Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks

Sander Wiebing
Vrije Universiteit Amsterdam
s.j.wiebing@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
c.giuffrida@vu.nl

***Abstract*—Spectre-v2 vulnerabilities have been increasingly gaining momentum, as they enable particularly powerful *cross-domain* transient execution attacks. Attackers can train the indirect branch predictor in one protection domain (e.g., user process) in order to speculatively hijack control flow and disclose data in a victim domain (e.g., kernel). In response to these attacks, vendors have deployed increasingly strong domain isolation techniques (e.g., eIBRS and IBPB) to prevent the predictor in one domain from being influenced by another domain's execution. While recent attacks such as BHI and Post-barrier Spectre have evidenced (now patched) implementation flaws of such techniques, the common assumption is that, barring implementation issues, domain isolation can close the attack surface in the practical cases of interest.**

In this paper, we challenge this assumption and show that even *perfect* domain isolation is insufficient to deter practical attacks. To this end, we systematically analyze *self-training* Spectre-v2 attacks, where *both* training and speculative control-flow hijacking occur in the same (victim) domain. While self-training attacks are believed to be limited to the *in-domain* scenario—where attackers can run arbitrary code and inject their own disclosure gadgets in a (default-off) sandbox such as eBPF—our analysis shows cross-domain variants are possible in practice. Specifically, we describe three new classes of attacks against the Linux kernel and present two end-to-end exploits that leak kernel memory on recent Intel CPUs at up to 17 KB/sec. During our investigation, we also stumbled upon two Intel issues which completely break (user, guest, and hypervisor) isolation and re-enable classic Spectre-v2 attacks.

1. Introduction

Spectre-v2 attacks [1], [2], [3], [4], originally demonstrated in 2018 [1], allow attackers to speculatively hijack an indirect branch in a victim protection domain (e.g., kernel) and redirect it to a *disclosure gadget*. The latter transmits secret data back to the attacker over a covert channel such as FLUSH+RELOAD [5]. Both *in-domain* and *cross-domain* Spectre-v2 attacks are possible [6]. In the former scenario, attackers can run arbitrary code in a sandbox environment (e.g., eBPF) and speculatively execute their own injected gadgets. In the latter scenario, attackers can only run code in their own domain (e.g., user process) to train the indirect branch predictor and then lure the victim domain into speculatively executing a disclosure gadget in its own code.

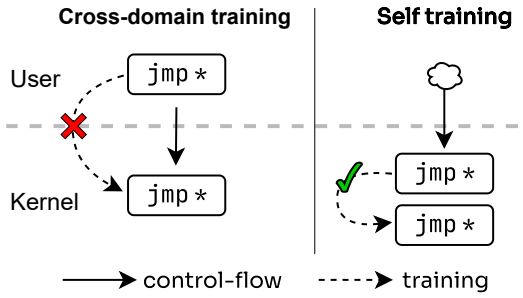


Figure 1: Cross-domain training versus self training. With self training, an attacker (e.g., an unprivileged user) can lure the victim (e.g., a kernel) to train itself, effectively bypassing even *perfect* domain isolation implementations.

While clearly more constrained, cross-domain attacks are much more powerful and have received much attention in recent research [2], [3], [4]. This is due to their wide applicability across arbitrary security boundaries and our inability to mitigate them by simply forbidding untrusted sandbox execution (e.g., disallowing privileged eBPF [4]).

In response, hardware vendors have deployed increasingly sophisticated mitigations based on *domain isolation*, either by means of explicit indirect branch prediction barriers (e.g., IBPB) or implicit predictor mode separation (e.g., eIBRS) [7]. The key assumption behind these mitigations is that, while in-domain attackers can run their own code to directly train the predictor in the victim domain, such *self-training* capabilities are not available to cross-domain attackers, who can only run arbitrary code for training purposes in their own domain. And that, due to domain isolation, training in one domain cannot control indirect branch prediction in another domain—barring implementation flaws that lead to *imperfect* isolation of the branch history [2], [4] or other indirect branch prediction structures [8].

In this paper, we show that this assumption is flawed and that even a *perfect* implementation of domain isolation is insufficient to deter practical attacks. More specifically, we demonstrate that *cross-domain self-training* Spectre-v2 attacks (Figure 1) are feasible in practice for the very first time. To this end, we conduct a systematic attack surface

analysis of these attacks, focusing on the user/kernel interface and the Linux kernel in particular—a prime target of state-of-the-art attacks [1], [2], [3], [4], [8].

We start our analysis by reverse engineering the behavior of state-of-the-art mitigations and of the indirect branch predictor in the context of self-training Spectre-v2 attacks. Our investigation reveals several new insights across microarchitectures, including details on the undocumented behavior of recent mitigations such as BHI_DIS_S and BHI_NO as well as different training strategies. Based on such strategies, we describe three self-training Spectre-v2 attack classes affecting recent Intel CPUs—although some classes also affect ARM, as detailed later.

The first class of attacks exploits *branch history collisions*, similar in spirit to BHI [2] but with a cross-domain attacker exploiting self-training and triggering branch history collisions exclusively in the victim domain for the first time. To exploit this class, the attacker must find a code path in the victim that gathers a desired branch history before speculatively executing the victim indirect branch. For practical exploitation, we show the attacker can abuse Linux kernel filters (widely used by SECCOMP [9] and socket filtering [10]) to lure the kernel into executing a sequence of direct branches and building up the desired history. To demonstrate the viability of this approach, we present an end-to-end exploit leaking arbitrary Linux kernel memory on last-generation (Lunar Lake) Intel CPUs at 1.7 KB/sec.

The second class of attacks exploits *IP-based branch collisions* in a history-agnostic indirect branch prediction scenario. To exploit this class, the attacker must first reliably disable the history-based kernel indirect branch predictor from userland. We describe a number of new techniques for this purpose, including one portable across microarchitectures and mitigations. We also show collisions are scarce, yielding a smaller attack surface than the other classes. Nonetheless, we present evidence of potentially exploitable (898) gadgets in large-scale attack scenarios.

The third class of attacks exploits *direct-to-indirect branch collisions*. We show these collisions occur in a history-agnostic fashion and provide a full speculation window on recent Intel CPUs, increasing the attack surface of IP-based branch collisions. To exploit this class, the attacker must find a direct branch in the victim colliding with the victim indirect branch. We discovered two variants for this class on recent Intel CPUs. The most severe variant has now been branded as *Indirect Target Selection* or *ITS* by Intel. For ease of exploitation, we show attackers can again abuse Linux filters and present an end-to-end ITS exploit leaking arbitrary Linux kernel memory on 10-11th-generation Intel CPUs at 17 KB/sec. We also present evidence of exploitation without Linux filters by means of gadget analysis.

Finally, we examine the broader implications of our findings, and in particular of our third class of attacks. Since collisions can occur between direct and indirect branches, domain isolation techniques targeting only indirect branches are insufficient. We experimentally confirm that indirect branch prediction barriers such as IBPB are ineffective for both our two variants, breaking isolation guarantees and re-enabling traditional user-to-user or guest-to-guest Spectre-v2 attacks (normally addressed by IBPB). We also experimentally verified other mitigations guarding the user/kernel interface such as eIBRS are not affected.

However, after disclosing our findings to vendors—which responded by developing several industry-wide mitigations, including microcode updates, new instructions, and OS/hypervisor changes—Intel eventually evidenced that ITS *does* break eIBRS isolation for the guest/host interface on certain microarchitectures. We experimentally confirmed this behavior, with a traditional cross-training Spectre-v2 proof of concept (PoC) leaking hypervisor memory on 10th-generation Intel CPUs at 8.5 KB/sec.

Contributions. To summarize our contributions:

1) We systematically analyze the attack surface of self-training Spectre-V2 attacks by means of reverse engineering, gadget analysis, and exploit development.
2) We describe three classes of self-training attacks, study their exploitability, and present two end-to-end exploits based on Linux kernel filters. All our artifacts are available at https://github.com/vusec/training-solo, including our test suite used for reverse engineering.
3) As a by-product of our investigation, we uncover two new hardware issues on Intel CPUs—i.e., CVE-2024-28956 (ITS) and CVE-2025-24495. Both issues completely break user and guest isolation and one (ITS) also breaks hypervisor isolation, collectively re-enabling classic cross-training Spectre-v2 attacks across a variety of security boundaries.
4) We discuss mitigations, some of which have been picked up by affected vendors.

2. Background

Branch Prediction. Branch prediction, performed by the Branch Prediction Unit (BPU), is a crucial CPU optimization. By predicting and speculatively executing the next instruction block, the CPU mitigates memory latency impact. Although direct branch targets are encoded in the instruction, the CPU stores executed targets in the Branch Target Buffer (BTB), indexed by the instruction pointer (IP) [11], [12], [13], [14]. The CPU uses this IP-based prediction to fetch the next block even before decoding completes, although speculative execution is typically prevented until the target is validated.

This differs for indirect branches, where the target is unknown at decode time and speculative execution is allowed before the branch operand's value is available. On Intel and AMD CPUs, indirect branches also use the BTB for IP-based prediction, but additionally use an Indirect Branch Target Buffer (iBTB) indexed by the Branch History Buffer (BHB) for history-based prediction [11], [12], [15]. On Intel, the BHB is a shift register gathering specific bits of the source and target address of a taken branch [11], [16], [17].

Spectre v2. Spectre-v2 attacks train the Indirect Branch

Table 1: Domain isolation techniques and indirect branch prediction strategies on analyzed microarchitectures.

Vendor	Model	Code name	μArch	*Isolation technique*							*Prediction*	
				BHB Clr.	IBPB	IBRS	a/eIBRS	BHI_DIS_S	IPRED_DIS_S	BHI_NO	BTB	iBTB
Intel	9900K	Coffee Lake R	Coffee Lake	○	◐	●	–	–	–	–	✗	✗
	10700K	Comet Lake	Comet Lake	◒	◐	○	●	–	–	–	✓	✓
	11700	Rocket Lake	Cypress Cove†	◒	◐	○	●	–	–	–	✓	✓
	11800H	Tiger Lake	Willow Cove†	◒	◐	○	●	–	–	–	✓	✓
	14900K	Raptor Lake	Raptor Cove‡	–	◐	○	●	●	○	–	✓	✗
			Gracemont§	–	◐	○	●	●	○	–	✗	✗
	155H	Meteor Lake	Redwood Cove‡	–	◐	○	●	●	○	–	✓	✗
			Crestmont§	–	◐	○	●	●	○	–	✓	✗
	258V	Lunar Lake	Lion Cove	–	◐	○	●	○	○	●	✓	◔
			Skymont§	–	◐	○	●	○	○	●	✓	◔
AMD	7950X	Raphael	Zen 4	–	◐	○	●	–	–	–	✗	✗
	9950X	Granite Ridge	Zen 5	–	◐	○	●	–	–	–	✗	✗

● Enabled by default, ◐ Enabled for cross-context, ◒ Enabled for cross-privileged, ○ Available, – Not available, ◔ Prediction after x privileged-branches, † Based on Sunny Cove, ‡ Based on Golden Cove, § Atom μarch.

Predictor (IBP) to predict an attacker-chosen *target* of an indirect branch. This is to speculatively execute transient instructions in a victim domain and transmit secrets back to the attacker domain via a covert channel [5].

To mitigate the most serious (i.e., cross-domain) Spectre-v2 attacks, hardware vendors have deployed a variety of mitigations based on *domain isolation*, which seeks to prevent training in one domain from affecting indirect branch prediction in another domain. Some mitigations such as the Indirect Branch Prediction Barrier (IBPB) provide software with an explicit barrier to flush indirect branch prediction entries [18]. This is typically done upon context switch to address cross-process and cross-VM attacks. Other mitigations such as Intel enhanced Indirect Branch Restricted Speculation (eIBRS) [18], ARM CSV2 [19], and AMD Automatic IBRS (AutoIBRS) [20], in turn, offer implicit predictor mode separation, preventing the predicted targets of indirect branches from being controlled by code running in a lower privileged mode or on another logical processor—addressing cross-SMT, user/kernel, and guest/host attacks.

These mitigations have not been free of implementation flaws. For instance, recent research shows Intel IBPB implementations fail to flush certain prediction entries, enabling post-barrier Spectre attacks [8]. As a fix, Intel released a microcode update. Branch History Injection (BHI) [2], [4], in turn, demonstrated that Intel eIBRS and ARM CSV2 do not isolate the branch history across predictor modes, enabling attacks based on cross-domain branch history collisions. As a fix, vendors originally suggested explicit software- (Intel) or hardware-based (ARM) barriers for the branch history. On recent Intel CPUs starting with Alder Lake and Sapphire Rapids, Intel has also released more efficient mitigations such as BHI_DIS_S and BHI_NO, which seek to prevent indirect branches in a privileged mode from being predicted based on an unprivileged branch history [7].

Linux Kernel Filters. The Berkeley Packet Filter (BPF) [21] was initially designed to efficiently filter packets in the kernel. As part of its filtering framework, the Linux kernel implements two variants: classic BPF (cBPF) and its more advanced counterpart, extended BPF (eBPF) [22].

cBPF is used for both network filtering via Linux Socket Filtering (LSF) [10] and syscall filtering via SECure COMPuting with filters (SECCOMP) [9]. Both have seen widespread adoption in production (e.g., in Snort, Docker, Chrome). cBPF supports only simple filters to inspect packet fields or syscall arguments—limited to two 32-bit internal registers, forward jumps, and no register dereferences [22].

Extended BPF (eBPF), in turn, provides a much more powerful sandbox environment, with 10 internal 64-bit registers, access to kernel helper functions, and management of key/value stores (*maps*) to persist data across syscalls. The extended functionality allows user-level attackers to run their own complex code in the kernel with crafted indirect branches, disclosure gadgets, and self-training primitives. Not surprisingly, eBPF has been previously abused to mount in-domain Spectre-v2 exploits and disabled for unprivileged users in response [2]. Outside the in-domain scenario, (cross-domain) self-training Spectre-v2 attacks have only been previously theorized [23] but never shown in practice.

Recent work suggests hardening eBPF programs against Spectre to re-enable eBPF for unprivileged users [24], [25].

3. Threat Model

We consider a standard cross-domain Spectre-v2 threat model, where an unprivileged attacker seeks to leak memory across security boundaries. We specifically focus on (*self-training*) attacks against the Linux kernel. We assume all the Spectre-v2 mitigations (e.g., eIBRS, BHI_NO) to be enabled. We also assume such mitigations have no implementation flaws and provide *perfect* domain isolation. Finally, we assume other vulnerabilities (e.g., memory errors) are mitigated through appropriate defenses.

4. Analysis of Domain Isolation Techniques

We start with an analysis and reverse engineering of domain isolation techniques through the lens of cross-domain self-training Spectre-v2 attacks. We focus on x86 mitigations for the user-kernel interface and later generalize.

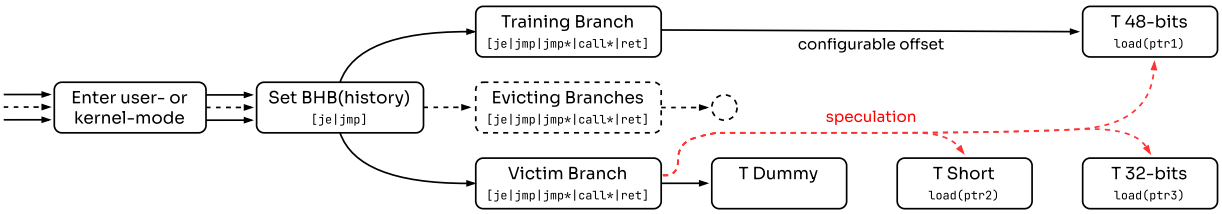


Figure 2: Workflow followed by our analysis test suite.

Details on our tested microarchitectures are available in Appendix A. We discuss our results for the microarchitectures in Table 1.

Intel (e)IBRS. Intel IBRS and eIBRS ensure user/kernel predictor mode separation. To analyze their behavior in a self-training scenario, we rely on a kernel module with a single indirect branch and a static branch history. We self-train the branch to jump to a target accessing a FLUSH+RELOAD buffer, then jump to a dummy target.

On Coffee lake, which enables IBRS by default, our experiments reveal no misprediction to the trained target. This indicates that indirect branch prediction is disabled altogether in kernel mode. On all the other microarchitectures, which support eIBRS, our experiments do reveal mispredictions to the trained target in all cases.

AMD Auto IBRS. AMD Auto IBRS is widely perceived as the AMD counterpart of Intel's eIBRS, preventing the use of branch targets trained in a lower privilege mode. However, repeating our eIBRS reverse engineering experiments on AMD Zen4 and Zen5 microarchitectures reveals a completely different behavior for Auto IBRS. Surprisingly, on both microarchitectures, we only observe prefetching of the predicted target, evidencing that AutoIBRS disables speculative execution for kernel indirect branches altogether.

Intel BHI_DIS_S. Intel CPUs from Alder Lake and Sapphire Rapids onwards support the BHI_DIS_S mitigation, which, according to Intel, prevents indirect branch targets in privileged modes (CPL0-2) from being selected based on a user (CPL3) branch history [7]. To understand the implications for self-training attacks, we change our kernel module to dynamically generate a branch history and check for correlation between the history and the predicted target. Surprisingly, our results reveal that the predictor fails to capture this correlation, entirely ignoring the branch history and only relying on IP-based prediction on the tested microarchitectures with BHI_DIS_S enabled.

BHI_NO. Recent Intel CPUs enumerate BHI_NO, which, according to Intel, should simply result in the behavior of "BHI_DIS_S being enabled by default" [7]. However, repeating our BHI_DIS_S reverse engineering experiments on recent Lunar Lake CPUs reveals a different behavior for BHI_NO. Specifically, our results show BHI_NO *does* allow history-based prediction for the kernel. However, the prediction depends on the number of preceding kernel branches.

Other mitigations. There are other x86 Spectre-v2 mitigations that are less relevant for our target scenario. Some, such as LFENCE;JMP, CET-IBT, and FineIBT are prone to race conditions [4], [26] and generally not deployed by default. Other software-based mitigations, such as retpoline [18] and BHB clearing [7], introduce nontrivial overhead (and sometimes other security issues [3], [7]) and have been replaced by more modern mitigations. Other hardware-based mitigations such as IBPB [18] are also costly when enabled at the user/kernel boundary and are thus only used to protect user/user or guest/guest boundaries by default. Yet others, such as IPRED_DIS_S disable indirect branch prediction altogether and are disabled by default [7]. As part of our reverse engineering efforts, we also considered these mitigations but did not found any unexpected behavior in the context of self-training Spectre-v2 attacks.

Summary. We derive a number of new insights from our analysis. Some CPUs such as Intel Coffee lake and last-generation AMD CPUs with AutoIBRS completely disable speculative execution for kernel indirect branches. As such, we exclude these CPUs from further analysis. The other CPUs in Table 1 may or may not use the branch history for indirect branch prediction depending on the microarchitecture and the mitigations. We use these insights to further investigate the predictor's behavior and analyze the attack surface of self-training attacks in the next sections.

5. Training Solo

With the current landscape of deployed domain isolation techniques, an attacker can pursue two possible strategies to mount kernel self-training attacks: (i) train indirect branches using in-kernel history and (ii) train indirect branches using IP-based kernel branch collisions. To study the behavior of history-based predictions, IP-based predictions, and the interplay between the two prediction strategies, we designed a test suite, which follows the workflow shown in Figure 2.

The core of the test suite consists of a *training branch* and a *victim branch*, whose IPs can be both randomized across the 48-bit address space at run time. The training branch is trained with a chosen target T. The victim branch jumps to a fixed dummy target. Both branches support different types and jump to a configurable offset. If the victim branch type is configured as an indirect branch, the address storing the dummy target is flushed from memory to extend the speculation window.

As shown by prior work [8], [16], [27], branch predictors may store different target lengths. To capture speculation to different target lengths, the test suite monitors, in addition to the full (48-bit) target *T*, its matching 32-bit target, and its matching short target (of configurable length). The test suite can be configured to use a classic FLUSH+RELOAD covert channel [1], loading a unique pointer at every monitored target, or a prefetcher-based covert channel [27] by testing if the target code is loaded into cache.

Besides the two core branches, multiple *evicting branches* can be configured to execute before or after the training branch. Finally, each *branch path* can be configured to execute in user/kernel mode, with a custom history and history branch type. Our test suite allows us to perform a systematic analysis across microarchitectures. All configuration options can be specified via a single test-case file per experiment, and the test-case files used in this section are publicly available. We present the results and implications of our analysis in the next sections. When not otherwise specified, the training and victim branch are configured with matching [39:0] IP bits and executed in user mode.

5.1. History-based Training

We start by studying in-kernel history-based collisions in absence of BHI mitigations and for the same indirect branch type. We repeatedly run our test suite by randomizing the training/victim branch histories and check for evidence of collisions, i.e., speculation to our chosen target *T*. We also run experiments with randomly different lower IP bits for the training/victim branch. Our results in both scenarios evidence in-kernel history collisions on Intel CPUs, confirming the results of prior cross-privileged analysis [7]. As an aside, we instead found collisions on AMD CPUs with AutoIBRS=off only with matching IP bits.

Collisions across indirect branch types. To better qualify the possible collisions pairs, we configure the test suite to test all the combinations of training/victim indirect branch types while randomizing both histories. Our results reveal that the type of the branch (jump/call/ret) is shifted into the iBTB tag. Namely, two branches with different branch type do not collide if both histories are equal, but randomizing the history does yield a collision.

BHI_DIS_S. Next, we analyze the impact of the BHI_DIS_S mitigation in more details. We first repeat our earlier experiments on Golden Cove with BHI_DIS_S enabled. Our results show that the CPU consistently mispredicts to the 32-bit target regardless of the particular branch history. This suggests the prediction is served from the BTB via the IP-based predictor, always predicting the last executed target matching the necessary IP aliasing requirements—detailed in Section 5.3. Repeating our analysis on Crestmont shows the BTB can store both 32-bit and 48-bit targets—if the source and target IP bits [47:32] differ, the BTB stores the full target. On Gracemont, BHI_DIS_S appear to disable speculative execution altogether.

BHI_NO. Next, we repeat our experiments on Lion Cove (P-core of Lunar Lake), which enumerates BHI_NO. We again observe IP-based predictions to the 32-bit target, but only if we execute a limited number of kernel branches before the victim branch. Specifically, we observe that the CPU switches to history-based prediction only if execute at least 129 preceding kernel branches. This appears to be a crude way to ensure the kernel does not use a (potentially malicious) user history, i.e., enabling history-based prediction only when the user history is shifted out of the BHB.

To confirm this intuition, we examine the size of the branch history (i.e., BHB) used by the predictor. To this end, we use a deeper victim branch history and randomize the location of each preceding branch one by one until the predictor can distinguish between the two paths. Our results show that the predictor uses a history of up to the last 66 branches. Moreover, with 194 or more preceding kernel branches, the branch history used by the predictor increases to 194 branches. On Skymont (E-core of Lunar Lake), our experiments reveal similar behavior but with different BHB sizes. After the 13 kernel branches executed by the syscall prologue, history-based prediction is already enabled, capturing up to the last 9 branches. After executing 34 kernel branches, the branch history size increases to 34.

Userland impact. Lastly, we test the impact of the BHI mitigations on userland. We configure our test suite to use in-user training and victim branches with distinct branch histories and enter/exit kernel mode before executing the victim branch. Our results show that the predictor can still differentiate between the two user branch paths, suggesting the BHB is not flushed on kernel entry. Next, we test if user-mode prediction is affected by the branch history gathered during kernel execution. To this end, we configure our test suite to use identical branch histories for the in-user training and victim branches and optionally execute a kernel branch before the victim branch. Our results show that the predictor successfully captures the correlation with the execution of the kernel branch, showing that the kernel branch history is still used for user-mode prediction (but not viceversa).

Summary. A self-training attacker can exploit arbitrary history-based collisions across matching kernel indirect branch types. However, BHI_NO / BHI_DIS_S disables history-based prediction for early / all branches, resorting to IP-based prediction except on Gracemont.

5.2. Predictor Selection

In the previous section, we detailed different factors that cause the CPU to use history- vs. IP-based indirect branch prediction. In some configurations (e.g., BHI_DIS_S enabled), the attacker can only exploit IP-based predictions. In others (e.g., BHI_NO enabled), the attacker can target specific kernel indirect branches to force the CPU to use IP-based (or history-based) prediction. We now want to assess

Table 2: Update policy for the BTB and iBTB prediction structures after executing an indirect branch.

BTB	iBTB	BTB updated	iBTB updated
Miss/Incorrect	Miss	✓	✓/✗†
Miss/Incorrect	Incorrect	✓	✓
Miss	Hit	✓	–
Incorrect	Hit	✗	–
Hit	Incorrect	–	✓
Hit	Miss	–	✗

†Updated roughly half of the time.

whether the attacker can implement a more portable strategy to select IP-based predictions, regardless of the particular configuration. In detail, we study the impact of user-mode iBTB and BTB manipulation on predictor selection.

iBTB manipulation. Prior work [2], [16] evidenced that Intel CPUs supporting eIBRS record privilege mode information in the iBTB. In this section, we want to study if an attacker can evict kernel iBTB entries from a lower privilege mode. To this end, we disable SMAP/SMEP and configure our test suite to execute a user training branch twice, first in kernel mode jumping to a target *K* and then in user mode jumping to a target *U*. Then, we execute a victim branch in kernel mode to inspect the iBTB state. In the experiment, we use the same history H_V for all the branch paths. Our results show that the user-mode training branch fails to evict the kernel-mode entry (*K* signal), suggesting that privilege information is used when matching the iBTB tag or set.

Next, we replace the user-mode training branch with a full iBTB eviction set walk in user mode. Based on the results of recent work [16], we build an iBTB eviction set by executing 6 indirect branches with different [15:5] IP bits and the same history H_V. With this change, our results now show a misprediction to the matching 32-bit target of *K*, evidencing eviction of the kernel iBTB entry and the CPU resorting to the matching BTB entry. These results indicate the privilege level is encoded in the tag, but not used for set indexing. We confirmed this eviction strategy is feasible on all tested microarchitectures.

BTB manipulation. Armed with the ability to evict a kernel iBTB entry, we now want to study whether an attacker can force a victim kernel branch to use IP-based (rather than history-based) prediction. We start by repeating our last experiment—i.e., using an iBTB eviction set to evict the H_V entry—but now with a different history H_T for the training branch. Our results again show a misprediction to the matching 32-bit target of *K*. Next, we test if we can train across iterations, which is a more realistic attack scenario than starting with a cold predictor state at each iteration. This time, our results no longer reveal a misprediction to our target, indicating that the presence of the updated BTB entry is dependent on the predictor state. As a result, we study in which states the predictor updates iBTB and BTB.

For both the BTB and the iBTB there are 3 possible states post prediction: a miss, a (correct) hit, and an incorrect hit. To simulate all the states, we need to be able to evict the BTB entry as well. Confirming prior findings [16], we find that we can evict the BTB entry with a single user direct branch which aliases the victim branch. Lastly, we need to be able to inject a BTB and iBTB target to simulate both correct and incorrect hits. We do this by executing the training branch with either the correct target *K* or an incorrect dummy target. If we want to inject distinct BTB and iBTB targets, we have to execute the training branch twice and use a randomized history for the BTB injection. Now by injecting either a correct or incorrect BTB/iBTB entry and using the two eviction methods, we can create all possible predictor states. For each state, we prime the predictor, then execute the training branch. Next, we infer the updated entries by observing the victim misprediction.

The results (Table 2) show the training branch does not update the BTB if there is an iBTB hit and an incorrect BTB entry is present. This clarifies why the BTB entry was not successfully updated earlier: when executing the training branch, the correct iBTB entry was present, preventing the update of the existing (incorrect) BTB entry, which was, in turn, updated by the victim branch in the previous iteration. Hence, for self-training to reliably update a BTB entry, the attacker needs to either evict the existing BTB entry or evict the H_T iBTB entry before executing the training branch. Besides evicting the H_V iBTB entry, we configure our test suite to also evict either the BTB entry or the H_T iBTB entry. This stably yields a successful victim misprediction, forcing reliable IP-based prediction on all our microarchitectures.

Summary. A self-training attacker can exploit iBTB and BTB evictions and force a victim kernel branch using history-based prediction to switch to IP-based prediction from a lower privilege mode.

5.3. IP-Based Training

In this section, we study the conditions to trigger IP-based collisions. First, we reverse engineer BTB properties to confirm results of prior work and disclose details for the newer microarchitectures. Following previous analysis techniques [12], [27], we configure our test suite with the same direct branch as training/victim and enable the prefetcher-based covert channel to detect collisions. Next, we infer the bits used for BTB indexing/tagging by flipping one or two bits of the victim IP and testing if we still observe the signal.

To infer which bits are used for set indexing, we want to test which IP bits we can change without indexing a different set. Therefore, we first find a BTB eviction set (of direct branches), which, when executed, evicts the BTB entry of the training branch. To prevent a match on both tag and set, we ensure the eviction branches are not a full alias of the training branch nor of each other. With a BTB eviction set, we can now test when victim and training branches index different sets by flipping the IP bits of both branches one by one. For each test, we execute the training branch, walk the eviction set, and execute the victim branch. If we observe a signal, and thus the eviction set fails to evict the BTB entry, training and victim branches index different sets.

Table 3: BTB properties for the tested microarchitectures.

μArch	Set index	Tag	Target length
Comet Lake	[13:5]	[29:22]⊕[21:14]	32 / 10
Sunny Cove	[13:5]	[33:24]⊕[23:14]	32 / 12
Golden Cove	[14:5]	[23:15]	32
Lion Cove†	[12:4]	[21:13]	32
Gracemont	[14:5]	[24:15]	32
Crestmont	[15:6]	[25:16]	32
Skymont	[15:6]	[25:16]	32

†Indexing based on entry-point address

Finally, we check if the BTB supports short targets and, if so, of which target length. To this end, we randomize the training target offset and the number of short target bits, while testing the signal for both short and 32-bit targets. To be able to distinguish short targets from 32-bit targets, we flip one or two IP bits below bit 32 of the victim branch, while maintaining a full alias of the training branch.

Results. Table 3 presents our results. We observed the presence of short BTB targets only on Comet Lake and Sunny Cove. Our results also show that the likelihood of a short entry being filled increases if the evicting branches, with a 32-bit target, create contention on the same BTB set.

Furthermore, we observed a significant change in the way the BTB is indexed on Lion Cove. Our experiments show that the entry-point address (i.e., the last jump target) rather than the branch address itself is used to index the BTB. The branch address is only validated before speculative execution. While this may seem like a subtle change, it has substantial implications: the BPU can look up targets earlier, and a single branch can now allocate multiple BTB entries—one for each possible entry point. We detail our findings on this new indexing scheme in Appendix B.

Collisions across branch types. Next, we want to analyze collision behavior across branch types. We configure our test suite with a training branch and a victim branch. We randomize for both branches: the type (call/jump/ret/je and direct/indirect), the offset, and the IP bits. To randomize the IP bits, we alternate between randomizing (i) the lowest 35 bits, (ii) only the TAG bits, or (iii) only the SET + lower remaining bits. We run our experiments for 48 hours on all our microarchitectures and discuss the main findings below.

Indirect-to-indirect branch collisions. In contrast to what we observed for history-based prediction, our results only reported jump-to-jump or call-to-call collisions (i.e., successful IP-based training). This suggests that the BTB maintains a branch type field, instead of shifting the type in the tag.

Moreover, our results show that Willow Cove (based on Sunny Cove) can fill and consume short (12-bit) BTB targets for indirect branch prediction. We did not observe indirect branch targets filling short entries elsewhere.

Direct-to-indirect branch collisions. Our results also revealed unexpected behavior on Comet Lake, Sunny Cove and Lion Cove. Specifically, for particular test suite configurations, our results reported an indirect branch prediction to a direct branch target. We further investigate this behavior in the next section.

Table 4: Prediction scenarios when the direct branch BTB entry is served as predicted target for an indirect branch.

Victim prediction State		*Source of predicted target*		
BTB	**iBTB**	**Indirect BTB**	**Indirect iBTB**	**Direct BTB**
Hit	Hit	–	✓	–
Hit	Miss	✓	–	–
Miss	Hit	–	–	✓
Miss	Miss	–	–	–

Summary. A self-training attacker can exploit IP-based collisions across matching kernel indirect branch types. Some microarchitectures feature short-target speculation or even direct-to-indirect branch collisions.

5.4. Direct-to-indirect Branch Training

A closer inspection of our results show that the direct-to-indirect branch collisions on Comet Lake and Sunny Cove (i.e., our 10th- and 11th-generation CPUs) occur in different conditions compared to Lion Cove (i.e., our Ultra Series 2 last-generation CPU). We first discuss our findings on Comet Lake and Sunny Cove and defer the Lion Cove discussion at the end of this section.

On Comet Lake and Sunny Cove, the direct-to-indirect branch collisions occur when the BTB entry matches the full tag and all the set bits except bit 5. More specifically, we only observe a collision if the victim indirect branch is on the lower half of the cache line (i.e., IP[5] == 0) and the training direct branch is on the upper half of the cache line. Interestingly, this implies that the two branches occupy a different BTB set, yet the direct branch is still used for the indirect branch prediction. We will research this behavior further below.

Predictor state requirements. Our results show that most direct-to-indirect collisions result in an unstable misprediction of the victim. However, the stability seems to be dependent on the predictor state. A closer inspection reveals that increasing the branch offset of the training branch results in a more stable misprediction of the victim branch, hinting that the code executed before the training branch modifies the predictor state. We hypothesize that when the CPU executes non-branching code on an IP that aliases a BTB entry, the BTB entry is invalidated. As the code executed before the training branch aliases the victim's BTB entry, it may evict the BTB entry of the victim.

To test this hypothesis, we repeat our experiment, but, instead of adding an offset to the training branch, right before the training branch we jump to a series of `nop` instructions whose IP aliases the victim's BTB entry. With the change, we now successfully observe stable misprediction to the direct branch target. Our results also show that BTB entry invalidation due aliasing code only occurs if the

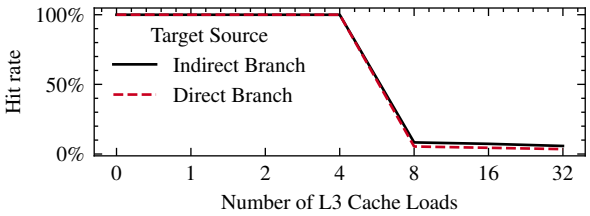


Figure 3: The speculation window size for indirect branch misprediction to an indirect or direct branch target. The size is measured by the median hit rate on a L3 load chain.

executed code is in the same privilege level as the BTB entry. However, an attacker can evict the BTB entry via an aliasing user branch instead, as shown earlier in Section 5.2.

Next, we want to check for additional constraints on the prediction state. To this end, similar to the predictor-state experiment in Section 5.2, we simulate all predictor states and check for mispredictions to the direct target. Table 4 presents our results. Summarizing, we only observe mispredictions to the direct target on a BTB miss and an iBTB hit. Interestingly, this behavior deviates from regular indirect branch prediction, where iBTB entries take precedence.

With the new insights, we run a new experiment to test all the IP[5:0] combinations with the correct predictor state. Our results show that all the IP[4:0] combinations result in a collision, as long as the indirect branch and direct branch are on the lower and upper half of the cache line, respectively.

Speculation window size. To characterize the nature of the collisions and for instance assess whether we are observing the effect of early-frontend speculative execution and branch type confusion similar, in spirit, to phantom jumps on AMD [28], we compare the speculation window size when training with a direct vs. indirect branch. To this end, we create a chain of dependent loads at the training target. To limit the number of required loads, we ensure the load addresses are evicted from L2 and thus served from L3.

Figure 3 presents our results. As shown in the figure, the hit rate for both branch targets is close to 100% for up to 4 L3 loads, after which it decreases to 4% at 32 L3 loads. Our results demonstrate the window size induced by direct branch targets matches the one of normal indirect branch speculation, ruling out early-frontend speculation.

Branch and target types. As discussed earlier, the BTB records a jump/call/ret branch type, preventing training across different branch types. However, our results show that training across branch types *is* possible for the direct-to-indirect branch training scenario. For instance, we can train an indirect call with a direct jump or an indirect jump with a direct call. In case of a RSB underflow, we can also train return prediction via a direct branch. Moreover, other that than 32-bit target prediction, we also observed short target prediction on all the tested microarchitectures—which we previously only observed for Willow Cove's normal training scenario.

We also evaluate whether the conditional branch predictor (CBP) is involved when using a conditional branch for indirect branch training. To this end, we configure our test suite to train the conditional branch to be not-taken with history H_V. The training branch is executed twice: the first time by setting the BHB to H_T and taking the branch; the second time by setting the BHB to H_V and not taking the branch. Next, the victim branch is executed with history H_V. We still observe a consistent misprediction to the direct branch, demonstrating the CBP is not involved.

Training on Lion Cove. In contrast to the variant discussed above, Lion Cove's direct-to-indirect collisions result in a stable misprediction of the victim. The training is successful when the direct and indirect branches fully alias in the BTB and share the same branch type, similar to IP-based indirect-to-indirect training. However, an iBTB entry is not given precedence for direct-to-indirect training, i.e., the predictor selects the direct target regardless of the iBTB state.

Next, we repeated the window-size experiment and again ruled out early front-end speculation. We also run the CBP experiment again and found that, unlike before, the CBP is involved when we train with a conditional branch. Specifically, when the training branch is executed with history H_V and not taken—whether on the first or second training execution— we observe no prediction for the victim branch, indicating that the CBP predicted not-taken.

Summary. A self-training attacker can exploit direct-to-indirect branch collisions with a full speculation window. On Comet Lake and Sunny Cove, the direct and indirect branches must have matching BTB tag and set bits excluding IP[5] (which must be 0 and 1, respectively). Lion Cove needs a full IP-based collision.

6. Self-training Attacks

Based on our analysis results, we can now derive three self-training attack classes (Figure 4). First, our results show that (BHI) mitigations for history-based training only prevent a lower-privilege branch history from affecting higher-privilege indirect branch prediction behavior—unless they disable indirect branch prediction altogether. If attackers can find a way to create iBTB collisions with solely privileged branches, they can bypass the mitigations. We refer to this class of attacks as *self-training history-based* attacks.

Second, our results show that attackers can piggyback on mitigations such as BHI_DIS_S and BHI_NO or, more portably, rely on iBTB and BTB evictions to force IP-based predictions for a target privileged indirect branch. If attackers can also find a way to trigger an aliasing privileged indirect branch with a matching disclosure gadget, they can speculatively hijack control flow and leak information. We refer to this class of attacks as *self-training IP-based* attacks.

Finally, our results uncovered two variants of direct-to-indirect training attacks. This behavior extends the self-training IP-based attack surface to disclosure gadgets at

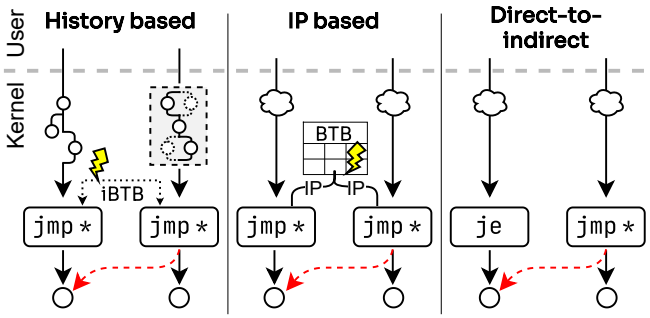


Figure 4: Self-training attack classes. The history-based attack creates an iBTB collision via a history-crafting gadget (HG), the IP-based attack creates a BTB collision with an aliasing victim branch, and the direct-to-indirect attack uses direct branches to train indirect branch prediction.

direct branch targets. The first variant, found on our Comet Lake and Sunny Cove CPUs (10th and 11th generations), can train across branch types with loose IP collision requirements. Following Intel's brand name for this issue after our disclosure—Indirect Target Selection (ITS)—we refer to attacks for this variant as *self-training ITS* attacks. The second variant, found on our Lion Cove CPU (Ultra Series 2), requires a full IP-based collision—similar to indirect-to-indirect training. We refer to attacks for this variant as self-training direct-to-indirect attacks on Lion Cove.

7. History-Based Exploitation

In response to BHI [2], the Linux kernel has disabled unprivileged eBPF [29], which would otherwise allow *in-domain* attackers to run code in the kernel with arbitrary history randomization, indirect branches, and disclosure gadgets. In this section, we investigate whether a *cross-domain* self-training attacker can craft similar primitives without eBPF for successful history-based exploitation. For our analysis, we consider all the microarchitectures with history-based prediction enabled in kernel mode and use the default (pre-compiled) Ubuntu kernel version 6.8.0-38.

Controlling the branch history. The first primitive the attacker needs to craft is a way to control the kernel branch history. While one can in principle look for gadgets in arbitrary kernel code paths, our initial analysis in this direction did not lead to actionable results. As such, to ease exploitation, we turn our attention to cBPF filters, which an unprivileged attacker can freely install in the kernel by means of SECCOMP [9] or LSF [10]. Although crafting indirect branches and disclosure gadgets is no longer an option with cBPF, attackers can specify simple filter conditions, which are lowered to direct branches and hence provide attackers with control over the branch history.

For simplicity, we focus on SECCOMP, which executes cBPF filters right after entering the kernel to handle a syscall. As such, the filter-generated kernel branch history

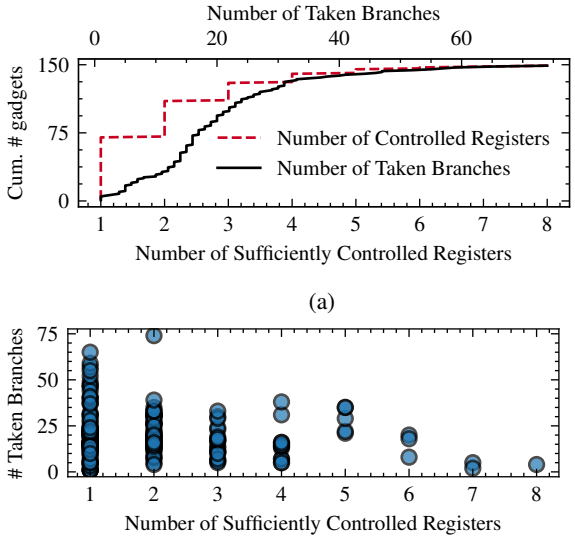


Figure 5: CDF (a) and scatter plot (b) of # taken branches and controlled registers for the indirect branches found.

is not affected by mitigations such as BHB clearing or BHI_NO. To confirm this behavior, we craft a filter which compares each bit of the syscall arguments and takes a branch if the corresponding bit is 1. Next, we create a custom syscall containing a training and a victim indirect branch and attempt to mispredict the victim branch to the training target. This is done by merely randomizing the syscall arguments in search for history collisions. Our experiments show we can successfully mispredict the victim branch to the training target on all our machines.

Victim branches. With the Linux syscall-dispatch indirect branch hardened in response to Native BHI [4], [30], we look for victim indirect branches in syscall-handling functions. To this end, we use InSpectre Gadget, a state-of-the-art Spectre scanner using symbolic execution [4]. We start our analysis at each syscall function entry point and repurpose the InSpectre's dispatch gadget detection to detect victim branches. We adapt the scanner to hook common Linux kernel functions in order to reduce the number of states (e.g., locks) and propagate user control at `copy_from_user` and similar. In addition, to increase the scanning depth, we help the scanner resolve indirect branches with a list of targets derived by offline dynamic analysis. We create the list by hooking indirect branches with kprobes and executing a workload based on the Syzkaller corpus [31].

As shown in Figure 5, we found 149 indirect branches with at least one register "sufficiently" controlled, i.e., able to point to the kernel address space. For Comet Lake and Sunny Cove, which have a BHB size of 29 and 66, the attacker needs to control at least 9 and 8 branches, respectively. Depending on the syscall, it can take from 10 up to 18 taken branches before entering the syscall function entry

Table 5: Branches in the kernel text and module region.

Source	# Indirect branches	# Direct branches
Kernel text	11,250	882,457
Module region	8,955	339,011
Total	20,205	1,221,468

point. Even assuming 18 taken branches, we still found 6 and 138 unique indirect branches with at least 1 sufficiently controlled register, reachable by 6 and 70 different syscalls, for Comet Lake and Sunny Cove, respectively.

End-to-end exploit. As victim, we select the `security_task_prctl` indirect branch, which allows the attacker to control 7 registers. As gadget, we rely on the dispatch gadget `unix_poll`, used by Native BHI [4]. This is to jump to an attacker-chosen location where we can load a secret value and encode it into an attacker-controlled buffer. To get a shared buffer with the kernel, we first break KASLR via the prefetch side channel [32] and use the huge-page finding technique from Native BHI. We tested our end-to-end exploit on the Tiger Lake and Lunar Lake CPUs and successfully leak kernel memory at 1.7 KB/sec.

8. IP-Based Exploitation

We now investigate whether a self-training attacker forcing IP-based kernel predictions (by means of BHI mitigations or iBTB/BTB evictions) can exploit such predictions to mount attacks. We focus here on out-of-place (different training/victim indirect branch) IP-based attacks and refer the reader to Appendix E for an analysis of in-place (same-branch) attacks—for which we found no exploitable gadgets.

To perform an out-of-place IP-based attack, the attacker needs to find two indirect branches with BTB-aliased IPs. Moreover, the training branch needs to architecturally target an entry point that can serve as a disclosure gadget. Finally, the victim branch must grant the attacker sufficient control over registers/memory matching the disclosure gadget. We now investigate the prevalence of such IP collisions on Linux and analyze their exploitability. We consider two collision scenarios, *text* collisions (i.e., within kernel text) and *module* collisions (i.e., between text and module region).

Collision frequency. Our Ubuntu kernel image has a kernel text size of 22 MB (covering 25 IP bits), and, as shown in Table 5, contains around 11,000 indirect branches. Looking at Table 3, kernel collisions may occur on all our microarchitectures except Crestmont and Skymont, whose BTB tag includes bits up to 26. Moreover, since Kernel Address Space Layout Randomization (KASLR) randomizes the kernel text base with a 2 MB alignment, collisions on Comet Lake and Sunny Cove are also KASLR entropy-dependent.

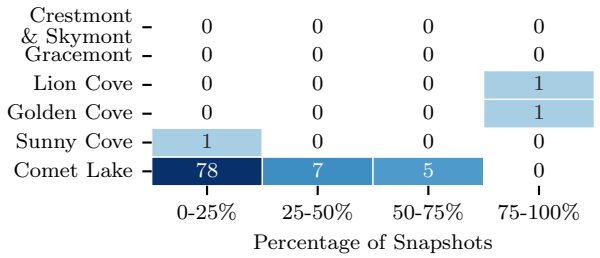
In addition to indirect branches in the kernel text, an attacker can exploit user-reachable indirect branches in the kernel modules (e.g., device drivers) to train an indirect branch in the text. On Linux, kernel modules are allocated by `vmalloc` in the module region, which follows the text region in the address space and thus can cause indirect branch collisions on all our microarchitectures. A kernel module is mapped to a page-aligned address, whose entropy is subject to the start of the module region (and thus to KASLR) as well as previous module allocations. To get an indication of the number of loaded modules for a default Ubuntu boot, we booted the Raptor Lake CPU and found 149 modules loaded after boot. We observed similar numbers on the other machines, with variations explained by the different device drivers loaded across configurations.

To investigate the collision frequency of indirect branches for both text and module collisions, while accounting for KASLR and allocator entropy, we analyzed snapshots (memory dumps) for different (100) kernel boots. For every snapshot, we boot a VM, load the modules present on bare metal, note the text start/end address of each module, and create a memory dump of the VM. Next, we extract the indirect branch locations from the dump and calculate the collision pairs for each microarchitecture. For module collisions, we exclude (very high entropy) module-to-module collisions and only report text-to-module collisions. For Lion Cove, which uses the entry-point address for BTB indexing, we constructed a CFG for every kernel and module function and extracted the entry points per branch (ignoring indirect branches in the CFG).

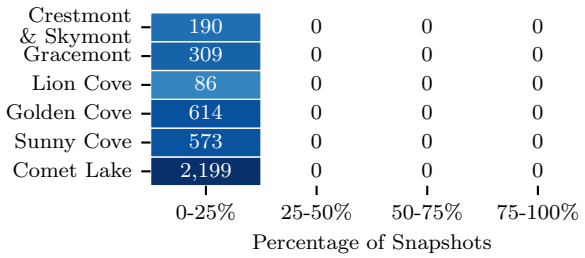
To analyze the stability of the collisions, we counted the number of occurrences of each unique collision pair across the snapshots. As shown in Figure 6, module collisions are much more frequent than text collisions. Nevertheless, we still found 90 unique text collision pairs on Comet Lake that are mostly unstable across snapshots due to KASLR. For module collisions, Comet Lake and Lion Cove incur the lowest and largest number of collisions (i.e., 86 and 2,199, respectively). Given the multiple sources of entropy, the collision pairs and the colliding branches are mostly unique. For example, the 2,199 unique collision pairs on Comet Lake cover 2,074 different text branches. In other words, a large-scale attacker can exploit KASLR and allocator entropy to, in principle, generate near-arbitrary module collisions with user-reachable branches. The only constraint is that the lower 12 bits of the branches must match due to page alignment, but a large-scale attacker can also exploit entropy across kernel versions, configurations, etc. to lift this constraint.

Gadget analysis. To evaluate the attack surface of the text and module collisions found, we instruct InSpectre Gadget to analyze the corresponding targets for exploitable gadgets up to a depth of 8 basic blocks. InSpectre Gadget supports two types of gadgets: disclosure gadgets based on the cache covert channel, which encode the secret via a store or load, and dispatch gadgets, which allow the attacker to speculatively hijack control flow to an arbitrary target. To better reflect available techniques, we also implemented support for the BTB covert channel [33], [34], with conditional branches comparing a secret against a controlled value.

To analyze the gadgets for text collisions, we rely on the



(a) Collisions within kernel text



(b) Collisions between kernel text and the module region

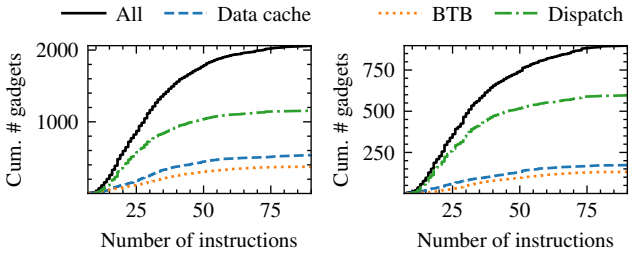
Figure 6: Distribution of unique indirect branch IP collision pairs across 100 kernel boots.

results of our dynamic analysis (Section 7) to get targets and controllability requirements for InSpectre. Out of the resulting 112 targets, we found 26 exploitable gadgets. However, none of these gadgets match the controllability requirements on our particular kernel build.

To generalize, we configured the scanner to analyze all the targets profiled by our dynamic analysis. Our results show that, out of the 1,338 indirect branches dynamically reached by our analysis, 823 branches have a target containing an exploitable gadget matching the controllability requirements of at least one of our 149 victims branches. On average, a single victim branch has 39 indirect branches that yield matching exploitable gadgets upon collision. In other words, a kernel build is exploitable if any of the $149 * 39 = 5,811$ collisions pairs happen to collide.

Next, we calculated the probability of such collisions, assuming indirect branch locations are randomly distributed throughout kernel text. For a BTB algorithm entropy of 22 bits (Comet Lake), the probability of a collision between two fixed indirect branches is 2.4×10^{-5}%. However, assuming an average of 149 victim and 39 matching branches, the probability of an exploitable collision increases to 0.14%. For the other microarchitectures except Lion Cove (where entry-point matching complicates estimation), with an entropy of 24, 25, and 26 bits, the probability is slightly lower (i.e., 0.04%, 0.02%, and 0.01%, respectively). These results indicate that, while the probability of an IP-based collision on a single given kernel build is low, a large-scale attacker targeting thousands of machines is likely to find exploitable collisions, assuming sufficient entropy is introduced by different kernel versions, kernel configurations, or randomization features such as FGKASLR [35].

For module collisions, considering the high volatility of the collision pairs observed in our experiments and the challenges to operate dynamic analysis of device drivers, we instead opt for a static analysis approach. Specifically, we use Clang-CFI's type analysis [36] to find all valid targets for all the module indirect branches, resulting in total of 8,803 unique targets. We used again InSpectre Gadget to analyze all targets for gadgets. As shown in Figure 7a, we found 2,063 gadgets that are marked as exploitable, across 1,938 unique targets. Next, for each gadget, we verified if there is a potential victim with matching controllability



(a) All exploitable (b) With matching victim control

Figure 7: Gadgets at kernel module targets.

requirements, resulting in 898 gadgets (see Figure 7a) that an attacker may be able to abuse in a large-scale attack. These gadgets are targeted by 1,238 distinct module indirect branches. On average, a single victim branch has 59 indirect branches that yield matching exploitable gadgets upon collision. Repeating the probability calculation from earlier reveals a probability to find an exploitable module collision of 0.21%, 0.05%, 0.03% and 0.01% for 22, 24, 25, and 26 bits of entropy (respectively). Again, this shows a large-scale attacker is likely to find exploitable collisions. Note that, compared to text collisions, module collisions incur additional (module allocation) entropy.

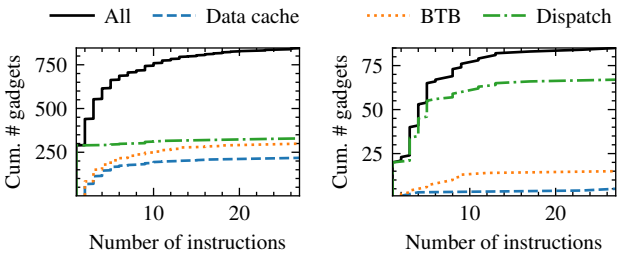
9. Direct-to-indirect Exploitation

We now investigate whether a self-training attacker can exploit direct-to-indirect training to leak kernel memory. We will first discuss ITS exploitation with or without cBPF. Next, we briefly discuss the harder-to-exploit direct-to-indirect training on Lion Cove.

9.1. ITS Exploitation With cBPF

As shown in Section 7, attackers can abuse cBPF to inject taken/not-taken conditional kernel branches. We now show attackers can also abuse cBPF to train a kernel indirect branch via ITS. This is challenging, as the victim branch has to alias with the cBPF-injected direct branch.

Victim branch and disclosure gadget. A cBPF-injected



(a) (b)

Figure 8: CDF of # instructions for exploitable gadgets found near victim indirect branches for cBPF ITS exploits, all gadgets in (a) and gadgets with a matching victim in (b).

branch can only forward-jump within the filter, but an attacker cannot inject a disclosure gadget in the same filter. To address this challenge, we exploit short branch targets, which cause the predictor to compute the final target by combining the branch IP with the stored short target offset. This allows an attacker to speculatively hijack the victim indirect branch to a nearby location—i.e., a forward 10- or 12-bit offset on Comet Lake and Sunny Cove (respectively).

To find a disclosure gadget and a matching victim branch, we configure InSpectre Gadget to analyze all the addresses within the short-branch offset from the victim indirect branches reported by our dynamic analysis (Section 7). We scan with a max depth of 5 basic blocks. Figure 8 presents our results. We found 846 unique exploitable gadgets, of which 85 have controllability requirements matching a victim indirect branch. From the results, we select a dispatch gadget in `security_mmap_addr` and a matching victim indirect branch in `security_mmap_file`, with one of the controlled registers being `rbx`. As the offset between the victim and gadget is only 138 bytes, i.e., within the 10-bit target offset, we can use the gadget on both Comet Lake and Sunny Cove. We configure the dispatch gadget to jump to an out-of-band disclosure gadget in `cmp_entries_key`, which requires a controlled `rbx`. Listing 1 in Appendix C shows the complete gadget chain.

cBPF memory massaging. Our next step is to massage the cBPF memory allocator to predictably allocate a cBPF filter in a target page which contains the IP we want our victim indirect branch to alias. To this end, we first use the prefetch side channel [32] to break KASLR and leak the start/end address of the module region (where cBPF filters are allocated). Next, we request many 4 KB dummy cBPF filters to fill any gaps in the module region and force the allocator to start allocating filters consecutively in (new) memory. We detect this behavior by scanning the module region with the prefetch side channel and looking for evidence of a new huge page (2MB), which the allocator creates to store a new filter when there are no gaps left [37]. As a result, we leak the location of the huge page and thereby the base address of the next cBPF filter (4 KB higher in memory). Having forced predictable allocation behavior,

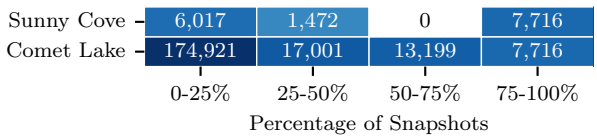


Figure 9: Unique ITS collision pairs across 100 kernel boots.

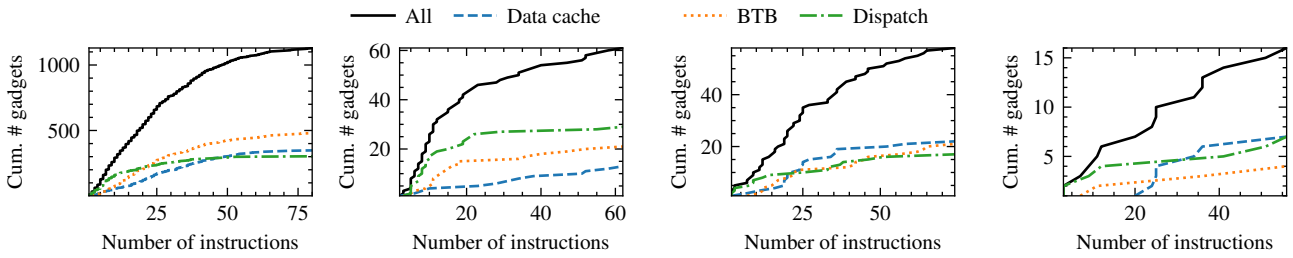
we keep requesting dummy filters until we know the next filter will land on the target page. To bypass the limit on the number of filters per process [10], we distribute the workload across as many child processes as necessary.

Breaking cBPF randomization. By now, we know the address of the next cBPF filter in memory and, since cBPF instruction generation is deterministic, we could easily craft a filter with the training direct branch at the right offset to match the target IP. However, Linux adds a randomized offset (hole) filled with illegal instructions [38] at the start of each filter, resulting in a nondeterministic direct branch address. Nonetheless, we can again exploit predictable allocator (memory reuse) behavior to brute-force the right offset. To this end, we spawn a child process and have the child request the target cBPF filter, check for evidence of successful training, and exit (deallocating the filter) upon failure. We repeat the process until we observe evidence of successful training, which we can gather by loading data into the cache on the speculative path. For this purpose, we rely on a FLUSH+RELOAD covert channel and brute-force the alias of the user reload buffer in the kernel direct map at 2 MB granularity, as done in prior work [4]. However, we do so while also brute-forcing the target cBPF filter placement. Despite the two sources of entropy, we observe a signal within 45 seconds on both tested CPUs. By creating contention on the direct branch 32-bit BTB set, we can reliably fill a short BTB entry (as shown in Section 5.3).

Leaking kernel memory. We now have the target cBPF filter (and training direct branch) in the right location and know the kernel location of our reload buffer. Hence, we can set up the gadget chain to jump to our disclosure gadget, load the secret, and transmit it with FLUSH+RELOAD. As our gadget uses a 32-bit secret, we rely on the known-prefix technique to leak arbitrary secrets [3], [4], [39], [40]. Specifically, our exploit leaks the root password hash from memory. We do so by calling `passwd -s` (which stores the hash into memory) and searching for the "root:" prefix, as done in prior work [39]. Our end-to-end exploit successfully leaks the hash from memory in 60s on average. The leakage rate, after initialization, is 15 KB/sec and 17 KB/sec on the Comet Lake and Rocket Lake CPUs (respectively).

9.2. ITS Exploitation Beyond cBPF

Rather than using cBPF, an attacker may use a direct branch present in kernel text to train the victim indirect



(a) All gadgets on Comet Lake (b) All gadgets on Sunny Cove (c) Subset w/ matching victim (d) Subsubset w/ reachability

Figure 10: CDF of # instructions for exploitable target gadgets for ITS collisions within kernel text, where (a) and (b) show all exploitable gadgets for Comet Lake and Sunny Cove, respectively, (c) the gadgets with a matching victim, and (d) the gadgets with both a matching victim and a direct branch reached by Syzkaller.

branch. In this section, we study the feasibility of ITS exploitation beyond cBPF.

Collision frequency. To analyze the frequency and stability of the collisions, we again create 100 snapshots after boot and calculate the collision pairs across the snapshots. As shown in Figure 9, we found significantly more ITS collisions pairs compared to indirect-to-indirect collisions—and, for this reason, we only consider text rather than also module collisions. First, there are more direct branches in the kernel text (see Table 5). Moreover, ITS does not require a match on the lower 5 IP bits. Nevertheless, roughly half of the direct and indirect branches are ruled out, as they reside on the wrong half of the cache line. The lower 6 bits are not subject to KASLR, but are likely affected by even minor changes to the kernel version, configuration, compiler, etc. Therefore, we expect collision pairs to generally fluctuate across kernel versions or builds.

Gadget analysis. Next, we want to analyze the (colliding) targets of the victim indirect branches we found, in order to locate exploitable Spectre gadgets. Since direct branches can insert a short BTB target if the branch offset is within the short bits, we also include the short targets computed by combining the victim branch IP with the short-target bits of the direct branch. This results in 4,611 distinct targets across the 149 victim branches.

Figure 10 presents our results. The scanner found 1,130 and 62 exploitable gadgets on Comet Lake and Sunny Cove, respectively. Of those gadgets, a total of 59 gadgets (58 and 1 for Comet Lake and Sunny cove, respectively) match the controllability requirements of their corresponding victim. Finally, the attacker also needs to be able to trigger the direct branch to inject the BTB entry. To have an indication of how many direct branches are reachable, we cross-reference the DWARF symbols with the coverage report from Syzkaller [31]. As a result, we ultimately found 16 gadgets for Comet Lake and 1 gadget for Sunny Cove with training branch and matching victim reached by Syzkaller. We manually analyzed the collision and the resulting dispatch gadget we found for Sunny Cove, which happens to also be present on Comet Lake, and confirmed it to be end-to-end exploitable (see Appendix D for more details).

9.3. Lion Cove Exploitation

Direct-to-indirect training on Lion Cove is subject to stricter collision requirements compared to ITS, requiring a matching IP and branch type. Moreover, Lion Cove's BTB does not support short target predictions. In this section, we discuss Lion Cove exploitation with and without cBPF.

Exploitation with cBPF. Direct branches injected via cBPF can only jump to addresses within a cBPF program. As such, and since there are no short targets, speculative redirection is constrained to the address range of the cBPF program. Nonetheless, as shown in prior work [41], an attacker can exploit the 4-byte controllable intermediate to inject instructions that are interpreted as such if the program is executed unaligned. To trigger unaligned speculative execution, the attacker first trains the victim by executing the cBPF program, then re-inserts the cBPF program with crafted gadgets that probabilistically fall unaligned at the predicted target. A greater challenge is to find a suitable victim indirect branch. Because the training and victim branches must share the same branch type —and only direct branches can be injected via cBPF—the attacker must rely on a victim indirect jump. Our analysis found only six indirect jumps in the Linux kernel image, none of which appear to offer attacker control. However, the kernel features thousands of switch statements, which may be lowered to indirect jumps depending on the compiler configuration. In other words, the attack surface appears slim on our system, but other real-world configurations may prove more vulnerable.

Exploitation without cBPF. To evaluate the impact of direct-to-indirect training for text and module collisions, we repeated our collision frequency experiment. We found 22 stable text collisions across 100 snapshots (75-100% category) and 2,296 module collisions, most of which were unique (0-25% category). Compared to indirect-to-indirect collisions on Lion Cove, the number is significantly larger given that direct branches are much more common. However, collisions remain far less frequent than for ITS (which

has less strict collision requirements), limiting exploitation opportunities to large-scale attack scenarios.

10. Breaking Isolation

In the previous sections, we introduced direct-to-indirect training attacks, and, in particular, showed ITS can be abused to mount end-to-end self-training kernel exploits. We now discuss how this class of attacks can also break deployed domain isolation techniques in a classic cross-training scenario.

Cross-context. To protect against user-to-user or guest-to-guest Spectre-v2 attacks, the kernel/hypervisor executes IBPB upon context switch to flush indirect branch prediction entries [18], [42]. However, direct branch entries may not be flushed. To confirm this intuition, we first test ITS by configuring our test suite with a user-mode ITS training scenario and execute IBRS between the training and victim branch. Our results on our Comet Lake and (both) Sunny Cove machines show that the 32-bit direct branch targets are flushed but the short targets are not, affecting victim prediction. This shows ITS breaks IBPB isolation and allows cross-context attackers to mispredict any victim indirect branch on the lower-half of a cache line to an attacker-chosen victim target a short offset away from the branch.

Next, we ran the same experiment for Lion Cove direct-to-indirect training. Our results shows the 32-bit direct-branch targets also bypass IBPB on Lion Cove. This contrasts with the results on our ITS-vulnerable machines, where we observed IBPB flushing all 32-bit targets. We hypothesize that IBPB has become more fine-grained on newer CPUs, flushing only indirect targets for performance reasons. As a result, direct-to-indirect training on Lion Cove breaks IBPB isolation with even fewer constraints than ITS: a cross-context attacker can train any victim branch to an attacker-chosen 32-bit target. Overall, both variants break the IBPB mitigation and re-enable classic user-to-user or guest-to-guest Spectre-v2 attacks.

Cross-privilege. Next, we evaluate whether ITS or the Lion Cove issue can also break user-kernel isolation enforced by eIBRS [18]. We configure our test suite with a user-mode training branch and a kernel-mode victim branch. Our results show eIBRS still guarantees isolation on vulnerable microarchitectures, including for short direct branch targets. To our surprise, after disclosing our findings to affected vendors, Intel eventually informed us ITS *does* break eIBRS isolation, but only in a guest-host scenario. To confirm this behavior, we configure our test suite to use a user or supervisor direct branch in the guest and a user or supervisor victim indirect branch in the (KVM) host. Our results show ITS cannot bypass eIBRS isolation on our Sunny Cove machines, but does break isolation (for short targets) on Comet Lake between guest and host (Table 6). To showcase the resulting attack, we created a PoC with a custom hypercall containing a victim indirect branch and a disclosure gadget located a 10-bit short target offset away from the branch. Our results show we can successfully train the victim branch from our VM, leaking hypervisor memory at 8.5 KB/sec.

Table 6: Guest/host training results on Comet Lake.

Training and victim branch location			Training
guest-user	**guest-supervisor**	**host-supervisor**	**Successful**
Training	Victim	–	✗
Training	–	Victim	✓
–	Training	Victim	✓

11. Mitigations and Disclosure

We consider both hardware and software mitigations. On the hardware side, vendors may: (i) change the history-based predictor to match BHB and IP constraints separately (addressing our history-based collisions); (ii) maximize the BTB tag entropy (minimizing our IP-based collisions); (iii) forbid training/prediction across branch types. On the software side, history-based collisions can be mitigated by performing a BHB-clearance operation after the execution of each history-crafting gadget (and cBPF filter in particular). Disabling unprivileged cBPF is not an option, due to its widespread use (e.g., Docker) through the ecosystem. Regular IP-based text/module collisions, in turn, could be mitigated (or at least minimized) by rearranging the kernel address space layout. ITS collisions are much harder to mitigate in software and one option is to resort to more costly mitigations such as retpoline [43], [44] (when safe to do so) or the more recent IPRED_DIS controls [7].

Disclosure. We disclosed our main findings to vendors in March 2024. Intel confirmed our (history-based, IP-based, and direct-to-indirect) findings, issued two CVEs (CVE-2024-28956 for ITS and CVE-2025-24495 for the Lion Cove issue), and published two advisories (Intel-SA-01153 for ITS and Intel-SA-01322 for the Lion Cove issue). AMD confirmed their existing mitigations are sufficient. ARM confirmed our history-based findings also apply to ARM and published a security update on their developer website.

Regarding history-based training, Intel recommends applying a BHB clearance operation after each cBPF program and any other future history-crafting gadget. Intel also created a new instruction, Indirect Branch History Fence (IBHF), to enforce a history barrier. This instruction is available with a microcode update from Alder Lake and Sapphire Rapids CPUs onwards. Older CPUs instead need to rely on the software BHB-clearance sequence.

To mitigate cross-context ITS, Intel published a microcode update to fix IBPB isolation. To mitigate both user/kernel and guest/host ITS, Intel recommends placing all privileged indirect branches (including returns) on the upper half of the cache line to prevent direct-to-indirect branch training. To mitigate direct-to-indirect training on the Lion Cove, Intel published another microcode update, which completely removes the direct-to-indirect training capability.

Intel and ARM engineers have developed patches to incorporate their recommended mitigations into the Linux Kernel. For ITS, the mitigation allocates a memory region containing a separate indirect thunk for each indirect branch on the lower half of the cache line. This approach can also be used to reduce (or eliminate) text and module collisions for IP-based exploitation in general. Indeed, with one thunk per indirect branch allocated after another in a small text region, IP-based collisions are drastically reduced (if possible at all). At the time of writing, vendors do not recommend any additional mitigations for IP-based exploitation.

12. Related Work

There is extensive literature on reverse engineering branch predictors and hardware mitigations in modern CPUs [12], [13], [16], [17], [27], [28], [33], [45], [46], [47]. A number of recent efforts focus on demonstrating implementation flaws in domain isolation techniques against Spectre v2 [2], [3], [8]. We similarly show that cross-training ITS and the Lion Cove issue break isolation, but also study the broader scope of self-training attacks bypassing *perfect* isolation. BHI [2] demonstrates that an in-domain attacker can rely on eBPF to bypass Spectre-v2 domain isolation. For the first time, we show that this goal is attainable even for cross-domain attackers, despite their limited (e.g., cBPF) capabilities.

Indirector [16] reverse engineers the behavior of the indirect branch predictor to mount high-precision BTI attacks. In contrast, we focus on reverse engineering properties (e.g., branch type collisions) useful to analyze the attack surface of self-training attacks. IP-based predictor selection has previously been demonstrated in a same-privilege scenario by randomizing the victim's branch history [8] or using a same-privilege aliasing direct branch [47]—methods unavailable in our self-training scenario. In contrast, we show how to precisely fill and use a BTB entry in a cross-privilege setting.

Zhang et al. [47] abuse IP-based collisions to construct transient trojans (i.e., malicious software hiding their activity), i.e., transferring data across the user/kernel boundary via the microarchitectural state. To this end, they studied IP-based collisions and also observed direct-to-indirect collisions on Haswell, Skylake, and Kaby Lake Intel CPUs (4th, 6th, and 8th generations). They reported stable collisions when both branches fully alias in the BTB. They also reported early-frontend branch collisions if IP bit 5 (bit 4 on Haswell) was set to 0 and 1 for the indirect and direct branch, respectively. With ITS, we observed similar IP-aliasing conditions on Comet Lake and Sunny Cove (10th and 11th generation). However, we observed a full speculation window suitable for ITS exploitation, rather than early-frontend branch collisions.

For completeness, we repeated our experiments on our 9th-generation Intel Coffee Lake CPU (previously excluded due to IBRS). Our results show that, in addition to the collisions on the lower/upper half of the cache line, direct-to-indirect collisions occur if both branches fully alias in the BTB, with a full speculation window in both cases. Moreover, repeating our IBPB isolation experiment shows that IBPB correctly flushes both short and 32-bit BTB targets. This means that the direct-to-indirect collisions do not break domain isolation on our Coffee Lake CPU. Despite the difference in behavior compared to our ITS findings, Zhang et al. may have observed a predecessor of the ITS issue. Finally, in contrast to their efforts, we assume a threat model involving an unprivileged attacker seeking to leak privileged data via self-training attacks and present several gadget scanning campaigns and end-to-end exploits to study the resulting attack surface.

13. Conclusion

In this paper, we showed self-training Spectre-v2 attacks are not limited to in-domain scenarios with powerful sandbox environments, but also extend to the more serious cross-domain scenario. To support this claim, we presented three different classes of attacks exploiting history-based, IP-based, and direct-to-indirect branch collisions as well as two end-to-end exploits against the Linux kernel.

The impact of our findings is significant. First, deployed domain isolation techniques are still plagued by serious flaws, as evidenced by the two new hardware issues on Intel CPUs we uncovered. Both issues completely break user and guest isolation. One issue (Indirect Target Selection or ITS) also breaks hypervisor isolation, re-enabling classic guest-to-host Spectre v2 attacks in the cloud. Second, even *perfect* domain isolation is insufficient to address cross-domain Spectre-v2 attacks. In response to our findings, vendors have deployed a number of mitigations, including the new Indirect Branch History Fence instruction, which effectively transitions Spectre v2 to the gadget-oriented mitigation era—similar in spirit to Spectre v1.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. We also thank Intel researchers for comprehensively triaging ITS in response to our disclosure and for sharing their isolation-break findings with us. This work was supported by Intel Corporation through the "Allocamelus" project, by NWO through project "INTERSECT" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

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

[3] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022.

[4] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "InSpectre gadget: Inspecting the residual attack surface of cross-privilege Spectre v2," in *USENIX Security*, 2024.

[5] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security*, 2014.

[6] "Refined speculative execution terminology," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html, 2022.

[7] "BHI and intra-mode BTI," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[8] J. Wikner and K. Razavi, "Breaking the barrier: Post-barrier Spectre attacks," in *IEEE S&P*, 2025.

[9] "Seccomp BPF (SECure COMPuting with filters)," https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[10] "Linux socket filtering aka berkeley packet filter (BPF)," https://docs.kernel.org/networking/filter.html.

[11] L. Rappoport, R. Ronen, N. Kacevas, and O. Lempel, "Method and system for branch target prediction using path information," 2003.

[12] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *ISPASS*, 2009.

[13] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security*, 2017.

[14] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[15] AMD, "Software optimization guide for the AMD family 19h processors," 2020.

[16] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector:high-precision branch target injection attacks exploiting the indirect branch predictor," in *USENIX Security*, 2024.

[17] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution," in *IEEE S&P*, 2023.

[18] "Branch target injection," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html.

[19] "Vulnerability of speculative processors," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[20] AMD, "AMD64 architecture programmer's manual, volume 2," 2024.

[21] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture." in *USENIX winter*, 1993.

[22] "Classic BPF vs eBPF," https://docs.kernel.org/bpf/classic_vs_extended.html.

[23] K. Sun, K. Hu, H. Kawakami, M. Marschalek, and R. Branco, "A software mitigation approach for branch target injection attack," 2019.

[24] D. Jin, A. J. Gaidis, and V. P. Kemerlis, "BeeBox: Hardening BPF against transient execution attacks," in *USENIX Security*, 2024.

[25] L. Gerhorst, H. Herzog, P. Wägemann, M. Ott, R. Kapitza, and T. Hönig, "VeriFence: Lightweight and precise Spectre defenses for untrusted Linux kernel extensions," in *RAID*, 2024.

[26] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing mitigations for branch target prediction attacks," in *IEEE EuroS&P*, 2023.

[27] Z. Zhang, M. Tao, S. O'Connell, C. Chuengsatiansup, D. Genkin, and Y. Yarom, "BunnyHop: Exploiting the instruction prefetcher," in *USENIX Security*, 2023.

[28] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *MICRO*, 2023.

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

[30] L. Torvalds, "Don't force use of indirect calls for system calls," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e3ad78334a69b36e107232e337f9d693dcc9df2, 2024.

[31] "Syzkaller," https://github.com/google/syzkaller.

[32] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *CCS*, 2016.

[33] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *ASPLOS*, 2018.

[34] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *RSA*, 2006.

[35] "Finer grained kernel address space randomization," https://lwn.net/Articles/811685/.

[36] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grain control-flow enforcement with indirect branch tracking," in *RAID*, 2023.

[37] S. Liu, "execmemalloc for BPF programs," https://lwn.net/Articles/914128/, 2022.

[38] D. Borkmann, "net: Bpf: Consolidate JIT binary allocator," https://lore.kernel.org/all/1410156289-3190-2-git-send-email-dborkman@redhat.com/, 2014.

[39] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the Spectre era," in *CCS*, 2020.

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

[41] D. Jin, V. Atlidakis, and V. P. Kemerlis, "{EPF}: Evil packet filter," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 735–751.

[42] "Spectre side channels," https://docs.kernel.org/admin-guide/hw-vuln/spectre.html#spectre-variant-2-branch-target-injection.

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

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

[45] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *CCS*, 2007.

[46] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor," in *ASPLOS*, 2024.

[47] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *ASPLOS*, 2020.

[48] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *USENIX Security*, 2023.

Appendix A.
Tested microarchitectures

Table 7: Details for the tested microarchitectures.

Vendor	Model	CPUID	Microcode
Intel	Core i9-9900K	906EC	0xf8
	Core i7-10700K	A0655	0xfc
	Core i7-11700	A0671	0x62
	Core i7-11800H	806D1	0x52
	Core i9-14900K	B0671	0x12b
	Core Ultra 7 155H	A06A4	0x1f
	Core Ultra 7 258V	B06D1	0x111
AMD	Ryzen 9 7950X	A60F12	0xa601209
	Ryzen 9 9950X	B40F40	0xb404022

Appendix B.
Lion Cove BTB Indexing

While running our test suite on Lion Cove to study BTB properties, we noticed that the entry-point address (i.e., the last branch target prior to the branch), influenced BTB collisions. To understand the behavior of the new BTB scheme, we created additional experiments in our test suite specifically for Lion Cove. In this section, we discuss our results.

Indexing scheme. The additional experiments revealed that the BTB on Lion Cove is indexed using the entry-point of the branch, rather than the branch address itself. To assess if the branch address is still involved, we configured the test suite with a training and victim branch having an aliasing entry-point but a mismatching branch address while evicting the iBTB. Results show that the injected target is still prefetched but not speculatively executed—indicating that (partial) branch address information is stored in a separate BTB field which is validated before speculative execution. From a predictor's point of view, it is natural to include information on the branch address location inside the BTB entry, since the prefetcher needs this information to know until which address it should prefetch before redirecting to the next target.

Implications. As the indexing is performed with the entry-point address, we hypothesize that one single branch now can occupy multiple BTB entries. We configure our test suite with two training direct branches having distinct entry points. Next, we execute the victim matching the entry point of either one of the branches. The results show that the prefetcher successfully captures the correlation between the entry point and the target to prefetch, confirming a single branch can use multiple BTB entries. As a result, this design may also reduce the impact of the BHI_NO mitigation. Namely, while BHI_NO forces the first few branches in a privileged domain to only use the BTB for prediction, the BTB can now store a target for each entry point of the indirect branch. Moreover, performance-wise, the BPU may resolve branch targets earlier as the lookup only requires the previous BTB entry. As a side effect, there is likely a higher pressure on the BTB as branches can have multiple BTB entries.

Appendix C.
ITS cBPF Gadget Chain

Listing 1: Gadget chain used in the ITS cBPF exploit.

```
security_mmap_addr+37:
  mov    rax, QWORD PTR [rbx+0x18]; load target
  mov    rdi, r12
  call   rax  ; call attacker-controlled target
  ...
cmp_entries_key+23:
  mov    rbx,QWORD PTR [rdx]  ; load secret addr
  mov    rsi,QWORD PTR [rcx+0x20]
  mov    eax,DWORD PTR [rbx+0xac] ; load secret
  shl    rax,0x4
  add    rax,QWORD PTR [rdx+0x8]  ; load base
  mov    edi,DWORD PTR [rax+0x8]  ; transmit
```

Appendix D.
Native ITS Case Study

As a case study, we discuss a native (no cBPF) ITS collision resulting in a dispatch gadget where both the training (direct) branch, the victim (indirect) branch as well as the dispatch gadget are present in the same function `do_vfs_ioct`. Listing 2 presents the assembly code of the gadget, including both the training and victim branch. The collision is present on both Comet Lake and Sunny Cove. Moreover, since both branches are within the same page, the collisions is KASLR-invariant. We manually verified what the attacker controls at the victim branch and the reachability of both the training and victim branch.

An attacker can train the victim branch (Line 15) by taking the conditional branch (Line 22) which fills a BTB entry to the dispatch gadget (Line 2). Upon execution of the victim, the CPU mispredicts to the dispatch gadget and, as `rbx` is attacker-controlled at the victim branch, the attacker can speculatively jump to an arbitrary location (Line 4). Although the victim allows the attacker to control 4 registers, they originate from the same register and are thus not independently controllable. For exploitation, the attacker can first train the dispatch gadget in transient execution and then jump to a matching disclosure gadget in the next iteration [48]. Alternatively, they can also resort to loading data from the stack (e.g., the pointer to the `pt_reg` struct).

Appendix E.
In-place IP-based attacks

An in-place IP-based attack confuses valid targets for a given indirect branch. This can potentially allow an attacker to exploit speculative type confusion, perform out-of-bounds

Listing 2: Assembly of the dispatch gadget `do_vfs_ioctl`, suitable for native ITS exploitation.

```
do_vfs_ioctl+1754:
  mov    esi,0x2 ; dispatch gadget start
  mov    rdi,rbx
  call   rdx     ; call attacker-chosen target
  nop    DWORD PTR [rax]
  mov    r12d,eax
  jmp    <do_vfs_ioctl+129>
  mov    rax,QWORD PTR [rbx+0xb0]
  mov    rax,QWORD PTR [rax+0x48]
  test   rax,rax
  je     <do_vfs_ioctl+1422>
  mov    rdx,rcx
  mov    esi,0x541b
  mov    rdi,rbx
  call   rax                ; victim branch:
  ↪ present on the lower half of the cache
  ↪ line. If trained via ITS, it mispredicts
  ↪ to +1754
  nop    DWORD PTR [rax]
  mov    r12d,eax
  cmp    eax,0xfffffdfd
  jne    <do_vfs_ioctl+129>
  jmp    <do_vfs_ioctl+1422>
  test   rdx,rdx
  jne    <do_vfs_ioctl+1754>; training branch:
  ↪ present on the upper half of the cache
  ↪ line. If taken, it fills the BTB entry
  ↪ to the dispatch gadget
```

transient loads, etc. To investigate this attack surface, we considered the victim branches as well as their possible valid targets from the dynamic analysis of Section 7. For each victim branch, our dynamic analysis evidenced different instances (i.e., from different kernel paths) with a different set of controlled registers. For each indirect branch instance, we instruct InSpectre Gadget to model the controlled registers and run on all the possible valid targets. Our results reported no exploitable gadgets across indirect branches.

While our analysis is simple and based on a dynamic analysis underapproximation, our results suggest kernel indirect branches exhibit limited type-polymorphic behavior across valid targets. To confirm this intuition, we created a LLVM pass to compare function signatures (number and types of arguments) across the valid targets of the branches we considered. Our results evidenced only one case of a signature mismatch across valid targets—i.e., functions `uevent_show` and `type_show` with different types for one argument (`struct kobject` vs. `struct device`)—confirming a limited degree of polymorphism.

Appendix F. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

The paper presents three new Spectre attack variants that can bypass existing domain isolation defenses. The authors demonstrate that in-domain training can be leveraged to bypass speculative execution mitigations without requiring cross-domain interference.

F.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Addresses a Long-Known Issue

F.3. Reasons for Acceptance

1) The paper identifies an impactful vulnerability by demonstrating that domain isolation techniques fail to prevent speculative execution attacks, highlighting a fundamental gap in existing defenses.
2) The paper provides a valuable step forward in an established field by reverse engineering CPU branch prediction structures and mitigations, providing valuable insights and allowing a deeper understanding of speculative execution vulnerabilities across multiple architectures.
3) The paper addresses a long-known issue where it builds on prior work on transient execution attacks, extending the attack surface by showing how in-domain training can circumvent existing protections, making this a significant contribution to Spectre research.