Breaking the Barrier: Post-Barrier Spectre Attacks

Johannes Wikner
ETH Zurich

Kaveh Razavi
ETH Zurich

***Abstract*—The effectiveness of transient execution defenses rests on obscure model-specific operations that must be correctly *implemented* in microcode and *applied* by software. In this paper, we study branch predictor invalidation through Indirect Branch Predictor Barrier (IBPB) for x86 processors, which is a cornerstone defense against cross-context and cross-privilege Spectre attacks, and discover new vulnerabilities in both its microcode implementation and application by software. Concretely, we demonstrate two new post-barrier speculative return target hijacks on Intel and AMD CPUs. First, we show an end-to-end cross-process attack that leaks the hash of the root password from a suid process. This attack works despite IBPB on recent generations of Intel processors due to a microcode implementation flaw. Second, we show that an unprivileged attacker can leak privileged memory on AMD Zen 1(+)/2 processors despite the deployed IBPB mitigation, due to how IBPB is applied by the Linux kernel. We propose using a chicken bit to disable exploitable return predictions on affected Intel CPUs and a software patch for the Linux kernel to safely use IBPB on affected AMD CPUs.**

1. Introduction

Despite many years of mitigations after the original Spectre attacks [1], new variants continue to appear [2], [3], [4], [5], [6], [7], [8], [9]. A complete flush of exploitable branch predictions, provided by the Indirect Branch Predictor Barrier (IBPB) on x86, is considered to fully mitigate these attacks. As such, it is currently deployed on major hypervisors and operating systems to mitigate Branch Target Injection (BTI)-based Spectre attacks that leak information across execution contexts and privilege levels. Unfortunately, as we show in this paper, both the *implementation* of IBPB in microcode and its *application* by privileged software suffer from vulnerabilities that enable attackers to leak sensitive information in a variety of scenarios.

Alternative branch predictions. Branches of different types can be provided either primary or alternate branch target predictions. For example, return target predictions primarily originate from a dedicated Return Stack Buffer (RSB), and indirect branch target predictions primarily originate from a path history-indexed predictor table that considers the most recently taken branches. When the primary predictor is unable to produce a branch target, an alternate prediction may be served instead. As an example, indirect branches may be provided a prediction solely based on the current instruction pointer (i.e., IP-based prediction) when a path-based prediction is unavailable [10], [11]. Furthermore, when a return target prediction from the RSB is unavailable, the Restricted RSB Alternate (RRSBA) predictor found in many Intel CPUs may provide an alternate prediction [12]. Given that Spectre-BTI attacks can poison predictions of different predictors, it is important that branch prediction barriers adequately invalidate not only primary predictors, but alternate predictors as well. How can we trigger alternate predictions for different branch types and are all these predictions adequately invalidated by IBPB?

Discovering alternate branch predictions. Our analysis shows the most recent Intel CPUs serve IP-based predictions as an alternative to path-based predictions for indirect branches. This is the case when targets of such indirect branches have no correlation with the global path history. Furthermore, we observe that RRSBA predictions trigger not only when ascending from deep call stacks, leading to RSB underflow [3], [8], but also if RSB entries have been invalidated (via IBPB). We also find that RRSBA employs IP-based and path-based predictions depending on path history, similar to indirect branches.

Branch target invalidation. IBPB prevents previously learned indirect branch target predictions from being used. However, IBPB has implementation-specific guarantees on various microarchitectures. For example, since return instructions have similar semantics as indirect branches, they are invalidated by IBPB on most processors. It has previously been reported that Intel processors continue to serve the top-most RSB entry despite invalidation, known as EIBRS Post-Barrier Return Stack Buffer (PBRSB) speculation [2], [13]. Moreover, it is known that AMD Zen 1(+)/2 processors retain return predictions post-barrier [2]. Although hypervisors and operating systems already mitigate post-barrier return speculation attacks [14], it is unclear whether these mitigations consider new attack surfaces exposed by phantom speculation [6], [9].

IP-based predictors are also sometimes exempt from invalidation through IBPB on many processors, for example, when serving direct branches. On AMD Zen 3/4 processors IBPB flushes the IP-based predictor to mitigate Inception [9], [15]. It is, however, unclear how such predictors behave on recent Intel CPUs under IBPB given that such processors do not transiently execute instructions under phantom speculation [6] nor allow RSB corruption in transient execution [9]. Should IBPB not adequately flush

the primary and alternative predictors for particular branch types such as indirect branches or returns, the systems that deploy these CPUs are exposed to previously unexplored attack surfaces.

Alternative branch predictors under IBPB. We systematically explore the behavior of IBPB for different branch types, in particular their alternate predictions. We find that while IBPB flushes both path-based and IP-based predictors for indirect branches, it fails to flush alternative IP-based predictions for return instructions. We call this new primitive Post-Barrier RRSBA (PB-RRSBA). PB-RRSBA allows an attacker, who can force the victim to use these vulnerable IP-based RRSBA predictions, to trigger stale branch predictions after IBPB. Unlike PBRSB speculation [2] and Retbleed [8], which exploit RSB and path-based predictions, PB-RRSBA speculation exploits IP-based predictions of returns.

Exploiting the implementation of IBPB. The perhaps most studied threat model in Spectre attacks considers *same-context* attacks, for example where an unprivileged attacker infers secrets from a more privileged victim, the OS kernel. This threat model overshadows the lesser studied one of *cross-context* attacks. To prevent information leaks across execution contexts where the attacker and victim run in the same privilege level, IBPB on context switch is conditionally used. We show that PB-RRSBA can be used in cross-context attacks to leak information despite IBPB, which we demonstrate by leaking the root password hash from a suid process. Our evaluation of the `RRSBA_DIS_U` chicken bit shows that it mitigates PB-RRSBA with a negligible performance penalty.

Exploiting the application of IBPB. Linux also offers IBPB-on-entry as the more comprehensive mitigation against cross-privilege attacks for AMD processors vulnerable to Retbleed [8] and Inception [9]. Furthermore, IBPB-on-entry is the only available mitigation against phantom speculation on AMD Zen 1(+)/2 processors [6]. However, we show that the IBPB-on-entry is still vulnerable to Inception attacks, due to how the mitigation is applied. In particular, we show that the instructions that execute before IBPB can be hijacked to mistrain the return predictor which retains its entries despite IBPB. This allowed us to build PB-Inception, a post-barrier Inception exploit that bypasses the IBPB mitigation on AMD Zen 1(+)/2, leaking arbitrary kernel memory to an unprivileged attacker. We propose a software patch to the Linux kernel that mitigates PB-Inception by stuffing the RSB with a negligible performance overhead.

Contributions. The following lists our contributions:

- We introduce PB-RRSBA, a vulnerability in the IBPB implementation on Intel Golden Cove and Raptor Cove (12th–14th generation) that lets us exploit IP-based RSBA predictions post-IBPB.
- We show that PB-RRSBA can leak sensitive information from IBPB-protected suid processes. As an example, we use PB-RRSBA to leak the root hash password from a helper suid process of polkit.
- We show that IBPB-on-entry in Linux is vulnerable to a new attack, called PB-Inception, where the return predictor is mistrained pre-IBPB and the poisoned return targets are retained post-IBPB. Our end-to-end exploit can leak arbitrary kernel memory on AMD Zen 2.
- We propose, implement, and evaluate mitigations against these attacks on vulnerable processors.

Responsible disclosure. The issues were disclosed to Intel and AMD in June 2024. Both confirmed their respective issues. Intel informed us that their issue, tracked under INTEL-SA-00982, had been found internally and fixed in a microcode update. This microcode update was, however, not available in Ubuntu repositories at the time of writing this paper. Because AMD's issue was previously known and tracked under AMD-SB-1040, AMD considers the issue a software bug. We are currently working with the Linux kernel maintainers to merge our proposed software patch. We will make the source code for all the experiments available at https://comsec.ethz.ch/breaking-the-barrier.

2. Background

This paper concerns the branch prediction barriers used in x86 processors, known as Indirect Branch Prediction Barriers (IBPBs), their various implementations and configurations, their particular vulnerabilities, and how they can be exploited. Before discussing IBPB, we will give some background on branch predictors, in particular branch *target* predictors used in x86 processors (Section 2.1). Then, we discuss how attackers exploit these predictors, making them produce attacker-controlled mispredictions to enable information leakage in various settings, and available mitigations like IBPB and its already-known issues that incite this research (Section 2.2). Finally, we discuss the necessary code sequences (i.e., gadgets) that are used as building blocks of Spectre attacks (Section 2.3).

2.1. Branch Prediction

Branch prediction is a vital feature to fully utilize the capacity of superscalar processor pipelines with out-of-order execution backends. Processors predict branch *direction* for conditional branches (i.e., taken or fall-through) using Pattern History Tables [16], and return target predictions are primarily provided by a Return Stack Buffer (RSB) to which return targets are pushed and popped onto upon function calls and returns [3]. Branch *target* address for all taken conditional or unconditional branches are predicted via Branch Target Buffers (BTBs) [6]. Branch target predictions are continuously learned and improved throughout program execution by observing and recording branch targets associated with their respective branch sources in BTBs.

Path-based branch prediction. To predict the correct branching outcome, predictions are associated with the current execution path history preceding the branch source.

Like conditional branches that can have one or two possible outcomes, indirect branches can have one or several possible branch targets. On Intel processors, path history is recorded in a global (yet thread-local) path history register, which x86 refers to as the Branch History Buffer (BHB) [7], [16]. The prediction outcome that best correlates with the current path-history then be forwarded when predicting the current branch. Most modern processors implement a TAGE-like branch predictor for conditional branch prediction [16], [17]. A TAGE predictor can also implement branch target prediction for indirect branches [18].

IP-based branch prediction. Whereas path-based predictions are important for indirect branches with many targets, branches that only have a single possible target can be adequately predicted without path history. Such predictions are based only on Instruction Pointer at the branch source, thereby consuming less space and are faster to resolve. On AMD processors, this includes direct branches, always-taken conditional branches and indirect branches that have only been with a single target [19]. Older Intel microarchitectures can also predict indirect branch targets using IP-based prediction. The IP-based branch target predictor can be identified both by how it indexes the BTB and how it only serves a portion of the branch target address [11] (the rest of the address is assumed to be the same as the branch source address). The more modern Intel Golden Cove microarchitectures claims that branch target predictions of targets within 2 GB proximity use less predictor resources [20]. Does it mean that modern Intel processors use an IP-based predictor as well, and if so, can we distinguish it from the path-based predictor?

Alternating predictors and cold starts. Branch predictors that use long path histories suffer more mispredictions compared to simpler predictors under cold starts and short-lived workloads. A cold start for the branch predictor means that insufficient branching feedback has been collected to make an accurate prediction. To combat this, meta predictors like ALPHA 7's could alternate between predictors to forward predictions from the currently best performing predictor, where occasionally a simpler predictor is more accurate [21]. Similarly, with a Cascaded predictor [22], a simpler static predictor could be used for easy-to-predict branches, avoiding pollution of more complex path-based predictor tables. State-of-the-art TAGE predictors benefit from Statistical Correctors that forward an alternative prediction when outcome and history do not correlate [23].

Return target prediction. Return target predictions are primarily forwarded from a stack that we refer to as Return Stack Buffer (RSB). Predictions are pushed onto the RSB on call instructions and popped from it on return instructions. For call stacks depths greater than the RSB capacity, the oldest RSB entries are overwritten before they can be used, eventually leading to an empty-state condition or underflow when ascending the call stack [3], [8], [24]. In this situation, Intel processors alternate return target predictors (RSBA), where the alternative is branch target prediction [8].

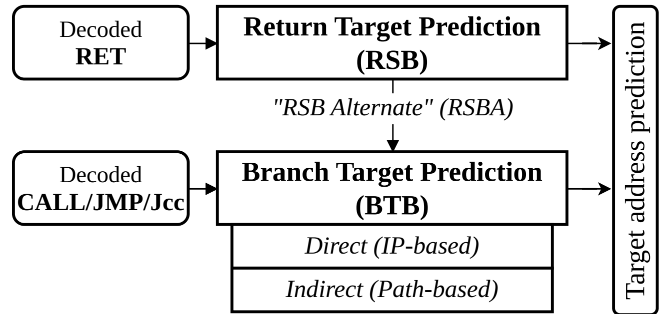

Figure 1: Branch target prediction alternates between IP-based and path-based. Furthermore return target prediction alternates between RSB and branch target prediction.

This behavior is also sometimes described as a "bottomless RSB" [25]. Modern return target predictors furthermore ensure that the fallback predictor respects privilege-level restrictions of predictions (i.e., eIBRS), known as Restricted RSBA (RRSBA). Given that branch target prediction, in turn, alternates between IP-based and path-based predictions, return target prediction becomes particularly complex. Figure 1 summaries our current model of return target prediction.

2.2. Spectre

In Spectre attacks, the goal of the attacker is the provoke a branch predictor to forward an incorrect prediction controlled by the attacker. Taking an incorrect prediction, the processor transiently executes incorrect instructions to make out of bounds memory accesses to secrets in memory. If these secrets then are operated on by subsequent transient instructions, the effect of the misprediction is *secret-dependent*, such that the secret can be inferred by the attacker through side channels. Such secret-dependent side channels include CPU cache accesses [26], [27], [28], floating point unit use [29], execution port use [30], power fluctuations [31] and branch predictor updates [32], [33], [34]. Upon branch misprediction, the number of mispredicted instructions is bounded by the reorder buffer capacity of the CPU backend, and by the latency until the misprediction is detected. The misprediction is detected and corrected as soon as all the dependencies necessary to conclude the correct branch outcome are resolved. For example, when these dependencies include memory operations, they may resolve with DRAM access latency.

Cross-privilege attacks. Because branch prediction state is shared across privilege levels, Spectre breaks the security boundary between privilege levels, for example between user mode processes and the operating system kernel [1], [7], [8], [9], [35]. In particular, an unprivileged attacker can trigger branch mispredictions in the kernel. Enhanced and Automatic Indirect Branch Restricted Speculation (eIBRS and AutoIBRS) restrict the branch predictor from forwarding indirect branch target predictions learned in a less privileged domain, reducing the attack surface for this attack

substantially [36], [37]. However, later work has found methods to evade eIBRS [7], [35] and AutoIBRS [9]. In addition to enabling eIBRS/AutoIBRS, operating systems may prevent the use of indirect branch predictions through software constructs [15], [25], [38], [39]. As a more comprehensive defense for AMD processors, in particular against [6], [8], [9], Indirect Branch Prediction Barrier (IBPB) is employed at system call, interrupt, and VMExit entry points to a more privileged domain [15], [40], [41].

Cross-context attacks. Branch predictions are also shared across execution contexts of the same privilege level. Although a victim process runs under the same privilege level as the attacker, it may for example be owned by the privileged root user to manage security sensitive data, like passwords. Considering this threat model, where IBRS mitigations are ineffective, exploitable branch predictions must be invalidated upon context switch. In x86, this is achieved using branch prediction barriers, most notably, IBPB. In addition to IBPB, Linux scheduler also employs RSB stuffing on context switch, preventing cross-context misuse of return predictions by filling the RSB with harmless targets.

Phantom speculation. Recent work shows that modern x86 processors perform branch prediction before an instruction is decoded [6]. This enables transient execution attacks on non-branch instructions, referred to as PhantomJMPs. PhantomJMP enables an attacker to redirect the speculative control flow to an arbitrary program address. Inception leverages the short speculation window provided by a PhantomJMP to perform a recursive PhantomCall [9] which poisons multiple RSB entries with the address of the following instruction. A return instruction following the PhantomJMP then consumes a poisoned RSB entry that can leak arbitrary information given the long return speculation window. While Inception can be mitigated in software using a specialized instruction sequence [41], a more comprehensive mitigation relies on IBPB. Furthermore, PhantomJMPs enable an extra memory load that can leak arbitrary memory with MDS-like gadgets [6]. The only known mitigation against all forms of phantom speculation on AMD Zen 1(+)/2 is IBPB on privilege transitions.

Indirect Branch Prediction Barrier (IBPB). IBPB is a defense against a range of Spectre V2 attacks, including cross-privilege mode attacks on AMD processors [8], [9] and cross-context attacks [42]. IBPB prevents forwarding of previously learned indirect branch target predictions for speculative execution. IBPB can be used to complement eIBRS/AutoIBRS in scenarios where those are ineffective [43]. This includes all scenarios where the attacker can inject predictions under the same privilege level as the victim. Whereas the IBPB routine alone can cost over 8 000 cycles on older hardware, excluding the cycles wasted on cold-start mispredictions, more recent implementations of IBPB are about an order-of-magnitude faster [9]. If enabled by its programmer, Linux uses IBPB before and after scheduling of a process to protect it against Spectre-BTI attacks. Hypervisors like KVM will always issue IBPB when switching between vCPUs.

Because returns are semantically similar to indirect branches, return predictions are commonly invalidated by IBPB as well. There are however exceptions to this rule, especially concerning returns. On AMD Zen 1 and Zen 2 processors (i.e., AMD family 17h) IBPB disregards return targets [2]. Certain Intel processors [44] retains a single return target prediction after IBPB, known as eIBRS Post-Barrier Return Stack Buffer (PBRSB) [13]. For both cases, exploitable return targets must be overwritten by a software routine. Post-barrier branch prediction exceptions have not been comprehensively studied. We will fill this gap in Section 5 by analyzing the speculative behavior of post-barrier return instructions under different conditions. In doing so, we find additional exceptions that expose new attack surfaces.

2.3. Spectre Gadgets

In code-reuse attacks, the attacker makes use of small snippets of code in the victim's execution domain, referred to as gadgets. Typically, the attacker forces the victim to execute these gadgets by exploiting a memory error vulnerability, taking full control over the victim. In Spectre attacks, the attacker does not rely on memory error vulnerabilities, or any type of software bug. Instead, the attacker forces a misprediction on a *victim branch*, transiently taking control over the victim. This transient control, called *speculation window*, lasts until the processor receives the correct branch outcome and redirects the execution path to the architecturally-correct branch destination. Such transient control-flow hijacks have been shown in the past on indirect and conditional branch prediction [1], [7], [17], and return target prediction [3], [4], [8], [9], [24]. We consider the following gadgets in this paper:

IC-gadget. An Instruction Cache-signal gadget (IC gadget) contains arbitrary executable memory. If branch target prediction provides the address of the IC gadget, the processor frontend speculatively fetches it from memory into the instruction cache. Consequently, loading the gadget over the instruction path will be observably faster, but also the over the data path, even if the system has non-inclusive L2 and LLC caches. This is because of various reasons, such as an inclusive shared TLB, the gadget evicted from L1i to L2, or undocumented snooping behaviors [1]. Besides tainting the instruction cache, the IC-gadget also leaves a TLB signal. As we will see, this is important for the attacker, since a TLB miss can be fatal for a successful exploit.

DC-gadget. A Data Cache-signal gadget (DC gadget) is a code snippet that contains a memory load. The memory address depends on a given general purpose register that is set prior to its execution. We can use this property to detect that a misprediction resulted in speculative execution.

1. L1i presumably snoops on L1d writes to accommodate self-modifying code.

Leak gadget. A leak gadget transmits secrets accessed during transient execution. They consist of at least two dependent memory operations, where the first operation loads the secret and the second operation subsequently uses the secret in a parameter (e.g., index), resulting a unique data cache footprint for every possible secret value.

The Flush+Reload side channel [26] enables us to 1) detect if an IC-gadget or DC-gadget were transiently executed by the processor, and 2) leak information with the leak gadget.

3. Threat Model

We consider a realistic threat model in which an unprivileged attacker with local code execution wants to infer secrets from the system. We assume that the system has no known software vulnerabilities and runs all the latest security updates. For the exploits, in Section 7, we assume that the victim software is publicly available (e.g., via the package repositories of the Linux distribution), so that the attacker can analyze it in an offline stage. We further assume the victim software runs on top of a CPU with either the Intel Golden Cove or Raptor Cove microarchitecture. In Section 8, we assume that the victim kernel runs on top of a CPU with the AMD Zen 1(+)/2 microarchitecture. We also assume that the attacker is able to analyze the victim kernel offline. For example, as in this case, the victim runs a generic Ubuntu kernel, which the attacker can identify and download to analyze for gadgets.

4. Overview of Challenges

Our aim is to understand CPU behaviors with post-barrier speculation, with a particular focus on return target predictions. We hence need to first be able to trigger and identify a particular branch predictor and then verify that its prediction gets invalidated by IBPB. This brings us to our first challenge:

> **Challenge C1.** Forcing the branch target predictor to alternate between path-based and IP-based predictions.

Section 5 experiments with the conditions under which we can force IP-based predictions with indirect branches, even though they are typically served by the path-based predictor. We find that randomizing the address of a single branch preceding to the victim indirect branch is sufficient to force IP-based instead of path-based predictions. Furthermore, we find out that IBPB invalidates these predictions, regardless of whether they are served by the path-based or IP-based predictor. Our next challenge is forcing alternative return predictors to take IP- and path-based predictions, and if IBPB is similarly invalidating these predictions.

> **Challenge C2.** Forcing the alternative return predictor to take either path-based or IP-based predictions.

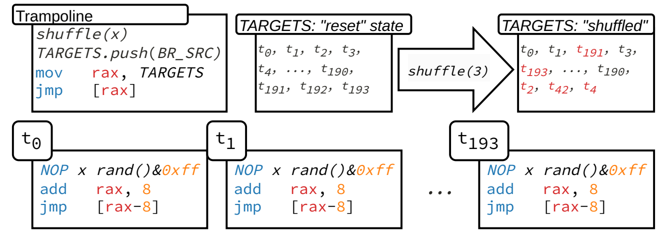

Figure 2: Execute a sequence (*TARGETS*) of basic blocks $t_{0\ldots193}$, to set the BHB to a "shuffled" state when executing the training or victim branch source (BR_SRC). Each t is of random size and jumps to the next block in the sequence. 194 blocks is sufficient to reset the BHB on Intel Raptor Cove. From *Trampoline*, we execute *TARGETS*, referenced by `rax`. Before that, we shuffle *TARGETS* by swapping x the *last* pointers for random ones in the sequence. The example shuffles 3 entries.

Section 6 shows that RRSBA predictions include IP- and path-based predictions and occur under RSB empty-state [8]. Similar to indirect branch predictions, we find that IP-based RRSBA predictions occur in case the preceding branch is randomized. More interestingly, we discover that *RRSBA IP-based predictions are not affected by IBPB*. We further analyze the behavior of the return predictor on AMD Zen 1(+)/2 CPUs under IBPB and discover that IBPB does in fact affect the behavior of the return predictor, but does not invalidate its entries. Armed with this information, the next challenge is showing exploitation in practical scenarios.

> **Challenge C3.** Practical exploitation of post-barrier return target predictions.

We conduct two case studies of the discovered issues with IBPB. In Section 7, we show a novel cross-context Spectre attack that can leak the root hash password from a suid process despite IBPB on affected Intel processors. This is the first known instance of a cross-context Spectre attack that works on a real suid target. Furthermore, in Section 8, we revive the Inception attack [9] on affected AMD processors.

5. Alternating Branch Predictors

We start by investigating how we can alternate between IP-based and path-based branch prediction on Intel processors. Details of the systems used for our reverse engineering experiments are provided in Table 3 of Section 9. We first construct the experiment for Intel Raptor Cove and adjust it for other microarchitectures. We want to test whether indirect branch targets are added to both the direct (IP-based) and indirect (path-based) branch predictors. We rely on a routine that resets the BHB by organizing and executing a sequence of basic blocks in a fixed order, setting the BHB into a "reset" state. A similar routine, shown in Figure 2, shuffles $x \in \{1, 2, \ldots, 8\}$ of the last basic blocks of this sequence before executing it. Each block has a randomized start address and size, made up of mostly `NOP` instructions.

As shown in Figure 3, ① we first execute a training branch at address A to a DC-gadget at address B. Then, we

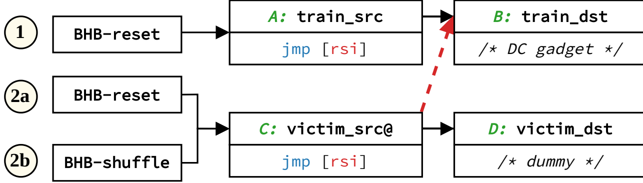

Figure 3: When the addresses A and C alias in the BTB, after (1), we can get mispredictions from C to B by taking step (2a), but also from (2b) if bits[47:32] of A and C match.

execute the victim branch at address C to a dummy target. C is different from A, yet aliasing with A in the branch predictor by inverting some of the bits in A, given by P (i.e., $C = A \oplus P$). Depending on microarchitecture, P may either set a high bit that is unused in BTB addressing [1] or two bits that cancel each other out by an XOR operation used in BTB addressing [8]. As expected, (2a) when the BHB at the victim and training branches match (i.e., "reset" state), we observe a DC-signal from executing the victim branch. Executing (1) again but then (2b) shuffling all the blocks of the BHB before the victim branch, we still observe the DC-signal. However, this signal appears only if C[47:32] match the training target B[47:32]. This characterizes the IP-based branch predictor, originally studied on Intel Haswell, where the lower 32 bits of IP at the branch source are updated [1]. This predictor has additional logic to handle branching over a 4 GB boundary, but cannot predict targets further away than about 6 GB.

Hence, if we were to separate A and B further than 6 GB from C, such that bits[47:32] of A and C mismatch, the predicted target in (2a) becomes $C[47:32] \parallel B[31:0]$. This allows us to put another DC gadget at this address, which will only be predicted by the IP-based predictor. In this setting, we repeat the experiment while shuffling increasingly more branches of the BHB, from the youngest to the oldest branch.

Results. We measure the number of hits of the two DC-gadgets and plot the results in Figure 4. The results show that as soon as the youngest branch preceding C is shuffled, the path-based predictor can no longer provide predictions. This coincides with for example a TAGE-like predictor that always looks for predictions that correlate with the youngest branches in the path history. As soon as two youngest branches of the BHB are shuffled, we start to see IP-based predictions. Depending on the exact microarchitecture, the prediction rate becomes almost perfect after two or four youngest instructions preceding B are randomized.

Observation O1. On Intel microarchitectures, we can alternate between IP- and path-based indirect branch predictors by shuffling only few preceding branches.

5.1. IBPB and alternating branch predictors

With the ability to choose which predictor should serve predictions for indirect branches, we want to find out whether IBPB effectively invalidates both the IP-based and path-based predictors. We perform the same experiment as before, but this time, we issue an IBPB in-between the training step ((1)) and the attack step ((2a) for path-based prediction and (2b) for IP-based prediction). The results of this experiment are shown in Figure 5, indicating that IBPB invalidates both predictors for indirect branches. The results in this section show that IBPB is an effective mechanism for protecting indirect branches while providing us with building blocks to experiment with other branch types, such as return instructions.

Observation O2. On Intel microarchitectures, IBPB properly invalidates indirect branch predictions from both path- and IP-based predictors.

AMD indirect branch target predictors. On AMD parts, it is documented that indirect branch predictions are serviced

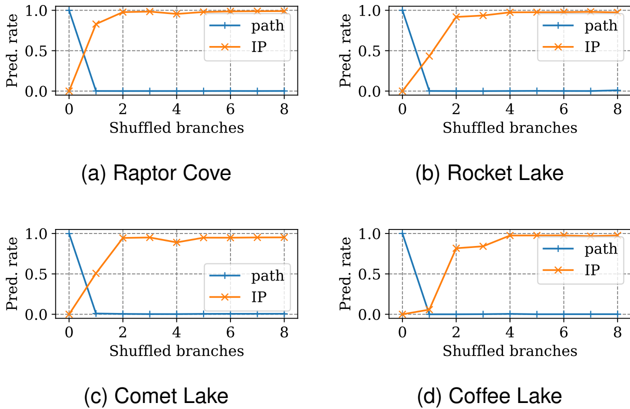

Figure 4: Transitioning from path-based (indirect) to IP-based (direct) predictions on various microarchitectures.

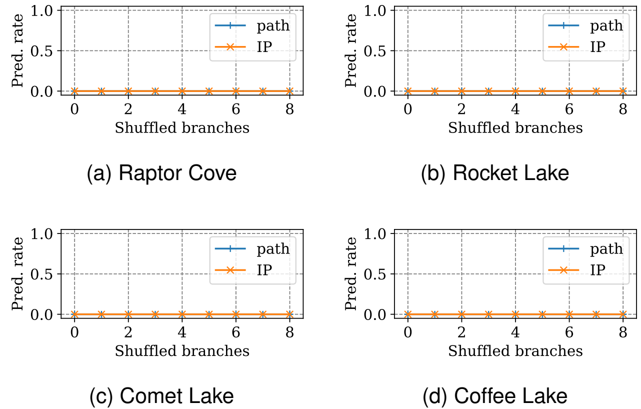

Figure 5: IBPB correctly invalidates both types of predictions for indirect branches.

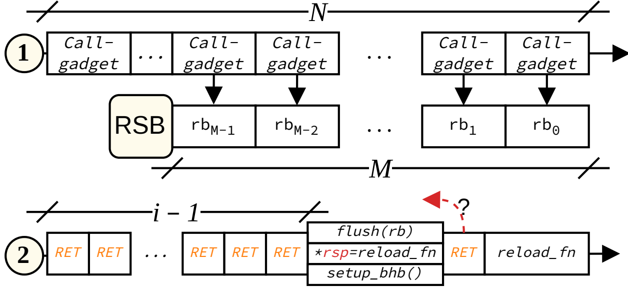

Figure 6: Part of the experiment design to observe RSB and RSBA predictions. ① Fill the RSB by executing a large number of calls. ② Execute $i-1$ returns, flush all reloadbuffer entries, overwrite the architectural return target (referenced by `rsp`), setup the BHB either for IP-based or path-based RSBA misprediction, and execute the ith return.

by a static predictor unless they have more than one target, upon which they are serviced by a dynamic predictor [19]. Unlike Intel processors, AMD processors provide separate performance counters for their static and dynamic predictors. We confirm that indirect branches are services by both static and dynamic predictors using these counters and that both predictors were correctly invalidated by IBPB.

6. RSB Alternate Predictions

Our next challenge is to find the conditions under which the return target predictor forwards branch target predictions. While it is well-known that the RSB empty-state results in branch target predictions on many Intel processors [8], other instances may lead to similar behavior. In particular, we want to understand the behavior of the return predictor when the current RSB entry is invalid because it belongs to a different privilege level or executed post-IBPB.

Tracking RSB predictions. We start by constructing a baseline experiment that showcases return target prediction from the RSB by assigning each RSB entry a different DC-gadget. The experiment is illustrated in Figure 6. We introduce a call gadget, which consists of a call and a DC-gadget. Executing its call adds its DC-gadget to the top of the RSB. Hence, we would observe a DC-signal by executing the next return, even if we overwrite the return address on the program stack. The RSB hosts M RSB entries. To detect M, ① we allocate all RSB entries by executing N call gadgets, where $N \gg M$. We let each DC-gadget i, load a distinct reloadbuffer entry rb_i, for $i \in \{0 \dots N-1\}$. To detect the RSB entry used by the ith return (if any), we check which DC-gadget emitted a signal after executing the return. ② Immediately before the ith return, we flush the entire reloadbuffer from the cache to disregard any signals from previous returns, and we overwrite the return address on the program stack so that a DC-gadget cannot execute architecturally. We can then precisely detect which RSB entry was used by the ith return by measuring access times to $\text{rb}_{0 \dots N-1}$.

RSBA predictions. The above procedure lets us explore the use of RSB entries, shown by the blue markers in the Figure 7. We introduce two additional reloadbuffer entries $\text{rb}_{N \dots N+1}$ to observe RSBA predictions from IP-based and path-based branch target predictors. To capture RSBA prediction, we train the targeted (i^{th}) return with a branch target prediction. We train this return by executing an aliasing return with a target distanced by more than 6 GB from the ith return. This distance makes the target of the ith return only predictable by path-based RSBA prediction, as we mentioned in Section 5. We put a DC-gadget touching rb_N at this return target. The IP-based predictor would instead predict only the lower 32 bits of this target, while retaining the upper bits of the return source address. We place another DC-gadget touching rb_{N+1} at that address.

As shown in Figure 6-②, to observe use of the IP-based branch predictor, we randomize the BHB before the ith return. To observe use of the path-based predictor, we set the BHB to the same state it had when training with the aliasing return before executing the ith return. For each i, we run the experiment twice: (1) with randomized BHB before the targeted return and (2) with the same BHB state as while executing the aliasing return. This experiment can measure three types of return target predictions for a given return: RSB, IP-based (direct), and path-based (indirect).

Results. Figure 7 shows a scatter plot over the RSB entry or branch target prediction being used (vertical axis) for every return executed (horizontal axis). To increase the visibility of faint signals (which also amplifies random noise), the size of each point is given by $\sqrt{\frac{c}{N}}$, where c is number of times a DC-signal was observed and N is the total number of times each return is tested. We see the following.

① **Last entry-reuse.** AMD family 17h parts (Zen 1(+)/2) have 31 RSB entries and do not alternate predictors. Instead, they tend to reuse one of the oldest entries, which could be useful for tail recursions. Occasionally, the RSB is halved, caused by sibling thread activity [9], resulting in noise from the 15$^{\text{th}}$ RSB entry. AMD family 19h parts (Zen 3/4) tend to reuse some of the oldest 4–16 entries.

② **RSBA with path and IP predictors.** Coffee Lake, Golden Cove, and Raptor Cove show RSBA predictions using both path-based and IP-based predictors.

③ **Possible stalling.** Rocket Lake does not show any RSBA prediction. Our experiment does not indicate whether returns stall or get predicted in a way that our method is unable to capture. This could explain why these parts are reportedly unaffected by RSBA bugs [44].

④ **Extra large RSB.** Gracemont has 128 RSB entries, likely to accommodate for the lack of a RSBA mechanism.

⑤ **RSB-wraparound.** Coffee Lake, Comet Lake, and Rocket Lake show complete RSB wraparound at every 64 or 128 returns, previously exploited on older processors [24].

⑥ **Single entry RSB-wraparound.** Gracemont, Golden Cove, and Raptor Cove show RSB wraparound for only the youngest entry.

⑦ **Random noise.** Golden and Raptor Cove exhibit random

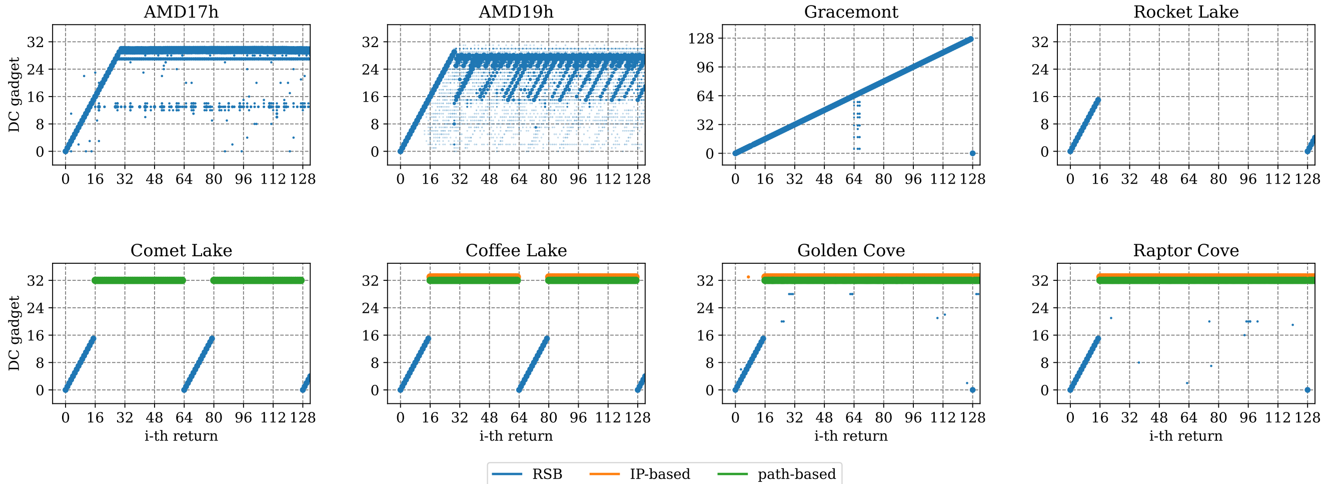

Figure 7: The AMD processors, Gracemont and Rocket Lake have no RSBA. The AMD 17h and 19h have 31 and 32 RSB entries, and attempt to predict one of the most recent return targets upon underflow. Gracemont has 128 entries and no RSBA. The bottom row processors all have RSBA. The Golden and Raptor Coves have similar RSBA behavior as Coffee Lake. Most processors have RSB wrap-around behavior, recycling either one or all stale RSB entries.

noise from various RSB-related DC-gadgets, which we do not study further. Similarly, Gracemont shows a weak signal from multiple DC-gadgets around 64 returns which we do not study further.

As we can see, return target prediction is a complicated procedure that varies across CPU vendors and generations. The key observation from the above that will be the primary focus of this paper is as follows.

Observation O3. Modern Intel processors predict return targets with at least three different predictors.

RSBA with non-empty RSB. We also check whether RSBA can trigger when the RSB still has valid entries, albeit potentially associated with incorrect privilege level. We perform an experiment which runs in privileged mode and is started via a system call. Our observation is that after 6 returns, we start to get RSBA predictions. This number of returns coincides with the number of calls made between the system call entry and the experiment entry point. However, we find that if we make additional calls in usermode, before running our experiment, the returns necessary to trigger RSBA increases linearly. If we disable Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP), we furthermore observe that the privileged returns use these usermode-added return targets. Furthermore, hypervisors must sometimes write to the speculation control Model-Specific Register (MSR) at VMEXIT to avoid using the RSB entries of the guest, even if eIBRS is enabled [43]. These act as indicators to the following observation:

Observation O4. RSB entries appear not tagged with privilege level.

6.1. Post-IBPB RSBA

Next, we want to know how RSBA predictions behave after IBPB. We execute the previously constructed experiment in privileged mode so that we can write to the MSR bit that triggers IBPB. We place this MSR write immediately after branch target training and call gadgets and compare the results with the previous results from Figure 7.

Results. Figure 8 shows the results once the mitigation has been applied before triggering any speculation. The processors that do not invalidate their return target predictions correctly are AMD family 17h, 19h, Golden Cove, and Raptor Cove. Although AMD 19h appears to invalidate the RSB entries, we still observe a faint signal ($\approx 0.01\,\%$ signal rate). Whether this effect can exploited, however, we leave for future work to investigate. On AMD family 17h, IBPB does not invalidate RSB predictions. However, unlike what has previously been reported [2], IBPB affects the behavior of RSB entry selection. After IBPB, *RSB appears to pick an arbitrary entry*, particularly after 12 returns, while having a bias towards using the oldest entry. Because the RSB is shared with the sibling thread on 17h, it is possible the sibling thread has an impact on the picked entry.

Observation O5. IBPB causes the RSB to pick an arbitrary entry on AMD family 17h.

On Golden Cove and Raptor Cove, IBPB invalidates all RSB entries and path-based predictions, but *IBPB fails to invalidate the IP-based predictions*. We refer to this new effect as Post-Barrier Restricted Return Stack Buffer Alternate (PB-RRSBA).

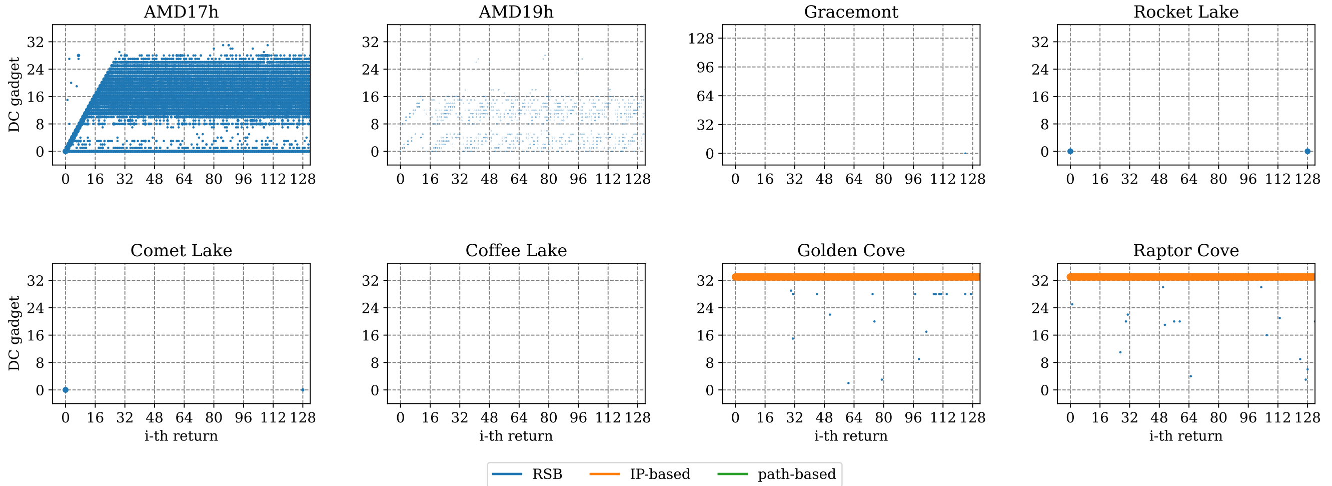

Figure 8: Post-barrier return speculation. AMD 17h does not invalidate return predictions. Comet Lake and Rocket Lake exhibit already-known PBRSB from the first return [2], [13]. A new insight is that this return re-appears every 128 returns. Our key insight is that Golden and Raptor Cove exhibit PB-RRSBA.

Observation O6. IBPB does not invalidate RSBA with IP-based predictions.

Another consequence of IBPB on Golden Cove and Raptor Cove is that RSBA prediction is immediately used from the first return on instead of after the 16$^{\text{th}}$.

Observation O7. RSBA triggers immediately after IBPB has invalidated all return targets.

These IP-based post-barrier return target predictions are not just easier to train, as they require no path history, but with IBPB, they are also triggered reliably if IBPB can undo any negative bias imposed by the path-based predictor.

Summary. We have learned that IBPB results in arbitrary RSB predictions on AMD Zen 1(+)/2 and it does not invalidate RSBA predictions on Golden and Raptor Cove. With the insights from our reverse engineering, the next two sections describe two attack scenarios on up-to-date Linux systems running on these processors.

7. Cross-process Attack

IBPB is supposed to prevent cross-process transient execution attacks. In this section, we demonstrate that cross-process attacks are not just practical, but that they are possible on the latest Intel processors despite IBPB, due to the PB-RRSBA effect. To this end, we build an end-to-end exploit that leaks secrets from a suid binary that is pre-installed on most Linux distributions. To do this, we moreover present a method to derandomize ASLR.

Branch predictor indexing. On Raptor Cove, we found that the RSBA-triggered IP-based predictor updates the lower 32 bits of the IP, based on the lower 24 bits of branch source. For example, when the victim executes the return at address A that leads to B, it will use A[23:0] to insert a target prediction with bits B[31:0]. When the attacker executes, if it executes a return from an address C where C[23:0] matches A[23:0], the branch predictor will forward D, where D[31:0] matches B[31:0]. D[47:32], however, will remain the same as the branch source C.

Vulnerable processes. Processes that handle sensitive information can enable IBPB to block cross-process Spectre attacks. Using the `prctl` system call ABI, the calling process can instruct the Linux scheduler to trigger IBPBs when rescheduling or preempting it. We can inspect the IBPB configuration of running processes by reading the *SpeculationIndirectBranch* value of their respective procfs files */proc/PID/status*. The values *conditional force disabled* and *conditional disabled* means IBPB is used, whereas *conditional enabled* means that it is disabled.

Upon inspection of the running processes on a Debian laptop, of 427 processes, the few processes that enable IBPB are renderer and IPC processes of Google Chrome. This suggests that application developers are either unconcerned with, or unaware of, cross-process Spectre attacks. Even suid binaries, like sudo, su, and polkit, do not use IBPB. Instead, to show that our cross-process attack is unaffected by IBPB, we enable it from our attacker process instead, which has the equivalent effect. These suid binaries manage privileged information, such as the password hashes from */etc/shadow*, and offer communication with the attacker via password prompts. Moreover, the attacker process can execute them and pin them to their own hardware thread to share BTB and cache, making them ideal victims for Spectre attackers.

7.1. Address Space Layout Randomization (ASLR)

As the attacker, to mount a Spectre attack, we need to derandomize ASLR to infer addresses of vulnerable branches

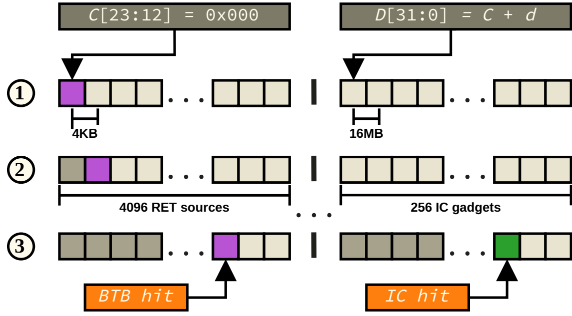

Figure 9: ASLR probing procedure in the attacker process. ① Execute a RET at $C[23{:}12] = 0$ and probe $C + d + 0 \ldots C + d + (255 \ll 24)$. ② Let $C = C + (1 \ll 12)$ and repeat. ③ Continue until C produces a BTB hit, observed through an IC hit, which occurs when $D[31{:}0] = B[31{:}0]$

and gadgets in the running victim process. ASLR randomizes the main binary, libraries, and the stack into different regions. For the libraries, our system randomizes bits[40:12], providing 28 bits of entropy. Previous work has shown that the BTB can be exploited for ASLR derandomization via toy examples, albeit with for us unfeasible methods, due to insufficiently recovered bits [34], the victim running an infinite loop [5], or the victim and attacker residing in the same process that uses a compile-time ASLR scheme [45].

We present a new ASLR derandomization technique, exploiting a return source A to return target B, of a victim process that reads characters from `stdin`. Our goal is to recover the lower 32 bits of B that is inserted in the BTB by the victim after executing A, by observing a return misprediction from C to D in the attacker process. By sending a character to its `stdin`, we trigger the victim to execute A. Because $A[11{:}0]$ is never randomized and $A[40{:}32]$ does not address the BTB, the BTB addressing culminates in 12 bits of entropy from $A[23{:}12]$. After victim read the character, our attacker process reschedules.

We will now infer $A[23{:}12]$ by guessing the aliasing branch C using the procedure in Figure 9. If we guess C correctly, the BTB forwards the target $B[31{:}0]$, updating lower 32 bits of the IP at C. We execute C from up to 4096 possible locations, until we hit the one aliasing with A. Assuming A and B are in the same region (e.g., libraries), the distance between the two is a constant d. Hence, if we guessed C right, $D[23{:}0]$ will be $C[23{:}0] + d$. Recovering $B[31{:}24]$, we assign IC-gadgets to all 256 possible branch targets. This means that for each C, we probe 256 possible values of D. If no IC was observed, we try another C by incrementing bits [23:12]. When guessed correctly, one of these IC-gadgets will be fetched, recovering $B[31{:}0]$ of the victim. We do not attempt to recover the 8 bits of remaining entropy, as they are ignored by IP-based branch prediction.

Victim binary. We analyze a few different suid binaries to find the most suitable for exploitation. The `su` binary refuses to read from `stdin`, unless it is terminal, making it difficult to trigger repeatedly. The `sudo` binary reads from `stdin` but does so from the main binary region. This allows

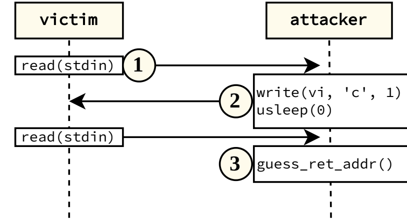

Figure 10: ASLR exploit procedure. `victim` and `attacker` run on the same hardware thread. ① `victim` reads from `stdin`. ② `attacker` writes an arbitrary character and reschedules `victim`. ③ `victim` receives the character and reads next character from `stdin`, which passes control back to `attacker`.

us to derandomize the main binary region, but not the much larger library region. However, a helper suid binary of polkit *polkit-agent-helper-1*, pre-installed in most Linux distributions, seems suitable as it reads from `stdin` from the library region. Polkit is an application-level toolkit for defining and handling the policy that allows unprivileged processes to speak to privileged processes. Its helper is invoked to authenticate an unprivileged user, for which it uses the Pluggable Authentication Modules library (libpam). libpam is responsible for reading */etc/shadow* via the glibc-provided *getspnam* function. The polkit helper presents a password prompt when run with the user to authenticate as in its first command-line argument and an arbitrary value in the second argument.

Exploitation. The exploit procedure is shown in Figure 10. The attacker process spawns the victim, pinning itself and the victim process to the same hardware thread to share cache and branch prediction state. ① After printing the prompt, the victim goes idle inside a `read` system call, initiated by an `fgets` call, awaiting a password. ② To trigger the victim to execute, the attacker process sends a character to its `stdin`, followed by a `usleep(0)`. The `fgets` function of the victim will then issue another `read`, since no newline was encountered and the input buffer is not full, passing control back. ③ At this point, the attacker process tries to recover $B[31{:}0]$ and ④ re-trigger the victim to execute if unsuccessful. The returns executed between each `read` are shown in Table 1. Any of the first three return branches that are highlighted in the table can be used.

TABLE 1: CALL AND RETURN TRACE OF THE `read` LOOP IN `fgets`.

Source	Insn.	Target	Object	Symbol
ffffffff81e001d2	sysret	7ffff7ccc7e2	kernel	entry_SYSCALL_64
7ffff7ccc7ea	retq	7ffff7c44c36	libc.so.6	read
7ffff7c44c6b	retq	7ffff7c45d96	libc.so.6	_IO_file_underflow
7ffff7c45db0	retq	7ffff7c3841c	libc.so.6	_IO_default_uflow
7ffff7c38417	callq	7ffff7c45a40	libc.so.6	_IO_getline_info
7ffff7c45d93	callq	7ffff7c44ab0	libc.so.6	_IO_default_uflow
7ffff7c44bf0	callq	7ffff7c45720	libc.so.6	_IO_file_underflow
7ffff7c457a0	retq	7ffff7c44bf5	libc.so.6	_IO_switch_to_get_mode
7ffff7c44c33	callq	7ffff7c43930	libc.so.6	_IO_file_underflow
7ffff7ccc7e0	syscall	ffffffff81e00040	libc.so.6	read

7.2. Cross-process Memory Leak

To leak memory from the victim, we hijack the speculative control flow at one of the exploitable returns in the victim using a ROP-gadget chain and leak a secret over a Flush+Reload covert channel. The secret is in our case the root password hash, and the first challenge is to invent a method to leak it from the victim process. Because this attack relies on exploitable PB-RRSBA predictions in the victim, the RSB must be in the empty-state when the victim return executes. To reach this state, the call stack of the `read` system call needs to exceed the RSB capacity, which fits 16 targets on Raptor Cove. Reading from a normal pipe does not exceed this number, but assigning a socket to the `stdin` of the victim does.

Before presenting the password prompt for authentication, libpam already verifies that the password entry of the requested user is valid. The */etc/shadow* is read into a `FILE` stream buffer, the root entry is parsed and recorded in the pam context, after which the file is closed, freeing the buffer. When the victim reads the password from `stdin` with the `fgets` call, a new stream buffer is allocated for `stdin`. If the `stdin` is not a terminal, it will be handled like a file and, as such, allocated a buffer of the same size as the just-freed buffer. The heap allocator conveniently recycles this memory, resulting in the `stdin` buffer containing the previously freed */etc/shadow* contents. We let A and B be the return source and target on the third highlighted row in Table 1, where three registers (`rdx-1`, `rsi`, `r9`) reference the secret via the uninitialized `stdin` stream buffer. Moreover, `rax` holds the most recently entered character byte, making it attacker-controlled, and `rcx` references the syscall return target inside the syscall wrapper, 18 bytes into `read`. The next challenge is to establish a Flush+Reload covert channel between the victim and attacker process.

Covert channel. In cross-process Spectre attacks, unlike user-to-kernel attacks, the attacker and victim do not share address space. However, read-only mappings of shared libraries are physically shared across processes [26]. This means that code pointers into libraries, like `rip` and `rcx`, in fact reference shared memory that can be used as covert-channel medium if the attacker process loads the same libraries. A typical leak gadget loads a single byte of the secret and left-shifts it before using it as offset in a shared memory buffer. With `rcx` as shared memory base pointer, ascii secrets can be left-shifted up to 13 bits while still referencing shared memory within glibc. The final challenge is hence to find a leak gadget that uses one of shared memory pointers as covert-channel medium.

ROP-gadget chain. The goal is to dereference one byte of the secret, while using `rax` as byte index, to left-shift the byte by 8 to 13 bits, and finally use the result as memory offset for dereferencing `rcx`. Unsurprisingly, finding such a leak gadget is non-trivial. Instead, since we can control the return target predictions of any victim return instruction, we investigate the idea of using a gadget chain, where each gadget does one of these operations, followed by a return.

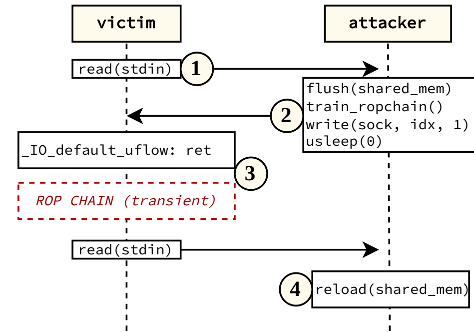

Figure 11: Leak exploit procedure. `victim` and `attacker` run on the same hardware thread. ① `victim` reads from `stdin`. ② `attacker` trains the BTB with the ROP chain, writes the character that `rax` should take in `victim`, and reschedules it. ③ `victim` receives the character and reads the next character, passing control back. ④ `attacker` infers the leaked byte.

This allows us to re-purpose classic ROP-gadget scanners, used in memory corruption attacks, for an advanced Spectre attack. Since we cannot trigger demand-paging with speculative execution, the memory at our disposal must already be physically backed. Hence, in an offline stage, we dump all executable pages of the victim process that are marked present. We then analyze the memory dump using the ROP-gadget scanner *rp++* [46].

TABLE 2: GADGET CHAIN

Gadget	Object	r-xp offset
`movzbl (%rax,%rdx,1), %eax; ret`	libicuuc.so.70.1	`0x005a69`
`rol $8, %eax; ret`	libc.so.6	`0x10e5b7`
`mov (%rcx,%rax,1), %al; ret`	libpcre2-8.so.0.10.4	`0x00074b`

The ROP-gadget chain we found is listed in Table 2. The pointer to the secret (offset by 1 byte) resides in `rdx`, and one byte from it is loaded into `eax` using the attacker-controlled `rax` as index. The secret is then left-rotated 8 times. Finally, the resulting value is used as index for referencing the shared memory pointer `rcx`. Rather than pushing each return target to the stack, as a typical ROP attack, the attacker trains each `ret` instruction to predict the next gadget in the chain. The leak procedure is shown in Figure 11. Since `rcx` points 18 bytes into `read` and will be used as shared memory for Flush+Reload, the attacker flushes it from the cache hierarchy. The attacker then trains A to predict the first ROP gadget, and connect each subsequent return of the ROP gadget chain. Finally, the attacker triggers the victim process to execute by writing a byte to it that corresponds to the index of the secret that they want to leak. The victim transiently executes the gadget chain when processing the input. Since the password input is not full, and no newline character was encountered, the victim waits for the next character, passing control back to the attacker process who infers the secret by reloading reloadbuffer entries relative to the `read` code.

Summary. We have described a method to leak cross-process information using our Spectre primitive. The major-

ity of binaries do not attempt at stopping such attacks and even if they did by using IBPB on context switch, IBPB on modern Intel processors fails at mitigating our PB-RRSBA Spectre primitive. Next we will study the security flaws of using IBPB in the cross-privilege setting.

8. Bypassing IBPB-on-entry

The most comprehensive mitigation for AMD processors that are vulnerable to recent Branch Target Confusion [8] and Phantom speculation attacks [6], [9] is to invalidate all exploitable branch target predictions using IBPB after transitioning to a higher privilege mode (i.e., user-to-kernel and guest-to-host) [6], [8], [9]. In this section we study the IBPB-on-entry mitigation and expose residual attack surface exploitable using our previous insights.

IBPB-on-entry is enabled by passing `retbleed=ibpb` or `spec_rstack_overflow=ibpb` boot parameter to the kernel. In this configuration, on kernel entry, the `entry_ibpb` function issues an IBPB as shown in Listing 1. After the function returns, the kernel code starts with all branch predictions invalidated. This means that no attacker-injected branch target predictions can be used, therefore securing the kernel against all kinds of BTI attacks — given that they do not occur *before* the IBPB.

```
1  entry_ibpb:
2  ffffffff8211eb40:  mov    ecx,0x49
3  ffffffff8211eb45:  mov    eax,0x1
4  ffffffff8211eb4a:  xor    edx,edx
5  ffffffff8211eb4c:  wrmsr
6  ffffffff8211eb4e:  jmp    __x86_return_thunk
7  ; ...
8  __x86_return_thunk:
9  ffffffff8213fbc0:  ret
```

Listing 1: `entry_ibpb` and `__x86_return_thunk` to be hijacked.

We are interested to see whether the speculative control flow can be hijacked before the IBPB. In particular, we want to see whether we can inject a phantom branch on the instructions in `entry_ibpb`, before the IBPB-triggering `wrmsr` instruction on Line 5. As we have seen, vulnerable AMD CPUs do not invalidate return target predictions, so any return target predictions injected before the IBPB will be retained. The Inception attack showed that phantom speculation can be used to inject sufficient return target predictions to overflow the RSB [9]. Therefore, we test whether we can inject such phantom speculation to overflow the RSB, consequently controlling the return target prediction for the return on Line 9.

8.1. Exploit procedure and new challenges

As the attacker, we inject a PhantomJMP from `entry_ibpb` to a target that is three bytes before a leak gadget. The number of bytes depends on the width of the PhantomCALL that the we inject at the PhantomJMP target. In this case, the PhantomCALL is injected using a three bytes wide `call r8` before the leak gadget. To set up a recursive call loop that corrupts the entire RSB, from the PhantomCALL source (i.e., the PhantomJMP target), we inject a call prediction to the address of the PhantomCALL itself. This way, as we execute `entry_ibpb`, we will predict a jump to a target where a recursive call is predicted. After the decode of the instruction in `entry_ibpb` where we have injected the PhantomJMP, this series of mispredictions is corrected, but we will have already overflowed the RSB. Hence, the return thunk from `entry_ibpb` speculatively executes the leak gadget. This method has been explained in finer detail in [9].

There is, however, a challenge that we need to address: injecting PhantomJMPs and PhantomCALLs from user mode triggers page faults. While this is not a problem in the original Inception attack, in our case the interrupt entry from user mode, triggered by page fault, immediately undoes the injected phantom branch, since interrupts entries are protected by IBPB too. Using Training in Transient Execution, initiated by RSB speculation (i.e., $\mathrm{TTE}_{\mathrm{RSB}}$) [9], we can suppress the page fault, using the following macro.

```
1  .macro TTE_RSB train_dst, train_src
2      call    1f           ; line 3 and 4 are never architecturally executed
3      mov     \train_dst, %r8
4      jmp    *\train_src   ; leads the training source, which will use %r8
5  1:  lea     2f(%rip), %rax
6      mov     %rax, (%rsp) ; overwrite return target
7      clflush (%rsp)
8      lfence               ; wait for real return target the be flushed.
9      ret                  ; speculatively returns to line 3
10 2: .endm
```

Listing 2: Macro for injecting PhantomJMP and PhantomCALLs with TTE. The speculatively executed indirect jump on Line 4 leads to the training call or jump

Another challenge is ensuring that attacker-provided inputs are served quickly to enable secret-dependent memory loads during the speculative window. We choose a leak gadget that reads attacker-provided input from the stack for this purpose. Because the general-purpose registers were just saved to the stack a few instructions earlier on system call entry, accesses to this memory will be served quickly by the store queue. Listing 3 shows our leak gadget.

```
1  ffffffff81998cc4:  mov   rdx, qword ptr [rdi + 0x28]
2  ffffffff81998cc8:  mov   rcx, qword ptr [rdi + 8]
3  ffffffff81998ccc:  mov   edx, dword ptr [rdx + 0x10]
4  ffffffff81998ccf:  add   edx, 8
5  ffffffff81998cd2:  add   rcx, rdx
6  ffffffff81998cd5:  mov   ecx, dword ptr [rcx]
```

Listing 3: Leak gadget found in `lpss_uart_setup`. Attacker-controlled memory referenced by `rdi`.

9. Evaluation

Our evaluation considers the various post-barrier primitives (Section 9.1), and the performance of the cross-process (Section 9.2), and PB-Inception attacks (Section 9.3).

Evaluation platforms. Table 3 provides the details of our evaluated platforms, including the processor models, microcode versions, distribution and kernel versions.

TABLE 3: EVALUATION PLATFORMS.

System	Distribution	Processor	Microcode	Kernel
CFL	Ubuntu 20.04.6 LTS	Core i7-8700K	0xf4	5.19.7
CL	Ubuntu 22.04.3 LTS	Core i7-10700K	0x59	5.15.0-91-generic
RL	Ubuntu 22.04.3 LTS	Core i7-11700K	0x59	6.2.0-36-generic
GC	Ubuntu 20.04.6 LTS	Core i7-12700K	0x32	5.15.0-92-generic
RC/GM	Ubuntu 22.04.3 LTS	Core i7-13700K	0x119	5.15.0-97-generic
RC-R	Ubuntu 22.04.3 LTS	Core i7-14700K	0x11d	5.15.0-97-generic
17h	Ubuntu 22.04.3 LTS	EPYC 7252	0x8301038	6.5.0-35-generic
19h	Ubuntu 22.04.3 LTS	Ryzen 5 5600G	0xa50000c	5.15.0-46-generic

TABLE 4: POST-BARRIER SPECULATION PRIMITIVES.

Pred. Type	CFL	GM	CL	RL	GC	RC	RC-R	17h	19h
Indirect	✗	✗	✗	✗	✗	✗	✗	✗	✗
Direct	✗	✓	✗	✗	✓	✓	✓	✗	✗ [2]
RSB	✗	✗	✓ [1]	✓ [1]	✗	✗	✗	✓	✓ [4]
RSBA-BHB	✗	✗	✗	✗	✗	✗	✗	– [3]	– [3]
RSBA-IP	✗	✗	✗	✗	✓	✓	✓	– [3]	– [3]

[1] The youngest RSB entry is retained [2].
[2] Microcodes before Aug. 2023 retain direct predictions [9].
[3] AMD parts do not alternate return target predictors.
[4] Zen 4 of family 19h is unaffected [47].

9.1. Primitives

Table 4 summarizes the evaluation of the primitives that we discussed in Sections 5 and 6 on various microarchitectures. In addition to RSB and RSBA predictions, we also investigated whether IBPB invalidates predictions for direct branches. We observe that on Golden Cove and Raptor Cove direct branch predictions are retained, but unlike PB-RRSBA, they are only speculatively fetched (not executed) after IBPB. Our conclusion is that Golden Cove and Raptor Cove have a more permissive implementation of IBPB.

9.2. Cross-process attack

ASLR. We evaluate the ASLR exploit against polkit-agent-helper-1 using the procedure presented in Section 7.1. We let the attacker process execute the victim binary and attempt to derandomize ASLR of its libraries up to 8 times before giving up. We measure the time from (re-)starting the victim until the procedure completes. As root, we compare the recovered address to the correct one via procfs. The procedure guesses the possible upper bits of the target (B_{guess}[31:24]) for each possible value of lower bits of the source (A_{guess}[23:12]). We repeat the experiment 1000 times, each time restarting the victim, re-randomizing its address space. Since we observe a strong false-positive bias when testing if the higher bits (i.e., B[31:24]) are 0, we ignore this value, missing 1/256 possibilities. Moreover, because there is a false-positive bias towards the most recently recovered value, from the previous invocation, we ignore this value as well. Our derandomization method had a 98.8 % success rate, with an average time of 0.40 seconds.

Memory leak. We now evaluate the memory leak procedure against polkit-agent-helper-1 that we introduced in Section 7.2. We evaluate the time it takes to leak 110 characters from the shadow entry of root. The password prompt fits at most 512 bytes (including newline and NULL-terminator), meaning we can only trigger the victim to execute our ROP chain 510 times before the buffer is full. At this point, we restart and derandomize ASLR of the victim again. Unfortunately, the procedure is ineffective without optimizations: attempting to leak a control value ('o' of "root" in the shadow entry) failed over a 24-hour-long test.

Optimizations. We consider two types of optimizations to combat this: making the gadget chain execute faster, through *prefetching*; or delaying the retrieval of the correct return target address, through *sibling noise*. By *prefetching*, we attempt to fetch the pages necessary for the ROP chain into the i-TLB, before the victim return executes. To do this, we train the returns preceding the victim return (see Table 1) to prefetch the ROP gadgets. By *sibling noise*, we attempt to induce memory noise with the intention of prolonging the time it takes until the misprediction is corrected. For this, the system needs to enable SMT, which it does by default. The sibling noise accesses every page of 256 MB of memory in an infinite loop and is pinned to the sibling thread of the attacker and victim processes. Both optimizations worked to observe an occasional signal matching the control value. However, because of the low success rate, the signal-to-noise ratio is high. To combat this, we reload no more than one cache line per page per interaction with the victim. Hence, to check all possible values of the secret requires 4 or 16 interactions with the victim, depending on if the secret is left-shifted by 10 or 8 bits, respectively. In addition, before we attempt to leak from the victim, we first ensure that we can correctly recover the control value, otherwise we restart the victim.

Results. With *prefetching*, we recover the secret in 17 hours, leaking 108 out of 110 characters correctly. We note that we cannot send a newline byte (ordinal value 10) to the victim, preventing us from recovering the 12th byte of the secret. Hence, we are guaranteed to have at least 1 error in the result. Using *sibling noise* was significantly more effective, allowing us to recover the secret in 22/116/180 minutes with 1/1/3 byte errors (min/median/max, 20 full runs). Using both prefetching and sibling noise, showed no visible improvement. Our results confirm that cross-process Spectre attacks are practical in a realistic setting, which is concerning since they are not considered a threat by most usermode programs.

9.3. IBPB-on-entry

We run the end-to-end attack of PB-Inception 10 times on two AMD 17h processors with Zen 2. For the gadget offsets, the attack first obtains the kernel image address by reading the unprivileged `/sys/kernel/notes` [48]. Finding a physmap pointer of the attacker-mapped reloadbuffer, necessary for covert channel, follows the same procedure as the original exploit [9]. As with the original Inception attack, the majority of time is spent on scanning the physmap region for the password hash, rather than actually leaking it. The results in Table 5 show that PB-Inception performs better

TABLE 5: PB-INCEPTION EVALUATION

Processor	Microcode	Mem.	Time To Shadow [min]			Bandwidth [B/s]		
			min	*median*	*max*	*min*	*median*	*max*
EPYC 7252	0x8301038	16 GB	0:59	10:17	27:08	128	236	292
Ryzen 5 3600X	0x8701021	8 GB	0:16	19:37	38:54	159	299	364

than the original attack, leaking root password hash in 10–20 minutes depending on system, with a bandwidth of 236–299 bytes per second.

The speed-up compared to the original exploit is likely because of two reasons. First, by training from user mode with TTE, we no longer trigger page faults while training. Secondly, we target a return very early in the entry path, before the kernel stack has been randomized. This means that we can know the page offset that will be used when entering the kernel and can evict these cache sets only.

10. Mitigation

We propose mitigations against PB-Inception on AMD 17h and PB-RRSBA on Intel Golden and Raptor Coves.

IBPB-on-entry. To mitigate Inception, Linux untrains returns on kernel entry and forces every return to mispredict [41]. One might think that resorting to this mitigation can mitigate PB-Inception. However, the reason why users enable IBPB-on-entry is not to mitigate Inception alone, but also because it is the *only* available mitigation against all forms of phantom speculation [6], [8], [39]. Instead, we will discuss how we can apply IBPB securely.

There are a number of entry points to the privileged domain that must be considered when mitigating cross-privilege Spectre attacks. In Section 8, we exploited the syscall entry point. However, a comprehensive mitigation must also consider the entry points assigned to the interrupt descriptor table (IDT entries) and the entry point from guest to hypervisor (i.e., during VMEXIT). Because IDT entries are taken in any privilege level, the mitigation should be conditionally applied depending on the current privilege level the interrupt occurred in.

Since we are exploiting the return in `entry_ibpb`, the naïve approach, to stop PB-Inception by avoiding this return, is unfortunately not a complete mitigation. This is because `entry_ibpb` is used in many different locations, some of which include functions with subsequent returns. Since the attacker can corrupt the entire RSB, any one of these subsequent returns could be hijacked. While the system call entry calls `entry_ibpb` directly, KVM and various IDT entries call this function at a deeper call depth.

A partial stuffing of the RSB, which has previously been used to mitigate PBRSB against the hypervisor [13], appears to mitigate our particular attack. However, since the seemingly arbitrary selection of RSB entry post-IBPB on AMD 17h (Observation O5) might be controllable by an attacker, we opt for a more robust mitigation that stuffs the entire RSB. To detect whether the processor correctly invalidates return target predictions or not, we add the detection of IBPB implementation, which AMD processors provide from the `cpuid` instruction. The mitigation is conditionally enabled through Linux's `ALTERNATIVE` macros, which rewrite the kernel at boot-time to toggle various features. The mitigation is enabled if the IBPB implementation ignores return target predictions.

We provide the patches we implemented in Appendix A. To test the performance overhead of the mitigation, we measure the number of `getpid` system calls we can perform in 60 seconds. This microbenchmark shows a 9.8 % overhead (66.3 M vs 59.8 M calls).

Mitigating cross-context attacks. For suid binaries that manage secrets, like the polkit helper, we recommend the developers to enable IBPB. A possible mitigation for operating system maintainers is to force all suid processes (i.e., that execute under a different user) to enable IBPB by default, forcing the developer to instead disable it if their process does not access secrets. To mitigate the PB-RRSBA effect on Intel processors, we propose use of a *chicken bit* that stops the cross-process leak. In response to the Branch History Injection (BHI) attack [7], Intel introduced a series of speculation controls for Golden Cove and Raptor Cove, including controls for the RRSBA behavior [25]. By enabling the bit `RRSBA_DIS_U`, we confirm that PB-RRSBA is prevented. This bit needs to be enabled for processes that need IBPB, since those would otherwise be vulnerable to classic Spectre. With this chicken bit always-on, SPECrate2017 reported a modest 0.4 % performance overhead.

11. Related Work

We discuss related work that compromise deployed mitigations against transient execution attacks, and side-channel attacks and defenses that target branch predictors.

Breaking hardware mitigations. Barberis et al. [7] showed that the BHB is shared across privilege levels, circumventing Intel eIBRS. Trujillo et al. [9] exploited TTE, which was not considered by AMD AutoIBRS, to build the Inception attack. Inception was possible because AutoIBRS only concerns decoded indirect branches, whereas TTE of RSB works on non-decoded phantom branches [6]. Milburn et al. [2] showed that returns mispredict after IBPB on AMD 17h, but they assumed IBPB does not affect the RSB and did not attempt to exploit it. In this work, we could exploit this issue end-to-end, due to the way IBPB-on-entry is applied in Linux. Post-barrier RSB vulnerabilities on Intel RSBs is a known issue [13]. In this work, we moreover showed that modern Intel processors also have post-barrier behavior for returns predicted by the IP-based predictor.

Breaking software mitigations. Retbleed [8] showed that Retpoline mitigations were ineffective against some forms of BTI attacks through RSBA on Intel CPUs and Branch Type Confusion (BTC) [39] on AMD CPUs. Milburn et al. [2] analyzed AMD processors under various SMT workloads that slows down redirect from a misspeculation, enabling Direct Branch Target Injection, which is another form of BTC. They further showed that the *lfence*-style Retpolines,

originally recommended for AMD systems, were still susceptible to BTI attacks under certain SMT workloads. These attacks show that software-based mitigation tend to only cover the most obvious attack vectors.

Branch predictor side channels and defenses. Evtyushkin et al. [34] demonstrated BTB side channels on Haswell processors, exploiting ASLR. A similar example was shown by Mambretti et al. [32] on newer hardware. These attacks target toy programs, and they only break 8 bits of ASLR entropy. Exploiting PB-RRSBA breaks 20 bits of entropy (i.e., the lower 32 bits) of a suid process of polkit that manages passwords. This is enough address bits to enable other Spectre attacks, such as Pathfinder [17]. Evtyushkin et al. [49] introduced a number of "gadgets" for their *microarchitectural weird machines*, from which we have adopted DC- and IC-gadgets. Zhang et al. [5] reverse engineered addressing of different branch predictors on Intel processors, involved in instruction prefetching. Yavarzadeh et al. [16] reverse engineered the BHB update function and BTB indexing on modern Intel processors enabling a partitioning scheme of the conditional branch predictor against Spectre attacks. Their reverse engineering also enabled precise control of conditional branches across contexts [17].

12. Conclusion

We demonstrated new post-barrier Spectre attacks on Intel and AMD processors. These issues are related to how the IBPB mitigation is implemented in microcode (for Intel) and how it is applied in software (for AMD). Specifically, modern Intel processors do not invalidate IP-based return target predictions. Such predictions are made when the RSB has no valid prediction. For AMD, the IBPB mitigation neglects return target predictions, meaning that care must be taken when using this mitigation, in particular since these processors are vulnerable to phantom speculation. We built two exploits using these insights that bypass the IBPB mitigation. The first exploit derandomizes ASLR and leaks the root password hash of a suid process on Intel CPUs, and the second leaks privileged kernel memory from an unprivileged process. We further proposed, implemented, and evaluated mitigations against these attacks.

Acknowledgements

We thank the anonymous reviewers and our shepherd for their valuable feedback. We further like to thank Chani Jindal of Google, for finding us a ROP gadget chain, and the rest of the Google "btb" group, namely Jordy Zomer, Matteo Rizzo, Alexandra Sandulescu, Eduardo Vela Nava. We thank Borislav Petkov of AMD and Andrew Cooper of XenServer for reviewing and providing feedback our on PB-Inception patch proposal. Part of the research was carried out during an internship at Open Source Security, Inc.

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 *Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[2] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing mitigations for branch target prediction attacks," in *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023.

[3] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Conference on Computer and Communications Security (CCS)*. ACM, 2018.

[4] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

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

[6] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting Decoder-detectable Mispredictions," in *MICRO*, 2023.

[7] 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.

[8] J. Wikner and K. Razavi, "Retbleed: Arbitrary Speculative Code Execution with Return Instructions," in *USENIX Security*, 2022.

[9] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing New Attack Surfaces with Training in Transient Execution," in *USENIX Security*, 2023.

[10] T. Zhang, K. Koltermann, and D. Evtyushkin, "Exploring branch predictors for constructing transient execution trojans," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020.

[11] J. Horn, "Reading privileged memory with a side-channel," 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[12] Intel Corp., "Return stack buffer underflow," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/return-stack-buffer-underflow.html

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

[14] Xenproject.org Security Team, "Xen Security Advisory CVE-2022-23824 / XSA-422," 2022. [Online]. Available: https://xenbits.xen.org/xsa/advisory-422.html

[15] AMD, "Technical update regarding speculative return stack overflow," 2023. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/corporate/cr/speculative-return-stack-overflow-whitepaper.pdf

[16] 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 *Symposium on Security and Privacy (S&P)*. IEEE, 2023.

[17] 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 *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2024.

[18] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, 2006.

[19] AMD, “Software optimization guide for the amd family 19h processors,” p. 30, 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/revision-guides/57095-PUB_1_01.pdf

[20] Intel Corp., “Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1 (248966-050US),” 2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671488

[21] R. E. Kessler, “The alpha 21264 microprocessor.” IEEE, 1999.

[22] K. Driesen and U. Holzle, “The cascaded predictor: Economical and adaptive branch target prediction,” in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998.

[23] A. Seznec, “Tage-sc-l branch predictors again,” in *JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[24] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, “Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks,” in *Workshop on Offensive Technologies (WOOT)*. IEEE, 2022.

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

[26] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *USENIX Security*, 2014.

[27] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of aes,” in *Cryptographers’ track at the RSA conference*, 2006.

[28] M. Hertogh, S. Wiebing, and C. Giuffrida, “Leaky address masking: Exploiting unmasked spectre gadgets with noncanonical address translation,” in *Symposium on Security and Privacy (S&P)*. IEEE, 2024.

[29] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, “Netspectre: Read arbitrary memory over network,” in *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 2019.

[30] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “Smotherspectre: exploiting speculative execution through port contention,” in *SIGSAC Conference on Computer and Communications Security*. ACM, 2019.

[31] A. Kogler, J. Juffinger, L. Giner, L. Gerlach, M. Schwarzl, M. Schwarz, D. Gruss, and S. Mangard, “{Collide+ Power}: Leaking inaccessible data with software-based power side channels,” in *USENIX Security*, 2023.

[32] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, “Two methods for exploiting speculative control flow hijacks,” in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[33] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, “BranchScope: A New Side-Channel Attack on Directional Branch Predictor,” in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2018.

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

[35] 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.

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

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

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

[39] AMD, “Technical guidance for mitigating branch type confusion,” 2022. [Online]. Available: https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf

[40] Xenproject.org Security Team, “Xen Security Advisory CVE-2022-23816,CVE-2022-23825,CVE-2022-29900 / XSA-407,” 2022. [Online]. Available: https://xenbits.xen.org/xsa/advisory-407.html

[41] The kernel development community, “Speculative return stack overflow (srso),” 2023. [Online]. Available: https://docs.kernel.org/admin-guide/hw-vuln/srso.html

[42] ——, “Spectre side channels,” 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html

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

[44] ——, “Affected processors: Guidance for security issues on intel® processors,” 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

[45] H. Y. Luyi Li and D. Tullsen, “Indirector: High-precision branch target injection attacks exploiting the indirect branch predictor,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[46] A. Souchet, “rp++: a fast ROP gadget finder for PE/ELF/Mach-O x86/x64/ARM/ARM64 binaries,” 2013. [Online]. Available: https://github.com/0vercl0k/rp

[47] AMD, “IBPB and Return Stack Buffer Interactions,” 2022. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1040.html

[48] J. Corbet, “When ELF notes reveal too much,” 2024. [Online]. Available: https://lwn.net/Articles/962782/

[49] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, “Computing with time: Microarchitectural weird machines,” in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021.

Appendix A. IBPB-on-entry mitigation

These are the key-changes made to mitigate PB-Inception, applied on top of v6.11.0.

```diff
diff --git a/arch/x86/include/asm/cpufeatures.h b/arch/x86/include/asm/cpufeatures.h
index 3c7434329661..7c56a45d8a32 100644
--- a/arch/x86/include/asm/cpufeatures.h
+++ b/arch/x86/include/asm/cpufeatures.h
@@ -348,6 +348,7 @@
 #define X86_FEATURE_CPPC  (13*32+27) /* Collaborative Processor Performance Control */
 #define X86_FEATURE_AMD_PSFD            (13*32+28) /* "" Predictive Store Forwarding Disable */
 #define X86_FEATURE_BTC_NO  (13*32+29) /* "" Not vulnerable to Branch Type Confusion */
+#define X86_FEATURE_AMD_IBPB_RET (13*32+30) /* IBPB clears return target predictions. */
 #define X86_FEATURE_BRS   (13*32+31) /* Branch Sampling available */

 /* Thermal and Power Management Leaf, CPUID level 0x00000006 (EAX), word 14 */
@@ -523,4 +524,5 @@
 #define X86_BUG_DIV0                   X86_BUG(1*32 + 1) /* "div0" AMD DIV0 speculation bug */
 #define X86_BUG_RFDS                   X86_BUG(1*32 + 2) /* "rfds" CPU is vulnerable to Register File Data Sampling */
 #define X86_BUG_BHI                    X86_BUG(1*32 + 3) /* "bhi" CPU is affected by Branch History Injection */
+#define X86_BUG_IBPB_RET               X86_BUG(1*32 + 4) /* "ibpb_ret" IBPB omits return target predictions */
 #endif /* _ASM_X86_CPUFEATURES_H */
index 07a34d723505..4035c8757f5d 100644
--- a/arch/x86/kernel/cpu/common.c
+++ b/arch/x86/kernel/cpu/common.c
@@ -1443,6 +1443,10 @@ static void __init cpu_set_bug_bits(struct cpuinfo_x86 *c)
             boot_cpu_has(X86_FEATURE_HYPERVISOR)))
                setup_force_cpu_bug(X86_BUG_BHI);

+       if (cpu_has(c, X86_FEATURE_AMD_IBPB) && !cpu_has(c, X86_FEATURE_AMD_IBPB_RET)) {
+               setup_force_cpu_cap(X86_BUG_IBPB_RET);
+       }
+
        if (cpu_matches(cpu_vuln_whitelist, NO_MELTDOWN))
                return;
diff --git a/arch/x86/entry/entry.S b/arch/x86/entry/entry.S
index d9feadffa972..83d999a076b7 100644
--- a/arch/x86/entry/entry.S
+++ b/arch/x86/entry/entry.S
@@ -9,6 +9,8 @@
 #include <asm/unwind_hints.h>
 #include <asm/segment.h>
 #include <asm/cache.h>
+#include <asm/cpufeatures.h>
+#include <asm/nospec-branch.h>

 #include "calling.h"

@@ -19,6 +21,11 @@ SYM_FUNC_START(entry_ibpb)
  movl $PRED_CMD_IBPB, %eax
  xorl %edx, %edx
  wrmsr
+ /*
+  * entry_ibpb should have similar semantics across processors, including
+  * those where IBPB may not clear return target predictions.
+  */
+ FILL_RETURN_BUFFER %rax, RSB_CLEAR_LOOPS, X86_BUG_IBPB_RET
  RET
 SYM_FUNC_END(entry_ibpb)
 /* For KVM */
```

Appendix B. 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.

B.1. Summary

This paper studies Spectre v2 countermeasures present in recent CPUs from Intel and AMD. After reverse engineering when the CPU's branch predictors use Instruction Pointer (IP) or pattern based prediction, and when the CPU falls back to alternate predictors, the paper proceeds to discuss what branch predictors that are sanitized IBPB and, more importantly, those which are not. The authors then proceed to describe exploits, showing an ASLR break on Intel machines and a kernel read primitive on AMD.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Identifies an Impactful Vulnerability

B.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field, by reverse engineering details of opaque prediction structures in modern CPUS. The discovery of how CPUs can fall back to various modes of prediction has the potential to branch off into new research directions on how to do this securely.
2) The paper identifies an impactful vulnerability, showing how RSBA is not properly flushed by IBPB, allowing for cross-process attacks despite barriers assuming suitable gadgets exist in the victim code. The security implication of this issue are then demonstrated via exploits, showing an ASLR break and a root password hash recovery on Intel machines and a kernel read primitive on AMD.