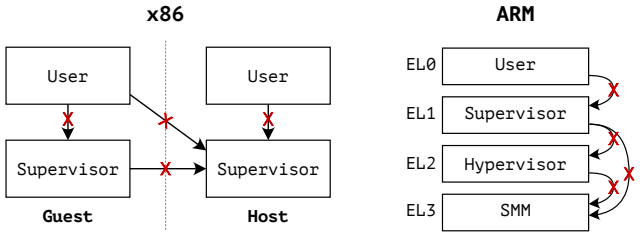

Figure 2: Predictor modes on x86-64 and Arm. The arrows show the possible transitions between lower-level to higher-level predictor modes and ✗ indicates which branch target injection attack vectors should be prevented by the respective cross-privilege hardware mitigations (i.e., eIBRS, CSV2).

- **Intel/AMD - STIBP.** *Single Thread Indirect Branch Predictors* restricts the sharing of the branch prediction state among hyperthreads on the same physical core [27, 51].
- **Intel/AMD - (e)IBRS.** Intel and AMD propose *Indirect Branch Restricted Speculation (IBRS)* to stop cross-privilege Spectre-v2 attacks [27, 51]. This solution intends to ensure that indirect branch prediction cannot be controlled by software that is executed in a lower-privilege predictor mode. There are 4 relevant predictor modes on x86-64: host-supervisor, host-user, guest-supervisor, and guest-user [27, 51]. As shown in Figure 2, predictor modes define different privilege level pairs. IBRS aims to prevent the lower-privilege parts from influencing the indirect branch prediction of the more privileged layers.

  The support for IBRS was added with a micro-code update and the defense is enabled by writing to the `IA32_SPEC_CTRL.IBRS` register on every mode switch to signal the predictor mode switch—inducing a substantial performance overhead [54]. To remediate this performance impact, both Intel and AMD document more efficient IBRS variants for their newer microarchitectures [26, 51]—Intel's variant is known as *Enhanced-IBRS* (eIBRS) whereas AMD's variant is known as "always-on" IBRS. These variants seek to provide the same security guarantees of IBRS, but require writing to the `IA32_SPEC_CTRL.IBRS` register only once at boot time.

  The documentation states that it is not necessary to enable Single Thread Indirect Branch Predictor (STIBP) when IBRS is enabled [27], suggesting that IBRS performs tagging based on both predictor mode and hyperthread ID. In addition, AMD specifies an "IbrsSameMode" bit in its documentation [53] to protect against same-predictor-mode mispredictions. This effectively provides similar guarantees to those of IBPB.
- **Arm - FEAT_CSV2.** Arm's newer microarchitectures also introduce an eIBRS-like hardware solution protecting against cross-privilege BTI: `FEAT_CSV2` [6]. Since Armv8.5, the `ID_AA64PFR0_EL1` register specifies in the `CSV2` field which new hardware mitigations against cross-privilege BTI are implemented. Currently, this field indicates support for two solutions: (i) one that should prevent the *speculative* control of indirect branch targets from different hardware contexts (when `CSV2=1`); and (ii) one that should enforce the same guarantees also for different software contexts defined by the `SCXTNUM_ELx` register (when `CSV2=2`) [7].
- **Intel - CET-IBT.** Intel also released *Indirect Branch Tracking* (IBT) as part of its new *Control-Flow Enforcement Technology* (CET) in its Tiger Lake microarchitecture. CET-IBT prevents the execution—both speculative and architectural—of all the indirect branch targets that do not start with the `ENDBR32/64` instruction [29]. While it does not prevent speculation altogether, it seeks to reduce the number of available gadgets and hinder exploitation.

4.3 A Complex Adoption

While plenty of defenses against BTI are available, it is still unclear when each of these solutions is effectively deployed. In this section, we examine the adoption of defenses against cross-privilege BTI attacks. In particular, we consider their adoption in the Linux kernel across architectures (both x86-64 and Arm). We then expand to other hardware-software configurations and discuss their impact on exploitation.

Software defenses. Older-generation CPUs lacking in-silicon solutions need to rely on software defenses. We show their adoption in the Linux kernel in Table 1. We detect their use on: old Intel microarchitectures (e.g., Coffee-Lake R), where Linux simply relies on retpoline; and on old Arm Cortex designs (e.g., Cortex-A76), where it invalidates the branch predictor from software. Interestingly, even in the presence of hardware defenses, their software counterparts are sometimes preferred by the OS due to the performance impact of the former [51, 54]. For instance, we observe this behavior for the Intel Core i9-9900K, where Linux resorts to retpoline despite the support of IBRS, and for all AMD CPUs where it follows the vendor's recommendations to employ AMD retpoline [51].

Hardware defenses. For modern Intel and Arm microarchitectures, the Linux Kernel relies on the new hardware mitigations (as of version 5.14). For instance, for modern Intel CPUs (from Cascade Lake onward) it simply relies on eIBRS instead of deploying retpoline. The same applies to modern Arm CPUs, where the Android kernel avoids invalidating the branch predictor entries when CSV2 is set. Curiously, we identified two revisions (with and without support for CSV2) of the Arm Cortex-A76 microarchitecture, showing the fragmentation of the Arm ecosystems.

Table 1: Linux Spectre-v2 defenses on the Linux kernel version 5.14 (Intel & AMD), version 5.10 (Google), and 4.14 (Qualcomm). Please note that IBRS is available only with updated microcode.

Vendor	Model	μArch	Software Defenses		Hardware Defenses	
x86 Defenses			**Retpoline**	**AMD retpoline**	**IBRS**	**eIBRS**
Intel	Core i9-9900K	Coffee-Lake R	●	○†	○	—
	Core i7-10700K	Comet-Lake	○	○†	○	●
	Core i7-11700	Rocket-Lake	○	○†	○	●
	Core i7-11800H	Tiger-Lake	○	○†	○	●
	Xeon Silver 4214	Cascade-Lake	○	○†	○	●
	Xeon Silver 4310	Ice-Lake	○	○†	○	●
AMD	Ryzen 5 5600X	Zen 3	○	●	○	—
	Epyc 7662	Zen 2	○	●	○	—
Arm Defenses			**Retpoline**	**Workaround_1**	**CSV2=1**	**CSV2=2**
Google	Tensor	Cortex A55 ‡	○†	—	—	—
	Tensor	Cortex A76	○†	—	●	—
	Tensor	Cortex X1	○†	—	●	—
Qualcomm	Snapdragon 855	Cortex A76	○†	●	—	—

(●) Default solution, (○) Disabled, (—) Not available, † Not recommended by the vendor, ‡ Not affected by BTI.

On the other hand, on AMD we observed that even very recent CPUs do not support the "always-on" IBRS but only the older and more expensive IBRS variant—we verified this on the Ryzen 5600X released in late 2020. For this reason, AMD still recommends the use of AMD retpoline [51].

Deployment in other contexts. Other environments seem to have followed a similar trend of adopting hardware defenses when available. For instance, we analyzed Windows on Intel CPUs and we observed that, despite having adopted retpoline for a long time [31], newer builds rely on eIBRS similar to Linux—we empirically verified this behavior on an Intel i7-11800H with Windows version 21H2 build 19044.1288. We observed the same trend on common hypervisors: both Xen [13, 22] and KVM [42, 66] enforce guest-host separation by relying on eIBRS and CSV2 as preferred options when the hardware supports them.

Adoption impact. Considering the widespread deployment of these hardware solutions across OSes and hypervisors, it is crucial to understand their effectiveness against cross-privilege BTI attacks. These defenses promise to prevent speculative control of targets of any indirect branch across privilege boundaries. For instance, while previously attackers could directly mistrain the kernel from userland without any major impediment, this should no longer be possible. However, despite these claims, little is known about their security guarantees. For instance, Arm itself states that, on systems protected by CSV2, an attacker can still control the speculative execution of indirect branches "*in a hard-to-determine way*" [7]. This leaves open questions on the possible shortcomings of the proposed mitigations. In the next section, we analyze these solutions and disclose fundamental limitations in some of their implementations, showing how we can still build cross-privilege BTI attacks even on systems with targeted hardware mitigations.

5 Branch History Injection

In this section, we will first take a closer look at indirect branch prediction in the era of eIBRS and CSV2 hardware defenses. Based on our observations, we develop and describe *Branch History Injection* (BHI), an attack primitive which allows attackers to mount BTI attacks even in the presence of these in-silicon defenses.

5.1 Bypassing eIBRS and CSV2

While little is known about the inner workings of eIBRS, details about the implementation of CSV2 on Exynos M-series CPUs were disclosed in prior work [23]. Considering their very similar security guarantees, we can model both solutions as an isolation-based defense where the tag of each BTB entry is hashed using the hardware (and possibly software) context as a key—in the case of the Exynos predictor, this key is actually used to encrypt the branch target itself [23]. As such, the context hash also defines for which predictor mode the BTB entry is valid. This observation is additionally supported by the Intel documentation [26], which states that eIBRS does not protect against intra-mode BTI. Indeed, isolating the BTB entries would still allow same-mode mispredictions by design.

To characterize the remaining cross-privilege attack surface, we need to verify whether all the components of the indirect branch prediction logic are correctly isolated. As shown in Section 2, indirect branch prediction is a complex procedure that depends on multiple components such as the indirect branch source address and the BHB value encoding the history of previous branches. We hypothesize that especially

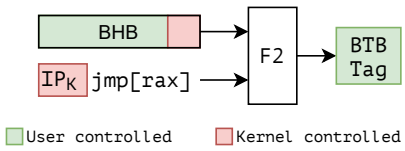

Figure 3: BTB tag computation for indirect branches after a user → kernel switch with improper BHB isolation. Even if the indirect branch source address IP_K and a small portion of the BHB is not under user control, an attacker can still affect the BTB-tag value.

the BHB may not be properly isolated due to (legitimate) performance reasons. The intuition is that the accumulated branch history before a switch to a different privilege level is essential to accurately predict the initial branches and improve critical-path performance. As an example, let us consider a syscall. When executing a syscall, one of the earliest operations performed by the Linux kernel is an indirect jump to the correct syscall handler function, whose address is stored in the `sys_call_table`. Since the initial kernel code path is always the same no matter the syscall, it is essential to use the recent *user* history to perform an accurate prediction. In this scenario, the BHB value would be filled with values mostly originating from the user history, as shown in Figure 3.

To test our hypothesis, we verify whether the BHB value is isolated after a mode switch to a privileged domain, by means of the experiment in Algorithm 1. The experiment associates two different histories with two separate syscalls by filling the BHB with different values. It does so by executing enough taken branches to fill the BHB. Specifically, history H_{load} is associated with a custom syscall leaving an observable microarchitectural trace—observable by means of a simple FLUSH + RELOAD (F+R) covert channel [61]. History H_{dummy}, on the other hand, is mapped to another arbitrary syscall (e.g., getpid). After the training in Lines 1–6, the indirect branch predictor associates the expected syscall target to the observed context (i.e., based on the user history). Finally, the experiment tests whether the predictor mispredicts to the custom syscall leaving a microarchitectural trace when mixing the history H_{load} with the dummy syscall. Surprisingly, despite the presence of the eIBRS and CSV2 defenses, we observed accurate mispredictions rates of $> 95\%$ for all Intel and Arm devices affected by Branch Target Injection (BTI).

> ***Observation* 1.** The BHB is not isolated among different privilege levels, allowing attackers to control indirect branch prediction for early branches in the higher-privilege mode.

We name this attack primitive *Branch History Injection* (BHI or Spectre-BHB). BHI is an "extension" of classic Branch Target Injection (BTI) in that it *partially* reintroduces the cross-privilege attack surface BTI lost after the introduction of eIBRS and CSV2. In fact, while BTI used to allow attackers

Algorithm 1 Experiment to verify BHB isolation between user and kernel mode. For the training phase, $n = 1$ is usually sufficient on the tested CPUs in Table 2.

```
1:  for i ← 1,n do                                  ▷ Training
2:      set_history(H_load)
3:      syscall → load(mem)                         ▷ H_load → load(mem)
4:      set_history(H_dummy)
5:      syscall → dummy                             ▷ H_dummy → dummy
6:  end for
7:  flush(mem)                                      ▷ Remove mem from cache
8:  set_history(H_load)
9:  syscall → dummy                                 ▷ Misprediction to load(mem)?
10: reload(mem)
```

to perform inter-mode attacks (e.g., user → kernel), eIBRS and CSV2 ought to limit them to intra-mode attacks (e.g., kernel → kernel). However, BHI can overcome these defenses, effectively allowing an unprivileged attacker to leverage their control over the BHB to divert execution—albeit, only to valid kernel targets—across privilege boundaries. Also note that, although BHI attacks are cross-privilege and rely on polluting the indirect branch prediction state with unprivileged data (i.e., history), they can still be considered a special subset of intra-mode BTI attacks—since the mistraining and the hijacking step still both occur in the higher-privilege mode.

We list the tested devices in Table 2. Our experiments indicate that Intel and Arm processors deploying eIBRS and CSV2 are vulnerable to BHI. Furthermore, we observed that the Arm Cortex-A76 deploying the software-based defense Workaround_1 is also vulnerable to BHI, suggesting that the mitigation does not invalidate the BTB as expected. In comparison, systems which rely on retpoline (i.e., AMD systems and most Intel systems without eIBRS) are not affected by BHI. Finally, we observed that AMD IBRS protects against both inter- and intra-mode BTI altogether, in line with their documentation [51]. That is, we could not speculatively control the target of *any* indirect branch executed after toggling IBRS—providing security guarantees comparable to IBPB.

BHI on other predictor modes. We extended our experiments to test if the BHB is isolated between the other major privilege boundaries on x86-64 processors[1]: guest-user → host-supervisor, guest-supervisor → host-supervisor and guest-user → guest-supervisor. In all scenarios, we observed identical results, demonstrating the general applicability of this vulnerability. Our results show that, even with the state-of-the-art eIBRS in-silicon mitigation, an attacker can still hijack more privileged indirect branch predictions by controlling only the BHB value.

[1] We did not extend these experiments to Arm processors, as our test platforms (Google Pixel 6 & Pixel 4) did not provide readily accessible interfaces to run custom code in EL2 or EL3.

Table 2: Vulnerability to BHI and corresponding attack surface of tested processors. "BHI vulnerable" specifies—for different defenses—if an unprivileged history can mistrain more privileged indirect branch predictions. "BHI attack surface" indicates whether BHI can be used to mistrain only from the same indirect branch (in-place BTI) or *any* indirect branch in the victim context (out-of-place BTI).

Vendor	Model	μArch	BTI vulnerable	BHI vulnerable		BHI attack surface*	
x86-64				**IBRS**	**eIBRS**	**BTI in-place**	**BTI out-of-place**
Intel	Core i9-9900K	Coffee-Lake R	✓	✓	—	✗	✗
	Core i7-10700K	Comet-Lake	✓	✓	✓	✓	✓
	Core i7-11700	Rocket-Lake	✓	✓	✓	✓	✓
	Core i7-11800H	Tiger-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4214	Cascade-Lake	✓	✓	✓	✓	✓
	Xeon Silver 4310	Ice-Lake	✓	✓	✓	✓	✓
AMD	Ryzen 5 5600X	Zen 3	✓	✗	—	✗	✗
	Epyc 7662	Zen 2	✓	✗	—	✗	✗
Arm				**Workaround_1**	**CSV2=1**		
Google	Tensor	Cortex A55	✗	—	—	✗‡	✗‡
	Tensor	Cortex A76	✓	—	✓	✓	✗
	Tensor	Cortex X1	✓	—	✓	✓	✗
Qualcomm	Snapdragon 855	Cortex A76	✓	✓	—	✓	✗

(✓) affected, (✗) not affected, (—) not available, (‡) Not affected by BTI, (∗) Adopting the default mitigation shown in Table 1.

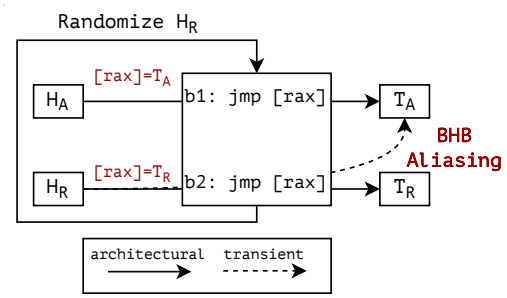

Figure 4: Brute-force approach to find colliding BHB values.

5.2 BHI Attack Surface

In the previous section, we demonstrated how Branch History Injection can re-enable cross-privilege BTI attacks. However, when limiting ourselves to the experiment in Algorithm 1, its attack surface is limited to that of intra-mode in-place BTI. That is, the only available gadgets would be valid kernel targets (*intra-mode*) of the specific indirect call being mistrained (*in-place*). In this section, we investigate the possibility of expanding the attack surface to other valid kernel indirect branch targets using BHI to cause *out-of-place* BTI [9, 11, 18, 34]. In other words, we want to understand if we can cause BTB-tag collisions between two distinct indirect branches by manipulating only the BHB value and not the branch address (see Figure 1). Enabling such primitive would allow us to identify gadgets across targets of different kernel indirect branches, considerably expanding the attack surface.

To verify this, we implemented a userland-only (i.e., same-predictor mode) experiment in which we execute two different indirect branches, one after the other, in a loop, as shown in Figure 4. The first branch has a fixed history H_A and a fixed target T_A, while the second one uses a random history H_R for every iteration and a fixed target T_R. Eventually, we want to observe a misspeculation to T_A under a random history H_R, rather than to the correct T_R, which would indicate colliding BHB values. We construct the histories by prepending the indirect branches with a chain of randomly located `jmp` instructions, as shown in Listing 1. Since the BHB value depends on the source and destination addresses of the previous branches [34, 62], we can randomize these values to randomize the BHB value as a result.

Listing 1 Example of a `jmp`-chain used to fill the BHB with a random value.

```
1 0x1337cafe: jmp 0xdeadbeef
2 ...
3 0xdeadbeef: jmp 0xd0d0caca
4 ...
5 0xd0d0caca: jmp [rax]
```

This experiment successfully found colliding BHB values for all the tested Intel CPUs, but not for any of the Arm and AMD processors in our test set. In Table 2, we report the results when operating cross-privilege mistraining with the default mitigations enabled, which shows that eIBRS-enabled Intel CPUs are susceptible to out-of-place BTI via BHI. We additionally observed that there are no requirements on the source address alignment of the two indirect branches. Indeed, we were able to find collisions no matter the values of *all* branch addresses in the experiment.

***Observation* 2.** On Intel systems, we can generate the same BTB tag for two distinct indirect branches by simply controlling the BHB value for one of these branches.

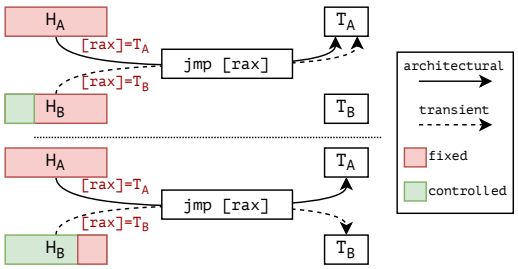

Figure 5: BHB size recovery. We start by providing two colliding histories $H_A = H_B$ so that we can observe a misprediction. We then keep H_A fixed and start changing old branches in H_B until we observe predictions to the right target T_B.

In other words, out-of-place BTI on Intel systems is possible just by controlling the BHB. This allows an attacker to transiently execute *any* valid indirect-branch target in the victim's context (i.e., any kernel indirect-call target).

Another interesting result is that a colliding history to a given branch context leads to high misprediction rates ($> 95\%$) when replayed against the same branch context. Hence, once found, an attacker can re-use a colliding history for a given branch context over and over to cause accurate branch mispredictions. Interestingly, we observed that colliding histories are even portable on different processors of the same generation with different microarchitectures. For example, the same collision found on a i7-10700K worked on a Xeon Silver 4214, suggesting that the same indirect branch prediction logic is shared among different microarchitectures.

In-place attack surface. In the remainder of the paper, we show how we can leverage BHI to exploit out-of-place BTI, since it provides a larger attack surface. Hence, we build end-to-end exploits only on Intel systems. Nevertheless, exploiting in-place BTI is still a realistic threat. For instance, Appendix A discusses an additional issue we reported to the Linux kernel developers where, on systems protected by eIBRS, all indirect jumps in the kernel were "funneled" to a handful of call sites due to the runtime patching performed for retpoline. Under such circumstances, given the plethora of valid targets available per indirect branch, in-place BTI attacks have essentially the same attack surface as out-of-place BTI ones.

Finally, even if we could not reproduce out-of-place BTI on Arm devices, when we repeated the same experiment with a single indirect branch, we observed colliding histories (Table 2)—confirming in-place BTI attacks are possible on Arm. Moreover, the fragmentation of the Arm ecosystem does not ensure the same implementation of predictors and the same security guarantees for all the systems. For instance, as we show in Table 1, we observed two different versions of A76-based architectures: one deploying CSV2, and one which does not. Similarly, two CPUs with the CSV2 register set may provide different guarantees depending on the licensee's design [23] and, as we discuss in Section 7, the mitigations deployed by Arm confirm the presence of vulnerable platforms.

Table 3: Indirect branch prediction reverse engineering of tested Intel processors. "BHB size" indicates how many taken branches affect the BHB value. "Tag entropy" specifies the brute-force entropy size of the BTB tag. "BHB control" specifies the minimum number of attacker-controlled branches to generate arbitrary BTB tags.

CPU	Tag entropy (*bits*)	BHB size (*#branches*)	BHB control (*#branches*)
Intel Core i7-10700K	14	29	≥ 9
Intel Xeon Silver 4214	14	29	≥ 9
Intel Core i7-11700	17	66	≥ 8
Intel Core i7-11800H	17	66	≥ 8
Intel Xeon Silver 4310	17	66	≥ 8

5.3 Understanding BHI on Intel

From an attacker perspective, it is useful to understand the capabilities and limitations of BHI. In this section, we investigate three main parameters. (1) We first want to understand what the "size" of the BHB is, i.e., the number of previous branches affecting the BHB value. (2) Then, since we opt for a microarchitecture-agnostic brute-force approach—in contrast to prior work that reverse engineered the BHB [34, 62]—we need to understand how long it takes to find a colliding history. (3) Finally, we want to understand how many user branches an attacker needs to control to generate BTB-tag collisions—i.e., how deep (from the syscall) the victim indirect branch can be.

BHB size. To find the BHB size (i.e., how many previous branches are tracked) we use the methodology described in prior work [34, 62, 64], based on a single indirect branch with two possible targets T_A and T_B. As shown in Figure 5, the experiment starts by training the predictor so that, with a history H_A, the indirect branch would always jump to target T_A, while with a history H_B it would always jump to target T_B. With two identical histories H_A and H_B, the predictor cannot make accurate prediction, always yielding a mispeculation to the latest target. Thus, by keeping H_A constant while incrementally modifying H_B, starting from older branches, we eventually stop observing mispredictions. This means that the predictor was finally able to correctly distinguish H_A from H_B, thus revealing the number of most recent branches affecting the BHB value. We report the results of this experiment in Table 3. As mentioned in Section 5.2, we observed that predictor designs may be shared across different microarchitectures. Our experiment suggests that two different designs are used for our test set and we cluster the respective CPUs in Table 3 accordingly. Our results showed that the latest 29 or 66 branches determine the BHB value, depending on the microarchitecture.

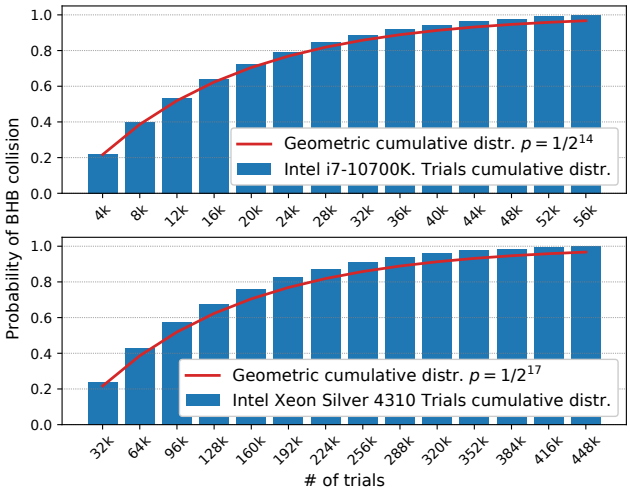

Figure 6: Cumulative distribution of the required trials to find a collision for a given context resulting in the same BHB value on an Intel i7-10700K (top) and an Intel Xeon Silver 4310 (bottom) over 10,000 individual trials. The results perfectly match geometric cumulative distribution to brute-force 14 and 17 bits of entropy.

History brute-forcing. In Section 5.2, we brute-force the BHB value in order to find BTB-tag collisions. However, this approach may be unfeasible for exploitation if the BTB tag is too large. For this reason, we repeated the experiment shown in Figure 4 multiple times to observe the distribution of trials needed to find a colliding history. We expect this experiment to follow a geometric distribution over the number of attempts, since all trials are independent, there are only two outcomes (collision/no-collision), and the probability of success is the same for every trial. Indeed, the results shown in Figure 6 follow a geometric distribution. For instance, on the i7-10700K, our experiment showed that the probability p of observing at least one collision is approximately 0.4 after 8,000 trials and 0.85 after 32,000 trials. We observed two cumulative distributions matching geometric distributions with respective success probability of $p = 1/2^{14}$ and $p = 1/2^{17}$ depending on the microarchitecture. This suggests that the brute-force space for these CPUs is exactly 14 and 17 bits, respectively. We refer to this brute-force space as the "BTB-tag entropy", as an attacker needs to overcome this entropy when attempting to forge a colliding BTB-tag value. The limited BTB-tag entropy and the reliable misprediction rates prove that brute-forcing is a viable option for an attacker seeking portable exploitation without having to reverse engineer the predictor internals each time.

> ***Observation* 3.** The BTB-tag entropy is small enough to allow brute-force approaches.

History controllabilty. Finally, another key parameter is the minimum number of branches an attacker needs to control

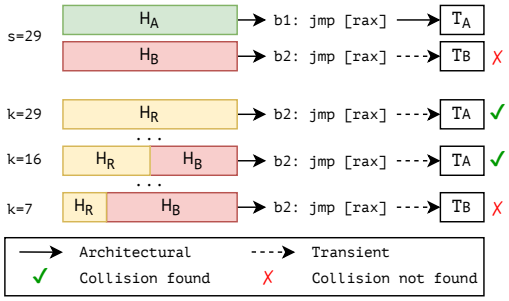

Figure 7: Minimum history control recovery for the Intel 10700K. Starting with two non-colliding histories (H_A and H_B), we aim to discover the minimum number of controlled branches in H_B to preserve collisions.

to generate arbitrary BTB tags. This is important for BHI attacks, as an attacker may only control a small portion of the BHB when exploiting indirect branches deeply nested in the kernel. To identify the minimum number of branches one needs to control, we used the experiment shown in Figure 7—based on the brute-force technique discussed in Section 5.2. The experiment starts by finding two non-colliding histories H_A and H_B of length equal to the BHB size(s). Next, we keep H_A constant and start generating random histories H_R until we find a collision—which will likely happen within a few thousand executions (Figure 6). Once we observe a collision, we fix the youngest branch (i.e., the closest to the indirect jump) to the latest branch of H_B ($H_B[s]$) and we randomize the previous $s-1$ branches in the jump chain. That is, the generated history will be the concatenation between $H_R[1:k]$ and $H_B[k:s]$, where k is the number of branches under the attacker's control—ranging from s to 1. We continue this process by fixing (i.e., losing control over) one more branch at every round—i.e., $k \rightarrow k-1$. The brute-forcing continues until a collision is found or when a timeout is reached (i.e., one million trials based on the results in Figure 6) for each value of k. When the attacker can randomize only few "old" branches, the chances of finding a colliding history decrease, causing the experiment to reach a timeout at every run. We show the results for a 24-hour run of this experiment in Figure 8. We observed reliable history collisions ($p \approx 1$) when controlling only the oldest 9 out of 29 or 8 out 66 branches (respectively), depending on the microarchitecture. For example, on a Xeon Silver 4310, an attacker needs to control only the 8 branches before a syscall to be able to mistrain an indirect branch as deep as 58 basic blocks after the kernel mode switch.

> ***Observation* 4.** The control of few branches is sufficient to generate colliding BTB tags.

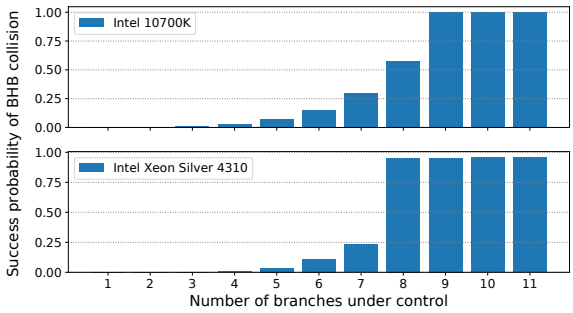

Figure 8: Probability of finding a colliding history to a given history H_A when controlling only the oldest k branches (x-axis). The plot is the result of a 24-hour run of the experiment described in Section 5.3.

6 End-to-end Exploitation

In this section, we showcase two end-to-end BTI exploits against the Linux kernel with eIBRS enabled. We first exploit our BHI primitive to leak data from the Linux kernel at 160 bytes/s. We then show how eBPF allows us to build even more complex kernel-to-kernel (i.e., generic intra-mode BTI) exploits. Finally, we conclude by discussing the implications of disabling unprivileged eBPF.

Attacker model. We consider unprivileged attackers who aim to disclose confidential information, such as private keys, passwords, or randomized pointers. We assume an x86-64 based victim machine running the latest microcode and operating system version, with all state-of-the-art mitigations against transient execution attacks enabled. We also consider a victim system with no exploitable vulnerabilities apart from the ones considered. Finally, we assume attackers can run (only) unprivileged code on the victim system, but seek to leak data across privilege boundaries.

Experimental setup. All the experiments in this section were performed on an Intel i7-10700K machine with eIBRS enabled and microcode revision 0xec, running Linux kernel version 5.14.10 (latest stable at the time of writing) without modifications and compiled with the corresponding default Ubuntu kernel config file (available with the rest of our code at https://vusec.net/projects/bhi-spectre-bhb).

6.1 Attacker Primitives

In order to craft an end-to-end BTI exploit against the kernel, we need to acquire the following primitives.

P1 – Victim branch. We need to find an indirect branch in the kernel we can mistrain. The victim branch needs to: (i) be triggered from userland; (ii) allow the attacker to control enough of the history to exploit BHI (see Section 5.3); and (iii) execute with attacker-controlled registers or stack values.

P2 – Disclosure gadget. We need a disclosure gadget to leak arbitrary data via a covert channel such as FLUSH + RELOAD (F+R) [61] or its eviction-based variant EVICT + RELOAD (E+R). We show a simple F+R transient gadget below.

```
reload[*secret_byte * 4096];
```

This gadget performs two dependent loads to leak arbitrary bytes by encoding them in a `reload` buffer—the `secret` is multiplied by 4096 to bypass the cacheline prefetchers [34]. However, the complexity of the gadget depends on the state of the registers when the victim branch is reached. Hence, more loads may be required. Depending on the attack primitive, the gadget must be between the valid targets of the vicitm branch (in-place BTI) or among any indirect branch target in the victim space (out-of-place BTI).

P3 – Covert channel. This primitive builds on top of the previous two. However, it introduces its own set of challenges. In fact, the disclosure gadget alone only leaves a microarchitectural trace in the cache but does not leak it to the attacker. Therefore, we need to build a valid channel across privilege boundaries (e.g., a shared F+R buffer [20, 34, 62]).

P4 – History brute-forcing. In Section 5.1, we rely on a synthetic covert channel to demonstrate how we can use BHI to build cross-privilege BTI attacks. However, as the end-to-end exploit targets an unmodified kernel, this option is no longer available. Since we do not know the inner workings of the BHB, we need a way to detect when our brute-forcing technique succeeds in finding two colliding histories. Therefore, we need another covert channel to leak this "bit" of information to an unprivileged attacker.

6.2 Exploiting BHI with eBPF

We now showcase an end-to-end exploit by leveraging Extended Berkeley Packet Filter (eBPF) to acquire all the aforementioned primitives.

P1 – Victim branch. In order to simplify the exploit, we looked for the "closest" indirect branch to the syscall entry point. In the Linux kernel, this is in the `do_syscall_64` function—only 4 taken branches away from the `syscall` instruction. This function uses an indirect branch to execute the correct syscall handler stored in the syscall jump table. Since the function is executed immediately after the user-to-kernel mode switch, the BHB value is almost entirely under user control—making BHI easier to exploit.

Listing 2 JIT'ted code for the eBPF program serving as disclosure gadget.

```
1   push   rbp
2   mov    rbp,rsp
3   ;load er_buf base address
4   movabs rsi,0xffffc900028ff110
5   ;rdi+0x18 = &pt_regs.r12 transiently
6   ;         = &bpf_sock architecturally
7   mov    rax,QWORD PTR [rdi+0x18]
8   test   rax,rax
9   je     fail
10  ;Dereference of user r12 value transiently
11  mov    eax,DWORD PTR [rax+0x14]
12  ;extract the byte to leak
13  and    rax,0xff
14  shl    rax,0xc
15  add    rsi,rax
16  ;maccess(er_buf[byte_to_leak*0x1000])
17  mov    rsi,QWORD PTR [rsi+0x0]
18 fail:
19  xor    eax,eax
20  leave
21  ret
```

Additionally, before calling `do_syscall_64`, all user registers are pushed onto the stack, and a pointer to them is passed as a first argument in `rdi`. In other words, the attacker controls a large portion of the stack when executing the disclosure gadget, greatly aiding exploitation.

P2 – Disclosure gadget. We rely on eBPF to make our attack independent of the kernel version and configuration. eBPF filters are user-developed programs that perform specific tasks, such as packet filtering, inside a kernel sandbox. On most Linux-based systems, eBPF programs are JIT-compiled for additional performance and unprivileged users have access to a restricted subset of their functionalities[2]. As a result, unprivileged attackers can "inject" code into the kernel, as long it conforms to the restrictions imposed by unprivileged eBPF, and they can then trigger their execution by simply writing to a socket. While eBPF JIT'ed code is hardened against traditional Spectre attacks (e.g., Spectre-v1 [34]), we can still build "transient type confusion" attacks. In fact, eBPF programs are executed from an indirect branch, making them ideal BTI gadgets.

For our purposes, we create an eBPF program that can leak data from arbitrary memory locations when executed transiently from our victim branch. We show the JIT'ed code of this disclosure gadget in Listing 2. Architecturally, this program operates on a `bpf_socket` struct pointer passed via `rdi`. However, when executed transiently from the `do_syscall_64` indirect jump, it operates on the stack-saved user registers referenced by `rdi`. We abuse this primitive to read arbitrary kernel addresses (stored in the user-saved `r12` register) and then transfer them to userland through an E+R buffer, one byte at the time.

P3 – Covert channel. To successfully build an E+R covert channel, we still rely on eBPF. We first allocate two `BPF_MAP_TYPE_ARRAY` eBPF buffers: one for *reload* stage and another one to perform the *eviction*.

To measure access times of the reload buffer, we create a second unprivileged eBPF program using the `bpf_ktime_get_ns` helper function, which provides enough timing resolution to discern between a cache hit and a miss. To propagate the measured timings, we use eBPF maps—key/value stores to exchange data between eBPF programs and userland. Finally, for the eviction stage, we simply access "enough" page-aligned addresses in the eviction buffer to evict the leaked entry from the LLC set (i.e., around $4K$ accesses for the tested machine) [59].

P4 – History brute-forcing. The last primitive to complete the exploit is a way to detect when BHI succeeded, i.e., when we craft a user-controlled history for the victim branch (i.e., `do_syscall_64`) that generates the same BTB tag of the eBPF gadget. eBPF simplifies this step as well, since its programs are always executed via the same indirect branch. Therefore, the user-crafted history can be reused to trigger transient execution of *any* eBPF program—as long as it was executed before the victim branch.

To find a history resulting in such a collision, we install a *one-bit* E+R gadget (i.e., it loads a single variable). This program allows us to determine whether the gadget was executed in the first place or not. We then brute-force different BHB values from userland, as shown in Listing 1. If the histories collide, our E+R gadget detects a hit. Thanks to the limited size of the BTB tag, only a few thousand trials—usually completing in less than a minute—are sufficient to find a BHB value that aliases to the target BTB entry (see Section 5.3).

Optimizations. While these primitives are sufficient to build a working exploit, the transient window is fairly small. As such, we observe a fairly weak signal. We improve the signal by evicting the cache line containing the `sys_call_table` entry of the mistrained syscall (e.g., `getpid`) before triggering the BHI-based misprediction. Doing so extends the transient execution window of the victim branch, greatly improving the reliability of the exploit. However, this step comes at the cost of an increased setup time, as we now need to programmatically find a matching eviction set. This can be achieved by starting with a large eviction set and trying different cache line offsets until we measure an increased execution time for the targeted syscall. After being sure that the correct cache line is evicted, the eviction set can be reduced using methods described in prior work [59].

[2]The corresponding build options are `CONFIG_HAVE_EBPF_JIT=y` and `CONFIG_BPF_UNPRIV_DEFAULT_OFF` not set. Default on major Linux distros, e.g., Debian 11, Ubuntu 20.04, and OpenSuse SLE 15.3 at the time of writing.

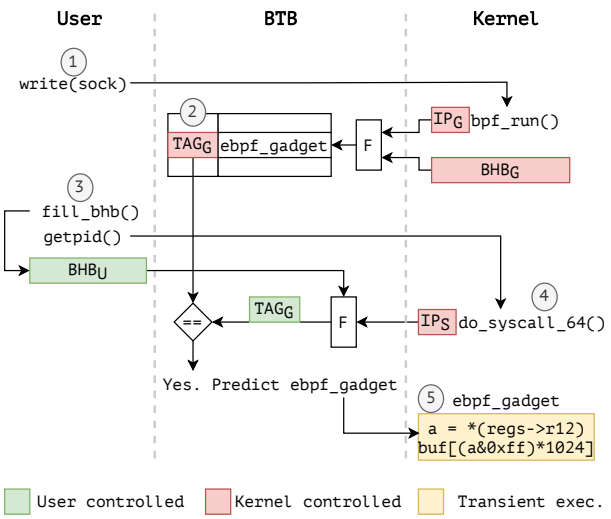

Figure 9: Visualization of our BHI exploit.

The full attack. With all these components in place, the attack can be summarized in 5 main steps (Figure 9):

① Initially, the attacker triggers the execution of the eBPF disclosure gadget by sending a packet to a socket with an eBPF packet filter.

② Since eBPF programs are executed from an indirect branch in `bpf_run()`, the execution of the eBPF gadget generates a BTB entry with tag TAG_G, based on the history BHB_G and the source address of the branch IP_G; i.e., $TAG_G = f(\,BHB_G\,, IP_G\,)$. This entry is kernel-tagged since the indirect branch is performed by the eBPF scheduler.

③ The attacker then fills the BHB from userland by executing the jump-chain they previously identified to cause a BHB collision right before performing a syscall.

④ Upon reaching the syscall entry point in `do_syscall_64`, the branch predictor tries to predict the indirect target of the correct syscall handler (e.g., `getpid`). Due to BHI, the attacker-controlled BHB value causes a BTB-tag collision with the eBPF-gadget entry, that is, $TAG_G = f(\,BHB_U\,, IP_S\,) = f(\,BHB_G\,, IP_G\,)$. As a result, the branch predictor will mispeculate and transiently execute the eBPF disclosure gadget instead of the syscall handler.

⑤ Finally, thanks to the "transient type confusion", the gadget in Listing 2 speculatively leaks the value referenced by the user-saved register `r12`. The leaked data can then be recovered using the E+R covert channel above (**P3**).

Evaluation. We evaluated our exploit by leaking 10 times 8 kB of contiguous kernel memory. Our proof-of-concept exploit requires an average setup time of 22 seconds to build an eviction set, and an average of 13 seconds to brute-force the BHB history. We observed a bandwidth of 160 bytes/s and an error rate of 1% on average.

6.3 Same-Predictor-Mode Exploit

Isolation-based mitigations, such as eIBRS and CSV2, were developed under the assumption that intra-mode exploit are impractical at best, if not impossible [4, 29]. In the previous sections, we unveiled fundamental implementation flaws of these mitigations (Section 5) and demonstrated how we can exploit BHI to build an end-to-end exploit against the kernel (Section 6.2). However, assuming a hypothetical implementation of these mitigations that inhibits cross-privilege BHI, would we still be able to mount attacks?

We now show that, even with completely isolated branch contexts, intra-mode (e.g., kernel-to-kernel) BTI exploits are not only possible, but even practical. Indeed, almost all the primitives we built in Section 6.2 remain untouched. We only move the BHB brute-forcing (**P4**) inside eBPF itself. However, differently from Section 6.2, we now exploit the BHB only to perform out-of-place BTI within the kernel, without performing BHI. To this end, we create an additional eBPF program that executes "enough" conditional branches to poison the BHB based on a value provided by the attacker via eBPF maps. At the end of this program, we use the eBPF `tail_call` helper function to append an indirect branch injecting a BTB entry for our disclosure gadget (**P2**). Then, we simply brute-force the BHB value during the setup of the exploit and, once we find a colliding history, we fix it for the rest of the execution. Hence, the general attack flow remains the same as the one shown in Figure 9, with the only difference being that we now generate a colliding BTB tag with the `fill_bhb` function inside the kernel when writing to a socket (step ①). We evaluated this generalized intra-mode BTI exploit and observed identical leak rate and setup time compared to the cross-privilege BHI variant shown in Section 6.2.

Implications. Intra-mode BTI exploits were previously deemed unfeasible. This assumption has motivated the design of mitigations such as eIBRS and CSV2. By demonstrating a working kernel-to-kernel BTI exploit, we show that adopting isolation-based solutions as full-fledged mitigations is too optimistic. In fact, thanks to user-accessible interpreters and JIT engines in the kernel such as unprivileged eBPF, an attacker can sufficiently control kernel execution to mount practical attacks. Furthermore, exploits may be feasible even in the absence of such features, as we discuss in the next section.

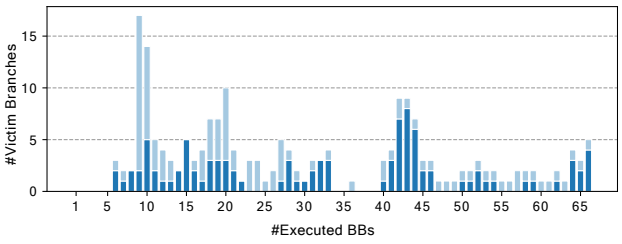

Figure 10: Victim branches vs. executed BBs. The stacked bins account for the branches with no user-controlled registers.

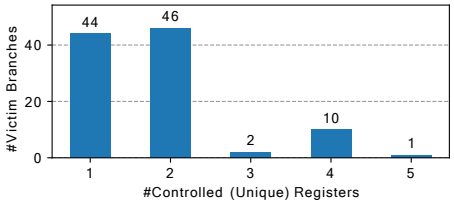

Figure 11: Victim branches vs. number of controlled registers. We account only for unique user-controlled values (i.e., two registers with the same value count as one).

6.4 Exploitation Beyond eBPF

In this section, we analyze the possibility of building attacks against the Linux kernel even with unprivileged eBPF disabled. In the absence of eBPF, an attacker mainly needs to identify a victim branch (**P1**) and a disclosure gadget (**P2**) to build a covert channel across privilege boundaries. In this section, we focus mainly on identifying exploitable victim branches in the kernel, as this is the most relevant step when exploiting BHI. We then present a preliminary analysis of indirect-branch targets in the kernel codebase.

6.4.1 Victim Branches

We built a tool on top of angr [47, 48, 50] to identify every kernel indirect branch reachable from the syscall entry point. When building intra-mode exploits, any indirect branch in the kernel may be a viable target for exploitation. However, when exploiting BHI, we can only abuse indirect branches close to the syscall entry point. Therefore, based on the results we presented in Section 5, we limited the depth of our search to 66 Basic Blocks (BBs). In total, we identified 202 reachable victim branches, 99 of which have user-controlled registers. In Figure 10, we show the number of victim branches over the number of previously executed BBs.

To better evaluate their exploitation capabilities, in Figure 11 we show the distribution of how many registers are under attacker control when reaching the indirect jumps. We can see that an attacker can often control at least two registers—minimum requirement to exploit a simple F+R

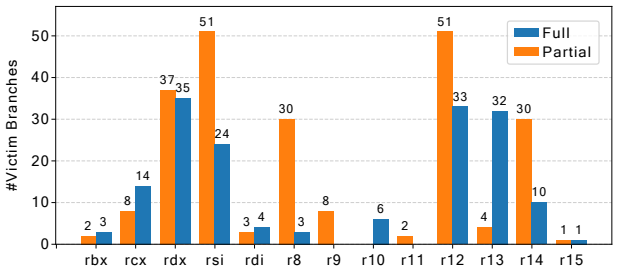

Figure 12: Controllability of each register when reaching the victim branches. One does not always control all the 64 bits. However, most of the time one lacks control only over one or two bits. Finally, as expected, one never controls `rax`, `rbp` and `rsp`.

gadget. We further show which registers are user-controlled for each victim branch in Figure 12. Having control over function-argument registers (as per the System V calling convention [38]), provides more options for an attacker. Indeed, since the main use case of indirect branches is executing function pointers, the function code will likely load data from the registers containing its arguments. Nonetheless, we observe that an attacker can often control registers used for function arguments (e.g., `rsi`, `rdx`, and `rcx`).

Moreover, while not all these branches are easily exploitable, some of them provide more control compared to what we exploited in Section 6.2. For instance, we identified an easily reachable indirect jump in `__x64_sys_prctl` with 9 controlled registers (with 5 unique user-controlled values)—only 7 basic blocks away from the syscall entry point.

Finally, while our analysis already reveals a nontrivial number of victim branches of interest, extending our results to the entire kernel to cover intra-mode BTI attacks beyond BHI would further increase these figures. And the larger the number of victim branches of interest, the higher the likelihood of fall-through targets (i.e., those following the indirect branch instruction in the binary) originating disclosure gadgets. Since such fall-through targets are transiently reachable through Straight-Line Speculation (SLS) on many Arm and x86-64 processors [14], our analysis also suggests that enabling SLS mitigations (e.g., available for both architectures in recent Linux kernel versions [36]) is of practical importance.

6.4.2 Indirect-Branch Targets

To find potential disclosure gadgets in the kernel, we first need all the *possible* kernel indirect-branch targets. In this section, we perform a preliminary attack-surface analysis for BHI and other intra-mode BTI attacks. We extracted an approximation of all the targets via static analysis. We implemented a LLVM compiler pass (using the `hasAddressTaken` function) to export all the function entry points used as indirect call targets. We identified 16715 possible targets for the same setup mentioned earlier in this section. While our analysis

does not consider switch-case indirect jump targets, it does at least cover all the kernel functions an attacker may be able to reach when exploiting intra-mode BTI.

Potential disclosure gadgets. We further reduced the number of targets to the ones containing a possible disclosure gadget (**P2**). We built a second tool based on angr to search for potential F+R gadgets. We define them as *potential disclosure gadgets* (i.e., not conclusively exploitable), since we rely on a loose definition of a F+R gadget: two tainted loads—one dependent on the other—originating from different registers (see the example in Section 6.1). We conservatively consider a transient execution window of maximum 30 instructions from the indirect-branch target. Therefore, angr stops its search after reaching this upper-bound limit. Overall, our tool identified 1177 potential F+R gadgets. While further analysis is required to study their practical exploitability, our results indicate that the possible attack surface is nontrivial even when unprivileged eBPF is not available. Moreover, our analysis is by no means exhaustive, only focusing on the most "convenient" disclosure gadgets. For instance, one may want to exploit the kernel with a more classic PRIME + PROBE covert channel [20], loosening the requirements for the disclosure gadget. Or one may consider control over the stack, unlocking more complex gadgets, such as the one used in Section 6.2.

7 Mitigations

To mitigate the limited isolation guarantees of eIBRS and CSV2, it is not sufficient to flush the BHB value on mode switches. Tagged-based isolation strategies are fundamentally limited by the presence of same-predictor-mode mispredictions. As demonstrated in Section 6.3, an attacker can still mount BTI attacks in a fully isolated system by using *privileged-to-privileged* mispredictions. In the absence of Simultaneous Multithreading (SMT), we verified that a complete flush of all previous BTB entries with IBPB on x86-64 can be used as an effective countermeasure. This strong barrier prevents unprivileged layers from interfering with higher-privilege predictor modes. However, this defense introduces a huge performance overhead since all privilege levels are affected by *every* flush.

A more cost-effective mitigation strategy is to selectively disable indirect branch prediction on privileged code. For example on most x86-64 systems, this can be achieved by re-enabling retpoline—known to be faster than other existing in-silicon defenses (i.e., IBRS [54]). Since all major OSes, compilers, etc. already support it, retpoline can be easily deployed on many systems. For hardware defenses, we believe that disabling indirect branch speculation for high-privilege predictor modes altogether—as already available in specific Arm microarchitectures (e.g., `CPUACTLR_EL1` [2])—is a viable option. Ultimately, this mitigation can provide similar security guarantees as retpoline, while avoiding RSB pollution. Note that, as our results suggest, it is also important to inhibit Straight-Line Speculation (SLS) [14] and eliminate potential fall-through disclosure gadgets, unless alternative compiler-based mitigations are in place [36].

One last hardening technique to complicate exploitation is to disable access to exploitation-friendly features such as the Linux kernel's eBPF for unprivileged users by default. As shown in our exploits, eBPF is the ideal playground to mount transient execution attacks due to the attacker-controlled code generation and the presence of precise timers. Controlling code executed in kernel space (or other privileged modes) should only be possible for administrators, since verifying the security of JIT'ed code against microarchiectural attacks is a nontrivial task.

Vendors' response. In response to our disclosure, Intel and Arm released whitepapers [3, 30] with suggested mitigations, while AMD confirmed they are not affected by BHI—matching our findings.

As a first line of defense, Intel suggests disabling unprivileged eBPF by default, a mitigation which was indeed deployed by the Linux developers [25]. To address more general BHI attacks, Intel recommends `lfencing` potential disclosure gadgets or adopting additional hardening options. In particular, Intel implemented dedicated indirect branch controls (`IA32_SPEC_CTRL` MSR) to disable higher-privilege BHB pollution, a feature available on some current and future processors. For other processors, software BHB-clearing sequences are available. To address intra-mode BTI attacks beyond BHI, additional indirect branch controls, where available, can disable user/supervisor indirect branch prediction altogether. Alternatively, Intel recommends re-enabling retpoline on processors where it is fully effective.

Regarding Arm, while we could not cause out-of-place mispredictions in our experiments on a limited sample of the Arm ecosystem, their extensive set of mitigations and the list of affected architectures [3], suggests BHI still represents a realistic threat on many Arm devices. Indeed, they proposed 5 different solutions depending on the microarchitecture of the device. A CPU can be protected against BHI via software during a mode switch by (1) a BHB-clearing sequence, (2) the new `clearbhb` instruction, or (3) a SMC to trigger the firmware BHB-clearing mitigation named "Workaround_3". Regarding in-silicon defenses, (4) "Exception Clears Branch History Buffer" will ensure that exception vectors are unaffected and additionally (5) CSV2.3 will be released to specify when a device is affected by BHI. We are not aware of planned Arm mitigations against intra-mode BTI attacks beyond BHI.

On a related note, AMD found flaws in their retpoline implementation during the disclosure process. AMD realized that the `lfence/jmp` sequence is racy and thus not safe (i.e., provably so on systems with SMT enabled) starting from the Zen microarchitecture (family 17h), reverting to generic

retpoline as a result. Similar behavior can be observed on modern Intel microarchitectures. Finally, the Linux kernel developers concurrently fixed the funneling issue described in Appendix A, by patching indirect jumps in-place instead of relying on the indirect thunks.

8 Related Work

Spectre [34] and Meltdown [37] set the stage for a large body of transient execution attacks [8, 20, 35, 39, 40, 43–45, 49, 55–58, 63]. In this work we focus on Branch Target Injection, also dubbed Spectre v2, and specifically focus on the security guarantees and limitations of related defenses. Throughout the paper, we show the incompleteness of isolation-based hardware mitigations alongside end-to-end Linux/Intel exploits against the kernel from unprivileged user applications.

Previous work [9, 11, 34, 62] already demonstrated out-of-place BTI attacks on Intel CPUs, either by abusing virtual address aliasing (i.e., same 32 LSBs) [9, 11], or by relying on the BHB [9, 34, 62]. In our work, we expand on these results demonstrating how, despite the introduction of dedicated hardware mitigations (i.e., eIBRS), the "leaky" isolation can lead to serious security threats. Furthermore, we expand the practicability of out-of-place BTI when controlling *only* the BHB, lowering the bar for BTI attacks in the general case.

Zhang et al. [64] implemented Spectre-v2 attacks abusing IP-address collisions and early front-end collisions in the context of transient trojans, i.e., software modules concealing their malicious activity. In our exploits, we only exercise BHB-based mispredictions against an unmodified kernel and exploit bugs in hardware defenses to leak data across privilege boundaries. We also consider a different threat model of an unprivileged attacker performing both user-to-kernel and kernel-to-kernel attacks, while Zhang et al. consider a privileged attacker performing kernel-to-user and kernel-to-kernel attacks based on injected trojans. Finally, while the feasibility of same-predictor-mode mispredictions has been demonstrated before, we are the first to show that this enables practical exploitation for an unprivileged attacker.

Schwarzl et al. [46] verified that the address translation attack [24] became viable again after switching defaults from retpoline to eIBRS in the Linux kernel on compatible systems. The root cause is the funneling issue described in Appendix A, which re-enables Spectre-v2 in-place attacks.

Lastly, many prior efforts have reverse engineered indirect branch prediction internals [19, 34, 62, 64]. Our work confirms their partial results, provides information on the most recent processor generations, and proposes a new approach based on history brute-forcing to mount BHB-based BTI attacks.

9 Conclusion

Despite the rise of in-silicon mitigations against transient execution attacks since the public disclosure of Spectre in 2018, we demonstrate that Branch Target Injection is still possible on modern processors deploying these mitigations. We analyzed indirect branch prediction internals and showed that BHB-based cross-privilege BTI attacks are not only *feasible*, but also *practical*. In more detail, we implemented end-to-end exploits breaking both partial, as well as full (hypothetical) indirect branch prediction state isolation on Intel systems, leaking secrets from the kernel at a rate of 160 bytes/s.

As countermeasures, we suggest re-enabling software defenses such as retpoline and forbidding unprivileged access to exploitation-friendly features such as eBPF to further reduce the transient execution attack surface.

Disclosure

We disclosed the security vulnerabilities described in the paper to Intel, AMD, Arm, and the Linux Kernel in Sep, 2021. Other affected software vendors have been notified by Intel. Following our reports, affected vendors have acknowledged our findings, developed mitigations, and released security advisories. Intel rewarded our findings with the Intel Bug Bounty program and issued two CVEs (CVE-2022-0001 for BHI and CVE-2022-0002 for Intra-mode BTI). Arm also issued a CVE (CVE-2022-23960). After initially proposing a 90-day embargo period, we later agreed to delay the public disclosure date to March 8, 2022, in order to provide vendors with sufficient time to implement and deploy mitigations.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. We also thank Alyssa Milburn and Andrew Cooper for their feedback. This work was supported by the EU's Horizon 2020 research and innovation programme under grant agreement No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by Netherlands Organisation for Scientific Research through projects "TROPICS", "Theseus", and "Intersect". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: restoring the performance of indirect branches in the era of Spectre. In *USENIX ATC*, 2019.

[2] Arm. CPUACTLR_EL1: ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3. https://developer.arm.com/documentation/100095/0003/way1382037666700. Accessed on 21-02-2022.

[3] Arm. Speculative processor vulnerability. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[4] ARM. Vulnerability of Speculative Processors. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, 2018.

[5] ARM. Speculative processor vulnerability FAQ. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/frequently-asked-questions, 2021.

[6] ARM. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability, 2021.

[7] Arm®. Architecture Reference Manual - Armv8, for A-profile architecture. DDI 0487G.b (ID072021), 2021.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM CCS*, 2019.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[10] Po-Yung Chang, Eric Hao, and Yale N Patt. Target prediction for indirect jumps. In *ACM SIGARCH Computer Architecture News*, 1997.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE EuroS&P*, 2019.

[12] Yu cheng Yu. Firmware interfaces for mitigating cache speculation vulnerabilities. https://developer.arm.com/-/media/developer/pdf/ARM_DEN_0070A_Firmware_interfaces_for_mitigating_CVE-2017-5715.pdf, 2021.

[13] Andrew Cooper. Mitigations for SP2/CVE-2017-5715/Branch Target Injection. https://lists.xenproject.org/archives/html/xen-devel/2018-01/threads.html#02169, 2018.

[14] Jonathan Corbet. Blocking straight-line speculation – eventually. https://lwn.net/Articles/877845, 2022.

[15] Karel Driesen and Urs Holzle. Accurate indirect branch prediction. In *ACM ISCA*, 1998.

[16] Karel Driesen and Urs Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *IEEE MICRO*, 1998.

[17] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Pibe: practical kernel control-flow hardening with profile-guided indirect branch elimination. In *ACM ASPLOS*, 2021.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE MICRO*, 2016.

[19] A. Fog. The Microarchitecture of Intel, AMD and VIA CPU. http://www.agner.org/optimize/microarchitecture.pdf, 2021.

[20] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *ACM CCS*, 2020.

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

[22] Julien Grall. xen/arm64: Add skeleton to harden the branch predictor aliasing attacks. https://github.com/xen-project/xen/commit/4c4fddc166cf528aca49540bcc9ee4f196b01dac#diff-6092d8e9c6cb4df6f4c9b08626778f9dbac0e8b7209445276498639a21398039R97, 2018.

[23] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, et al. Evolution of the samsung exynos cpu microarchitecture. In *ACM ISCA*, 2020.

[24] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *ACM CCS*, 2016.

[25] Pawan Gupta. Disallow unprivileged bpf by default. https://lore.kernel.org/bpf/20211027233943.kehyrdbibp2d2u4c@gupta-dev2.localdomain/T/, 2021.

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

[27] Intel. INTEL-SA-00088. https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/advisory-guidance/branch-target-injection.html, 2018.

[28] Intel. Retpoline: A Branch Target Injection Mitigation. https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

[29] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual combined volumes. 325462-070US, 2019.

[30] Intel. Branch History Injection advisory. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-history-injection.html, 2022.

[31] Mehmet Iyigun. Mitigating Spectre variant 2 with Retpoline on Windows. https://web.archive.org/web/20211126075707/https://techcommunity.microsoft.com/t5/windows-kernel-internals-blog/mitigating-spectre-variant-2-with-retpoline-on-windows/ba-p/295618, 2019.

[32] John Kalamatianos and David R Kaeli. Predicting indirect branches via data compression. In *IEEE MICRO*, 1998.

[33] Hyesoon Kim, José A Joao, Onur Mutlu, Chang Joo Lee, Yale N Patt, and Robert Cohn. Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. In *IEEE Transactions on Computers*, 2008.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, 2019.

[35] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX WOOT*, 2018.

[36] Michael Larabel. x86 straight line speculation cpu mitigation appears for linux 5.17. https://www.phoronix.com/scan.php?page=news_item&px=x86-SLS-Mitigation-5.17, 2022.

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

[38] Hongjiu Lu, Michael Matz, Milind Girkar, Jan Hubiaka, Andreas Jaeger, and Mark Mitchell. System v application binary interface amd64 architecture processor supplement (with lp64 and ilp32 programming models) version 1.0. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf, 2018.

[39] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM CCS*, 2018.

[40] Ken Johnson Microsoft Security Response Center (MSRC). Analysis and mitigation of speculative store bypass. https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-%store-bypass-cve-2018-3639/, 2019.

[41] Phronix. Amd retpoline benchmarks from fx to threadripper & epyc. https://www.phoronix.com/scan.php?page=news_item&px=AMD-Retpoline-Linux-4.15-FX-Zen, 2018.

[42] Sai Praneeth. [PATCH] x86/speculation: Support Enhanced IBRS on future CPUs. http://lkml.iu.edu/hypermail/linux/kernel/1807.3/00923.html, 2018.

[43] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[44] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE S&P*, 2021.

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

[46] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing: Reviving foreshadow. In *International Conference on Financial Cryptography and Data Security*, 2021.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, 2015.

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*, 2016.

[49] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[50] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, 2016.

[51] AMD64 Technology. Indirect Branch Control Extension - Revision 4.10.18". https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf. Accessed on 21-02-2022.

[52] AMD64 Technology. Software techniques for managing speculation on amd processors - Revision 7.10.18. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf. Accessed on 21-02-2022.

[53] AMD64 Technology. AMD64 Architecture Programmer's Manual: Volumes 1-5s - Revision 4.03, 2021.

[54] Linus Torvalds. Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lkml.org/lkml/2018/1/21/192, 2018.

[55] 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 *USENIX Security*, 2018.

[56] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *IEEE S&P*, 2020.

[57] 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 *IEEE S&P*, 2019.

[58] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *IEEE S&P*, 2021.

[59] Pepe Vila, Boris Köpf, and José F Morales. Theory and Practice of Finding Eviction Sets. In *IEEE S&P*, 2019.

[60] David Woodhouse. [PATCH v8 00/12] Retpoline: Avoid speculative indirect calls in kernel. https://lore.kernel.org/all/1515707194-20531-5-git-send-email-dwmw@amazon.co.uk/T/#u, 2018.

[61] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[62] Jann Horn Google Project Zero. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[63] Jann Horn Google Project Zero. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2019.

[64] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ACM ASPLOS*, 2020.

[65] Peter Zijlstra. objtool,x86: Rewrite retpoline thunk calls. https://lore.kernel.org/all/20210219220158.GD59023@worktop.programming.kicks-ass.net/T/#m18b6cfece23f71b4526106b93ec7bb4aeeb86df2, 2021.

[66] Marc Zyngier. KVM: arm64: Set CSV2 for guests on hardware unaffected by Spectre-v2. https://github.com/torvalds/linux/commit/e1026237f90677fd5a454f63057a62f984c2188d, 2020.

A Linux eIBRS Adoption Flaw

In Section 5, we discussed the BHI attack surface and the differences between in-place and out-of-place BTI [9]. However, during our security analysis, we discovered an issue with the adoption of eIBRS in the Linux kernel (versions < 5.14) that made the two scenarios almost equivalent. In fact, on eIBRS-supporting systems Linux used to "funnel" all the kernel indirect jumps to one single call site per register—essentially originating the same attack surface as out-of-place BTI since all the indirect jumps converge to few locations.

Such implementation was motivated by the fact that the Linux kernel does not know at compile time whether it will be running on a system that supports eIBRS. Indeed, since `cpuid` holds this information, the kernel discovers only at boot time whether it should use the in-silicon mitigation or the software-based retpoline. Thus, the Linux kernel developers opted for a hot-patching approach to gain the performance improvements of the in-silicon mitigation. Unfortunately, prior to version 5.14, the kernel used to patch a single thunk per register (Listing 3) for both environments. This caused no drawbacks on systems deploying retpoline, since retpoline does not allow the CPU to speculate on the target. However, on eIBRS-enabled systems, this resulted into `__x86_indirect_thunk_$reg` being patched to a simple `jmp $reg` at runtime. Doing so undermined the desired security guarantees of eIBRS (as well as indirect branch prediction performance), as all the branches in the kernel previously protected by retpoline are dispatched from the same few call sites. Effectively, this "funneling" allowed for straightforward in-kernel mistraining of the indirect branch predictor, since all indirect jumps are sunk to few source addresses, increasing the attack surface for Spectre-v2 (in-place BTI) attacks.

We disclosed this issue to the Linux kernel developers, who had observed it concurrently to our work and, in response, developed a patch set. As a result, as of Linux kernel 5.14 [65], indirect thunks are only used in the early boot process of the system, while for all the other indirect jumps the kernel performs patching directly "over" the original indirect jumps to completely eradicate this issue.

Listing 3 Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
do_syscall_64:
    ;...
    mov     rax, [sys_call_table + rax*8]
    call    __x86_indirect_thunk_rax
```

```
;with eIBRS support
__x86_indirect_thunk_rax:
    jmp     rax
```

```
;without eIBRS support (retpoline)
__x86_indirect_thunk_rax:
    call    B
A:  pause
    lfence
    jmp     A
B:  mov     [rsp], rax
    ret
```