INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution

Daniël Trujillo †
ETH Zurich

Johannes Wikner †
ETH Zurich

Kaveh Razavi
ETH Zurich

† Equal contribution first authors

Abstract

To protect against transient control-flow hijacks, software relies on a secure state of microarchitectural buffers that are involved in branching decisions. To achieve this secure state, hardware and software mitigations *restrict* or *sanitize* these microarchitectural buffers when switching the security context, e.g., when a user process enters the kernel. Unfortunately, we show that these mitigations do not prevent an attacker from manipulating the state of these microarchitectural buffers in many cases of interest. In particular, we present **T**raining in **T**ransient **E**xecution (TTE), a new class of transient execution attacks that enables an attacker to train a target microarchitectural buffer after switching to the victim context. To show the impact of TTE, we build an end-to-end exploit called INCEPTION that creates an infinite transient loop in hardware to train the return stack buffer with an attacker-controlled target in all existing AMD Zen microarchitectures. INCEPTION leaks arbitrary kernel memory at a rate of 39 bytes/s on AMD Zen 4 despite all mitigations against transient control-flow hijacks, including the recent Automatic IBRS.

1 Introduction

Transient execution attacks let attackers execute code in the victim's context to leak sensitive information [9, 30, 33, 36, 48, 49]. To hijack the transient control flow, attackers need to manipulate microarchitectural buffers involved in making branching decisions. A common approach is restricting or sanitizing these microarchitectural buffers when switching security contexts [6, 12, 14, 15, 40]. In this paper, we show that current approaches are insufficient against an attacker that uses privileged software and hardware as confused deputies to train microarchitectural branch predictors with transiently executed instructions.

Hijacking transient execution and mitigations. To hijack the transient control flow of privileged software, like the kernel, attackers manipulate microarchitectural branch prediction buffers, such as the Return Stack Buffer (RSB) [33] or the Branch Target Buffer (BTB) [49]. Consequently, to protect privileged software, mitigations sanitize or restrict these when switching to higher privilege mode. The RSB may be sanitized by means of *stuffing* [15, 33], preventing return instructions of other execution contexts to be hijacked by poisoned RSB entries. A combination of *retpoline* [47] and *jmp2ret* [6] mitigations transform all indirect branches and returns into a single return instruction, whose prediction is sanitized on kernel entry for certain AMD CPUs. Modern microarchitectures support hardware-level features, such as Automatic and Enhanced IBRS [12, 40], that restrict usage of potentially-malicious branch predictions, providing a more efficient mitigation against transient control-flow hijacks.

Training in Transient Execution. Restriction and sanitization of branch predictors assume that an attacker is unable to manipulate these predictors after entering the victim context, such as the kernel. This is unfortunately not true. We present a new class of transient execution attacks that do their **T**raining in **T**ransient **E**xecution (TTE). TTE expands the attack surface of transient control-flow hijacks by using the kernel and in some instances even the CPU as confused deputies for manipulating the BTB and RSB. Our evaluation of the TTE variants shows new capabilities in different scenarios: TTE of the BTB (TTE_{BTB}) trains the BTB in transient execution with a target that is later consumed by a branch to trigger attacker-controlled transient execution. Likewise, by executing a call instruction in transient execution, TTE of the RSB (TTE_{RSB}) trains the RSB with a target that is subsequently consumed by a return instruction. While TTE_{BTB} and TTE_{RSB} can use the kernel as a confused deputy to poison microarchitectural buffers after kernel entry, they require specific gadgets that are not necessarily trivial to find. Is it possible to lift this requirement by turning the CPU into a confused deputy instead?

INCEPTION. Recent work shows that PHANTOMJMPS enable transient control-flow hijacking from an arbitrary instruction on AMD Zen 1(+) and Zen 2 [50], as well as the more recent AMD Zen 3 and Zen 4 [51]. If PHANTOMJMPS allow manipulation of the branch predictor in their short transient window, synergies between PHANTOM and TTE would allow for new variants of TTE. Our investigation shows that TTE_{RSB} is possible inside a PHANTOMJMP, even in the absence of a call, using a new primitive which we refer to as PHANTOMCALLS. By triggering this PHANTOMCALL inside the transient window of a PHANTOMJMP, an attacker can push an arbitrary return address to the RSB by injecting a call prediction for an arbitrary instruction. In essence, the CPU trains the RSB autonomously with a non-existent control flow. PHANTOMCALLS manipulate the RSB regardless of *execution* of the target, bypassing AMD's hardware mitigations such as Zen 2's *chicken bit* and the brand-new Automatic IBRS feature for Zen 4.

Poisoning a single RSB entry alone, however, complicates exploitation. Therefore, our proof-of-concept exploit INCEPTION creates an infinite loop in transient execution using a recursive PHANTOMCALL, poisoning multiple RSB entries. Subsequent return instructions provide INCEPTION with a long-lasting transient execution window from an attacker-provided code location. On Zen 1(+) and Zen 2, this return instruction is in fact the one sanitized on kernel entry with *jmp2ret*, now again under transient control of the attacker due to TTE. Our analysis of possible mitigations suggests that a full flush of the branch predictor is necessary to mitigate INCEPTION. Unfortunately, our analysis shows that Zen 3 and Zen 4 do not provide hardware support for a full flush of the branch predictor, requiring mitigations at the microcode level.

Contributions. Our contributions are as follows:

- Introducing the new TTE class and an evaluation of its variants on Intel and AMD microarchitectures.
- Discovering PHANTOMCALL, allowing manipulation of the RSB despite recent hardware mitigations on all existing AMD Zen microarchitectures.
- Constructing INCEPTION by creating nested PHANTOMCALLS to pollute the RSB recursively. INCEPTION leaks `/etc/shadow` on fully patched AMD Zen 4 systems in 40 minutes, in 6 out of 10 trials.
- Evaluation of the `ibpb` mitigation against INCEPTION on Zen 1(+) and Zen 2. This mitigation introduces between 93.1% and 239.2% overhead on Zen 1(+) and Zen 2, depending on the specific microarchitecture. Our analysis shows that `ibpb` is not a sufficient mitigation against INCEPTION on Zen 3 and Zen 4.

Responsible disclosure. We communicated with Intel and AMD in February 2023. INCEPTION was under embargo until August 8, 2023 to provide adequate time for development and testing of new mitigations that require microcode patching. INCEPTION is tracked under CVE-2023-20569. Further information about INCEPTION can be found at: https://comsec.ethz.ch/inception.

2 Background

We discuss the necessary background concepts for this paper including speculative execution, branch prediction, control-flow hijacks and their mitigation.

2.1 Speculative execution

To prevent under-utilization of execution units due to pipeline stalls, a continual stream of instructions must be provided by the CPU frontend. Slow operations, such as memory requests, that dictate the control flow of a program, are examples of such stalls. Speculative execution is a key technique for avoiding stalls by predicting the control flow of the program.

A control-flow edge, or branch, needs a predicted branch target before its potential dependencies (e.g., memory loads) have been resolved. Branches are either conditional or unconditional, and direct or indirect. All types of branches need predictions to avoid stalls. In particular, unconditional indirect (e.g., *jmp [reg]*) and conditional direct (e.g., *cmp [reg], 0; je L*) branches that depend on slow memory operations greatly benefit from early predictions. Conditional branches can be predicted in two *directions*: taken or non-taken (i.e., *fall through*).

Direction prediction may be rule-based (i.e., static). For example, a conditional backward branch is likely a loop, thus likely taken, whereas a conditional forward branch is likely from an error check, thus likely non-taken (i.e., fall through). Programs are typically run in predictable patterns, so that over time, branch predictors that remember previous branch resolutions can predict current branches with barely any error.

2.2 Modern branch prediction

The branch prediction unit serves predictions for all types of branches. It predicts the direction of conditional branches, the target of conditional and unconditional direct branches, indirect branches, and returns.

A Branch Target Buffer (BTB) stores branch targets associated with different branches. The indexing and structure of BTB entries varies across CPUs. Their purpose however is to provide a branch target given the current instruction pointer and branch history. Both direct and indirect branch targets are provided by the BTB. Conditional direct branches are moreover associated with a Pattern History Table (PHT) that is indexed by the n last branch directions [54]. Modern CPUs are known to use other prediction structures than PHT, such as TAGE [43,45]. Return target predictions are managed by the Return Address Predictor or Return Address Stack (AMD terminology) or Return Stack Buffer (RSB) (Intel terminology). We refer to this buffer as the RSB throughout the paper. The RSB tracks return targets alongside the architectural program stack to provide faster return target predictions without needing to wait for memory-dependent return targets on the program stack. Although RSBs often behave like circular stacks [36], modern processors diverge from such semantics, for example by being able to detect and recover from incorrectly pushed and popped entries [4].

For an accurate prediction, the history of previous branches is sometimes taken into account. This is particularly important for indirect branches and conditional branches, where the target may change during program execution. On Intel CPUs, branch history is stored in a global per-thread Branch History Buffer (BHB) as a footprint of the source and target of the n previously taken branches [9, 30].

Branch predictors receive feedback throughout program execution. However, it is unclear at which stage in the processor pipeline feedback is provided. Can branch predictors receive branch resolution feedback from branches that have not advanced through all pipeline stages?

2.3 Speculative control-flow hijacks

Spectre attacks abuse the above-mentioned buffers to trigger controlled mispredictions, resulting in speculative control-flow hijacks. Spectre-PHT [30] forces the direction of a conditional branch to be mispredicted, Spectre-BTB [30] forces a poisoned BTB entry to be served for an indirect branch, and Spectre-RSB [33, 36] forces a mismatch between the return target on the program stack and RSB. While software and hardware defenses exist to mitigate these, researchers continue to find mitigation flaws that re-enable these attacks [9, 37, 48, 49].

Recent work on AMD CPUs shows that branch target prediction occurs at an early stage in the pipeline, before instructions are decoded [50]. This means that the type of branch (if any) is also subject to prediction, which introduces PHANTOM speculation [51], also known as Branch Type Confusion (BTC) [6]. As such, the prediction of branch type must also be tracked in a data structure, which is assumed to be the BTB.

2.4 Mitigating speculative control-flow hijacks

Spectre-BTB can be mitigated using retpolines [47] or IBRS [12, 40]. Retpolines replace indirect branches with returns, forcing the RSB to be used instead of the BTB for predictions. IBRS is a hardware mitigation that prevents branch targets entries learned in a lower privilege mode (e.g., user mode) to be used in a higher one (e.g., kernel mode). Enhanced IBRS (eIBRS) [12] and Automatic IBRS (AutoIBRS) [40], deployed in newer Intel and AMD processors respectively, are more efficient by not requiring MSR writes on privilege transitions.

Spectre-RSB is mitigated through RSB stuffing. By filling up the RSB with harmless return targets when switching execution context, the return predictions of the victim context can not be influenced by an attacker. Return target prediction can also be forced into BTB prediction by underflowing the RSB [49]. RSB stuffing can be used in combination with call-depth tracking to prevent this on Intel CPUs [56]. Modern Intel CPUs instead support Restricted RSB Alternative to prevent harmful speculation on RSB underflows [25].

Because return instructions can be confused with indirect branches and hence be served BTB predictions on AMD systems vulnerable to PHANTOM speculation, retpolines are insufficient. *jmp2ret* mitigates PHANTOM speculation on returns by replacing all returns (including those inside retpolines) with direct branches to a single, protected return. On privilege transitions, this return is sanitized (i.e., untrained) in the BTB by confusing it with a non-branch instruction [6].

PHANTOM speculation also occurs on non-branch instructions, known as PHANTOMJMPS [50]. To mitigate this issue, AMD revealed an undocumented MSR register bit, known as the *Spectral Chicken* (Linux terminology [57]) or *SuppressBPOnNonBr* (AMD terminology). When set, branch prediction is limited to control-flow edges.

2.5 Discussion

The mitigations discussed above can be categorized as either restricting or sanitizing predictions. Restricting predictions either prevent use of certain predictions (AutoIBRS, and eIBRS) or of an entire predictor (retpolines). Sanitizing predictions, such as *jmp2ret* and RSB stuffing, *sanitize* predictions before execution of vulnerable branches. The main assumption behind both categories, is that predictions must have been poisoned by the attacker *before* transitioning to the victim context (e.g., the higher privileged kernel). The question is whether this assumption is necessarily true, or if branch predictions can be poisoned *after* switching to the higher privilege through a confused deputy?

3 Threat Model

We consider a typical scenario where an unprivileged attacker process aims to leak sensitive information from the kernel. We assume the kernel to be free of software vulnerabilities, and running on a processor that supports speculative and out-of-order execution. Specifically, in this work we target the Linux kernel running on x86-64 Intel and AMD processors. We also assume the default configuration of all existing mitigations against transient execution attacks. These mitigations include retpoline [3, 13], call-depth tracking [56], jmp2ret and *SuppressBPOnNonBr* [6], user pointer sanitization [1], KPTI [23], and disabling of unprivileged eBPF [38]. For our TTE primitives, we consider CPUs from both Intel and AMD, but our end-to-end exploit requires the processor to be affected by PHANTOM speculation (AMD Zen 1 (+), Zen 2, Zen 3 or Zen 4 [51]). For Zen 4, we additionally consider Automatic IBRS, supported in Linux 6.3 and later [40].

4 Overview

To prevent transient execution attacks, mitigations *restrict* or *sanitize* branch predictors between privilege levels. Despite these mitigations, an attacker can still trigger (limited) transient execution windows under which potentially invalid control-flow transfers may be observed by the processor. While these limited windows do not immediately lead to information disclosure, they may be used to perform TTE. The first challenge that we try to address in this paper is to understand the conditions under which TTE is successful on either the BTB or RSB and the requirements that it puts on the attacker.

> **Challenge (C1).** Understanding the necessary conditions for TTE and its requirements for an attack.

Section 5 addresses this challenge by reverse engineering the conditions under which the BTB and RSB can be trained in transient execution. In Section 5, we focus on TTE variants that are relevant for our end-to-end attack, and we leave a more thorough analysis of other TTE variants to Section 8.

Our analysis shows that TTE expands the attack surface of transient execution, but the necessary gadget are sometimes difficult to find [26, 49]. Instead of using a kernel gadget as our confused deputy, we use the CPU to perform TTE with Phantom speculation. According to AMD however, the *SuppressBPOnNonBr* bit (on-by-default in Linux) prevents Phantom speculation arising from non-branch instructions (i.e., PhantomJmps) on Zen 2. Furthermore, Phantom speculation does not trigger transient execution on Zen 3 and Zen 4 [51]. This leads us to our second challenge:

> **Challenge (C2).** Understanding the impact of Phantom speculation on TTE, considering its mitigations and its limited effect on newer microarchitectures.

Section 6 introduces a new Phantom speculation primitive that we refer to as PhantomCall. PhantomCall enables training of the RSB in transient execution (TTE_{RSB}) using non-branch instructions, without requiring any execution. Because of this, PhantomCall is effective on Zen 3 and Zen 4 as well, and neither *SuppressBPOnNonBR* nor Automatic IBRS prevents PhantomCalls.

While this primitive enables us to poison one RSB entry in the kernel context, practical exploitation is difficult due to the undocumented RSB recovery mechanisms. This provides us with the last challenge:

> **Challenge (C3).** Practical exploitation with PhantomCall.

Section 7 describes Inception, our end-to-end exploit using TTE_{RSB}, and PhantomCall. Inception creates an infinite hardware loop without the corresponding software code in transient execution using recursive PhantomCalls, poisoning many RSB entries as a result. This mechanism enables Inception to hijack return instructions. There are a number of additional practical challenges, such as bypassing KASLR and finding disclosure gadgets, that we also discuss in Section 7.

```
1  void TTE_pht_btb (state_t *a, void (*b)()) {
2     if (*a) { /* mispredict as true */
3        b(); /* inject disclosure gadget referenced by b */
4     }
5  }
```

Listing 1: A code snippet vulnerable to $\text{TTE}_{\text{PHT-BTB}}$

5 Training in Transient Execution

The common setup of a transient execution attack is a *speculation gadget* that is trained to transiently execute an incorrect control flow where memory can be leaked through a *disclosure gadget*. A common disclosure gadget loads a secret from an attacker-controlled address, which is then encoded in a subsequent dependent memory access that leaves a trace in the cache, observable via a cache attack such as Flush+Reload [53] or Prime+Probe [39]. TTE has two interesting properties: (i) the injected branch target only ever executes transiently, meaning it can contain, beyond a disclosure gadget, arbitrary or invalid instructions, and (ii) the attacker can escalate a limited speculation primitive under certain conditions.

We experiment with various transient execution windows to see whether they can manipulate the BTB or RSB. We consider four methods to trigger a transient execution path where the BTB or RSB may be trained: (1) through conditional branches, (2) indirect branches, (3) returns, or (4) through Out-of-Order (OoO) execution, which causes a transient execution path, for example after a faulting instruction. We note that other methods are possible, for example transient windows caused by store-to-load forwarding [28] and speculative store bypass [24], which we leave for future work. We use $\text{TTE}_{\text{A-B}}$ to refer to using a transient execution triggered by method A to train the microarchitectural buffer B. In this section, we discuss the general method to accomplish TTE and leave the details of the individual variants to Section 8.

5.1 BTB training in transient execution

Listing 1 demonstrates an example of a code snippet, potentially exploitable with TTE_{BTB}. If the condition of the branch on line 2 holds, an indirect branch to *b* is executed, which trains the BTB. If the attacker can skew the direction of this conditional branch so that *b()* is executed transiently ($\text{TTE}_{\text{PHT-BTB}}$), we hypothesize that the BTB is also trained with the attacker-controlled value *b* as branch target.

```
void TTE_pht_rsb (state_t *a) {
   if (*a) {                /* mispredict as true */
      f();                  /* push return target to RSB */
      DISCLOSURE_GADGET; /* top of RSB will point here */
   }
   return; /* predict DISCLOSURE_GADGET as target */
}
```

Listing 2: A code snippet vulnerable to $\text{TTE}_{\text{PHT-RSB}}$

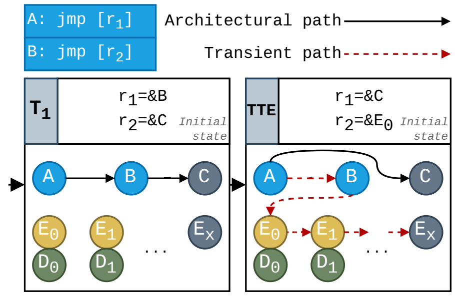

Figure 1: Injecting RSB entries in the transient execution window of an indirect branch ($\text{TTE}_{\text{BTB-RSB}}$). In T_1, A is trained to execute B. Next, in *TTE*, A transiently executes B, which in turn executes a series of *call gadgets* E_i, each followed by a *disclosure gadget* D_i.

```
id = 0
.rept N
   call 1f
   load (RB + id*4096), %rdi
   1: lfence
   pop %r8
   id = id + 1
.endr
```

Listing 3: Pseudo-assembly priming the RSB with N different return targets before the TTE_{RSB} experiment. Execution is measured with a cache side-channel. N is the RSB capacity.

```
.rept N
  push 1f
  ret
  1:
.endr
```

Listing 4: Inferring the RSB state by issuing N returns.

To turn TTE of the BTB into arbitrary transient code execution, two conditions must be fulfilled: (i) we can skew the conditional branch direction, and (ii) we control b to inject an arbitrary branch target. To furthermore leak memory, an attacker needs additional control over at least a memory pointer to some secret of interest. Leveraging TTE with this example, the attacker gains an extra primitive: they can train the indirect branch using transient execution in a preparatory step. Afterwards, they can run the victim again and provide an arbitrary value in b while still reaching the previously injected branch target.

However, indirect branches are commonly replaced by retpolines on many microarchitectures, since they are known to be vulnerable to Spectre-BTB. Additionally, meeting condition (ii) means the attacker already has arbitrary transient code execution but uses it for training instead of leaking data. More complex scenarios exist than the toy example in Listing 1, for example caused by speculative type confusion [29]. Regardless, TTE_{BTB} exposes new possibilities for the attacker, and we discuss its variants in Section 8. Next, we discuss TTE for the RSB, that loosens requirements from the viewpoint of an attacker.

5.2 RSB training in transient execution

Listing 2 demonstrates a piece of code that may update the RSB in with a transiently executed call instruction triggered by a mispredicted conditional branch ($\text{TTE}_{\text{PHT-RSB}}$). This piece of code yields some interesting results: the transiently executed call updates the RSB only on all AMD microarchitectures, although unreliably. To investigate further, we construct a more thorough experiment using $\text{TTE}_{\text{BTB-RSB}}$.

Experiment setup. Figure 1 illustrates how we verify RSB training through a mispredicted indirect branch with a training procedure (T_1) and a training in transient execution procedure (TTE). The goal of the experiment is to determine whether a *call* instruction executed in transient execution manipulates the state of the RSB. Green nodes (D_i, $0 \leq i < x$) are *disclosure gadgets* that we try to inject into the RSB by transiently executing call sites E_i, $0 \leq i \leq x$ (yellow nodes) that immediately precede the disclosure gadgets. For each D_i, a different memory load inside a reload buffer is used to indicate which D_i transiently executed. E_i calls the next call site E_{i+1}, in sequence, such that D_i becomes the return target of E_{i+1}, until reaching E_x. Gray nodes E_x and C, are *barrier gadgets* to stop speculation using a memory barrier instruction (e.g., *mfence*). We run our experiment for $0 \leq \text{x} \leq 50$ to be able to compare the results of executing different numbers of transient calls.

To recover from `call` instructions which are executed transiently, return target predictors implement mechanisms that restore the RSB to a consistent state [4]. This means that RSB entries manipulated in transient execution may become invalid and unusable as a consequence. The D_i gadgets are thus only observable for transiently pushed entries that were not invalidated. To also observe invalidated entries, we establish a known state of the RSB by priming it fully using N calls preceding N additional disclosure gadgets, as shown in Listing 3, where N is the size of the target RSB (e.g. 31 on Zen 1, Zen+ and Zen 2). We flush the $x+N$ reload buffer (RB) entries from the cache hierarchy before running the experiment.

We first execute T_1, which trains the BTB to transiently execute B in the TTE step. Next, we prime the RSB according to Listing 3. We then execute TTE, which triggers a series of calls to E_i, potentially manipulating the RSB. After performing the experiment, we examine the RSB state. To do this, we execute N returns, as shown in Listing 4. If the RSB was not manipulated by TTE, we expect to have transiently

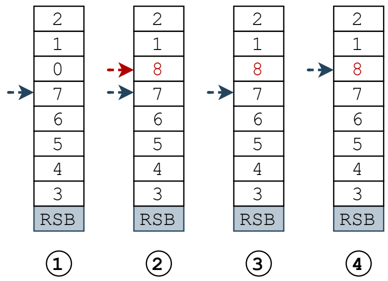

Figure 2: An implementation of a circular RSB with a committed top-of-the-stack pointer (shown in blue) and a speculative counterpart (shown in red). RSB entry numbers indicate their insertion order (0 first, 8 last). Entries depicted in red were inserted transiently. ① shows RSB state before transient execution. ② pushes an entry to the RSB transiently. In ③, the transient window is over and the speculative pointer is restored. ④ shows the RSB state after 7 returns.

executed the primed return sites in Listing 3. If an RSB entry was manipulated but invalidated, we expect it to no longer be used. If an RSB entry was manipulated but not invalidated, we expect to observe a memory access triggered by D_i.

Results. The results show that manipulating the RSB with $TTE_{BTB-RSB}$ is feasible on all considered AMD microarchitectures, but not on Intel microarchitectures (Table 1 in Section 8 includes all our TTE results). This is in line with the Software Optimization Guide, which states that transient pushes and pops to the RSB may occur [4, 5]. We observe that transiently executed calls evict the oldest entries, at the *bottom of the RSB*. That means that, in the case of a single transiently executed call, the last return will not use its corresponding primed RSB entry. Likewise, executing two transient calls evicts two entries at the bottom of the RSB, causing the last two returns executed to their corresponding RSB entries. RSB manipulation of the bottom entries could be explained by having two RSB pointers for a circular buffer: a committed one and a speculative one, as shown in Figure 2. Upon misprediction, the speculative pointer restores to the committed one, which effectively puts transiently injected entries at the bottom of the buffer.

However, the returns associated with the corrupted entries do not consume the injected D_i targets, nor the previously primed entries. Instead, RSB entries untouched by the transiently executed calls are recycled. For example, in step ④ of Figure 2, entry 1 may be predicted instead of the overwritten entry 8. This suggests that AMD's return predictors implement recovery mechanisms for handling transiently pushed entries. We find that we can bypass these mechanisms by executing multiple calls in a transient window. For example, on Zen 1(+) and Zen 2, this happens as soon as we overwrite *all* 31 RSB entries. We will discuss this in more detail in Section 7.1.

Observation (O1). We can corrupt return predictions on AMD microarchitectures with $TTE_{BTB-RSB}$.

On Intel microarchitectures, we were unable to poison any RSB entry with TTE. Intel patents describe speculative RSBs [19, 27], which could result in the behavior we observe.

We will further show in Section 8 that other transient execution windows have a similar effect on AMD's RSB. However, finding exploitable gadgets similar to Listing 2 in the victim code might be difficult. The question is whether we can relax this constraint by abusing other properties of AMD microarchitectures.

6 PHANTOM and TTE

AMD Zen microarchitectures are known to be vulnerable to PHANTOM speculation, leading to additional potential variants of TTE. PHANTOM is a class of transient execution issues arising when the predicted branch type, stored in the BTB, conflicts with the actual instruction [6, 51]. For example, the BTB may contain a prediction for an indirect jump, while the actual code location contains a return, resulting in the return being predicted as an indirect branch. PHANTOM can also occur in absence of any branch, triggering speculation from non-branch instructions, referred to as PHANTOMJMPS [50].

Chicken out from PHANTOMJMPS. In response to the discovery of PHANTOM, AMD revealed an undocumented configuration of Zen 2 CPUs that can be enabled by setting MSR bit `0xC00110E3[1]`, known as *SuppressBPOnNonBr*. This is a *chicken bit* that configures the CPU's branch target predictor to suppress predictions for non-branch instructions. The configuration promises that all speculative execution on non-branch instruction is prohibited, which reduces the attack surface of PHANTOM to arbitrary branches only. This mitigation is currently enabled by default on Linux.

PHANTOM on Zen 3 and Zen 4. While AMD originally claimed that Zen 3 and Zen 4 are immune to PHANTOM, later work has shown that these microarchitectures are also affected by PHANTOM, although only partially [51]. In particular, transient execution due to a PHANTOMJMP is not possible on these newer microarchitectures, but the predicted target is still fetched and decoded.

6.1 Synergies between TTE and PHANTOM

Recalling the hard-to-find gadget in Listing 2, we want to know whether it can be simplified using PHANTOM. Our first observation is that a conditional branch is unnecessary if we can trigger a call transiently using PHANTOM. Since PHANTOM on Zen 1(+) and Zen 2 results in transient execution, we can use any branch to trigger a transient call to manipulate the RSB. On Zen 1(+), we can even hijack non-branch instructions. Hence, PHANTOM would significantly relax the

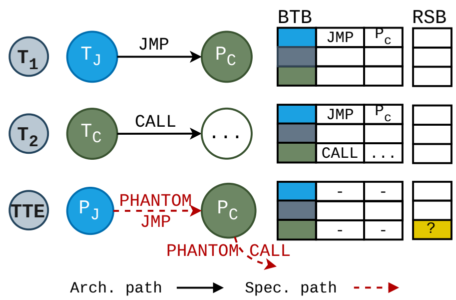

Figure 3: Experiment setup to test feasibility of executing a PHANTOMCALL in a PHANTOMJMP. Green and blue colors indicate two different BTB mappings.

constraints of the original gadget, since it can be split into two. The first half of the gadget only needs to contain an arbitrary instruction that can be poisoned with PHANTOM, followed by a return. The second half, being a call followed by a disclosure gadget, can be anywhere in the executable address space.

Training the RSB in a PHANTOM window. With PHANTOM, the architectural branch type solely dictates the length of the transient execution window [6], giving rise to two cases which lead to transient execution: a short transient window, concerning architectural direct branches or non-branch instructions, and a long transient window, concerning architectural indirect branches and returns. We are mostly interested in the cases yielding a short transient window, since they can be triggered on arbitrary branches on both Zen 1(+) and Zen 2.

We design an experiment to determine the feasibility of manipulating the RSB within a PHANTOM-induced transient window. For this, we execute an indirect branch to trigger a PHANTOMJMP at the victim instruction using out-of-place training. We set the target of the indirect branch to an address that contains a call instruction. To determine whether the PHANTOMJMP interacts with the RSB, we prime the RSB as as shown in Listing 3 and 4.

Results. Our results show that we can manipulate the RSB within a PHANTOM-induced window on both Zen 1(+) and Zen 2. On Zen 1(+) the PHANTOMJMP to the call can be triggered even on non-branch instructions, which is expected, since mitigations against this are only available on Zen 2.

> **Observation (O2).** We can manipulate the RSB using PHANTOM speculation on Zen 1(+) and Zen 2.

However, PHANTOM speculation on non-branch instructions is prevented with the SuppressBPOnNonBr mitigation on Zen 2. Likewise, on Zen 3 and Zen 4, PHANTOM speculation does not allow transient execution. Section 6.2 discusses how we bypass SuppressBPOnNonBr on Zen 2 with a new primitive we refer to as PHANTOMCALL, and Section 6.3 discusses how minor adaptations to PHANTOMCALL makes it effective on Zen 3 and Zen 4 as well, despite AutoIBRS.

6.2 Bypassing SuppressBPOnNonBr with PHANTOMCALL

We hypothesize that TTE of the RSB using PHANTOM works because of a *call prediction* on the PHANTOMJMP target, and not because of the call instruction itself. We design an experiment to test our hypothesis, as shown in Figure 3, with the state of the BTB and RSB shown after each step. In training step T_1, we first execute a branch from training branch source T_J to PHANTOMCALL source P_C. This creates a BTB entry for a branch, with its target set to P_C, which only contains NOPs. In training step T_2, we execute a 3-byte wide call instruction at call source T_C, inserting a BTB entry for a call (target not relevant). T_C and P_C map to the same BTB entry. After performing steps T_1 and T_2, we fully prime the RSB with distinct return sites, each issuing an identifiable memory access, as shown in Listing 3. In step TTE, we execute the NOP instructions at PHANTOMJMP source P_J, which collides with the BTB entry of T_J. Thanks to step T_1, we expect P_C as the predicted target of P_J. Thanks to step T_2, because there exists a call-prediction for P_C, we expect the CPU to transiently push a return target (P_C+3) onto the RSB. Lastly, we flush our reload buffer and execute return instructions according to Listing 4. We reload our memory pointers to determine which of the RSB entries are invalid or still intact.

The results confirm our hypothesis: the last return does not transiently execute the primed return site, meaning we have overwritten an RSB entry using a PHANTOMCALL inside a PHANTOMJMP-induced speculation window (i.e., nested PHANTOM speculation). If there exists a call-target prediction at our PHANTOMJMP target, we presume the CPU does not need to decode before pushing its predicted return target to the RSB. We therefore conclude that *the call prediction alone prematurely pushes to the RSB*, before instructions are decoded. Supporting this conclusion, we find that the PHANTOMJMP to the PHANTOMCALL manipulates the RSB even when both branches are injected on non-branch instructions, despite the Zen 2 SuppressBPOnNonBr mitigation. Given this, we hypothesize that SuppressBPOnNonBr, while suppressing transient execution of PHANTOMJMP targets, does not suppress BTB consultation, allowing RSB manipulation without execution.

We refer to this new primitive as PHANTOMCALL, allowing us to manipulate the RSB from any instruction, without any architectural call instruction on the transient path.

> **Observation (O3).** We can corrupt an RSB entry using a PHANTOMCALL on Zen 1(+) and Zen 2 microarchitectures, bypassing SuppressBPOnNonBr.

6.3 PHANTOMCALL on Zen 3 and Zen 4

Given that our results show that we can perform TTE_{RSB} without any transient execution using a PHANTOMCALL,

Figure 4: Triggering TTE$_{RSB}$ inside a PHANTOM speculation window on Zen 3/4. The colored boxes identify different cache lines. Architectural and transient branches are indicated by solid black and dashed red arrows, respectively.

we investigate whether we can use this primitive on Zen 3 and Zen 4 as well. After additional reverse engineering, we find that TTE$_{RSB}$ using PHANTOM on Zen 3 and Zen 4 is effective, but only under certain circumstances. In particular, TTE$_{RSB}$ using PHANTOM requires both the PHANTOMJMP and the PHANTOMCALL to be at specific memory addresses relative to those used for BTB consultation.

To successfully trigger the call in a PHANTOMJMP target, we consider four different cache lines as shown in Figure 4. We inject the PHANTOMJMP on a cache line A+1, which linearly follows the cache line of a preceding branch target. Similarly, we place the PHANTOMCALL on a cache line B+1, which is the cache line following that of the PHANTOMJMP target. We hypothesize that this is necessary to delay the decoder. The time it takes for the frontend to fetch the next cache line and feed it to the decoder may introduce enough delay to allow manipulation of the RSB before the decoder can detect that predictions are incorrect.

We find that AutoIBRS does not prevent RSB manipulation due to a PHANTOMCALL inserted in a lower privilege level. This is in line with our observations on Zen 2, where we bypassed the SuppressBPOnNonBr mitigation with PHANTOMCALL. We thus hypothesize that AutoIBRS only prevents transient execution at the target of a PHANTOMJMP, and not consultation and manipulation of the BTB and RSB respectively. We can also deduce this from statements previously released by AMD [6], which mention that IBRS is effective for branches decoded as indirect, indicating the restriction is enforced after instructions have been decoded.

> **Observation (O4).** PHANTOMCALL works on Zen 3 and Zen 4 as well under certain circumstances, bypassing AutoIBRS.

PHANTOMCALLS significantly simplify the requirements for exploitation with TTE$_{RSB}$. We can insert the address of an arbitrary disclosure gadget into the RSB by injecting a PHANTOMCALL right before it. Furthermore, by performing this PHANTOMCALL inside a PHANTOMJMP, we can trigger this from anywhere. We leverage these capabilities in our end-to-end exploit, INCEPTION, which we discuss next.

7 INCEPTION

To turn PHANTOMCALL into an end-to-end exploit, we need to overcome two challenges. First, to bypass mechanisms that restore the RSB, we need to overwrite multiple RSB entries, as pointed out in Section 5.2. We thus need to construct a chain of PHANTOMCALLS, where the last PHANTOMCALL has to precede a disclosure gadget. In particular, on Zen 1(+) and Zen 2 we need to overwrite all 31 RSB entries to bypass the recovery mechanism. Second, the short transient execution window, caused by a PHANTOMJMP, needs to somehow fit the chain of PHANTOMCALLS to overwrite all these RSB entries. Addressing this challenge requires new insights that we discuss in Section 7.1 and Section 7.2. We then proceed to the design of our end-to-end exploit INCEPTION in Section 7.3 through Section 7.7. Lastly, we evaluate INCEPTION on Zen 2 and Zen 4 in Section 7.8 and Section 7.9, respectively.

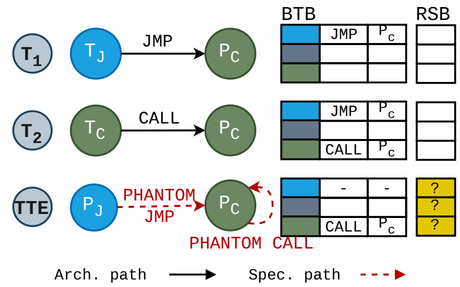

Figure 5: The experiment setup to test the number of entries we can pollute with a recursive PHANTOMCALL in a PHANTOMJMP. Green and blue colors indicate two different BTB mappings.

7.1 Recursive PHANTOMCALL

To turn PHANTOMCALLS into a practical exploit, we need a large number of PHANTOMCALLS in a single transient window. We therefore construct a chain of PHANTOMCALLS to determine how many we can execute using a single PHANTOMJMP. We realize that we can establish a single PHANTOMCALL that branches into itself, i.e. a *recursive loop* of PHANTOMCALLS. By avoiding changing the (transient) instruction pointer, we assume that the CPU can manipulate the most RSB entries in a single transient window.

Repeating the experiment described in Section 6.2, we monitor the number of RSB entries that get corrupted by a recursive PHANTOMCALL. However, this time, the call at T_C branches into P_C, at which the PHANTOMCALL will be triggered, thus establishing a recursive prediction. An overview of the experiment is shown in Figure 5. Since P_C executes after T_C in T_2, and P_C and T_C map to the same BTB entry, executing P_C should invalidate the prediction from T_C to P_C. To avoid this, we make sure that the indirect call in step T_2 page faults, by temporarily unmapping P_C. Regardless of the page fault, we expect the BTB to be primed with a prediction, as shown in previous work [49]. Interestingly, unmapping is unnecessary on Zen 1(+) and Zen 2. We believe that this could be due to a race condition that happens to be in our favor. The prediction associated with T_C may not have

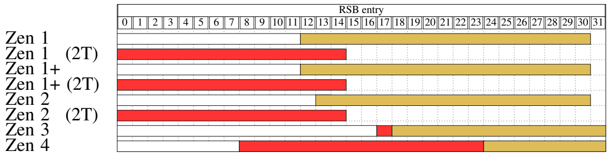

Figure 6: Entries affected by the recursive PHANTOMCALL. Yellow shows invalidated entries that remain unconsumed, while red indicates poisoned entries consumed for prediction, enabling arbitrary transient code execution. The size of the RSB is 31 on Zen 1(+) and Zen 2 (or 15 in 2T mode), while it contains 32 entries on Zen 3 and Zen 4.

updated the BTB when we are executing P_C.

Results. Figure 6 shows that we can corrupt a large number of RSB entries using our recursive PHANTOMCALL in a PHANTOMJMP on all Zen microarchitectures. An interesting observation we make is that the PHANTOMCALL at P_C is not invalidated after the TTE step for most of the iterations, unlike the prediction for the PHANTOMJMP. This is beneficial for our attack, since it allows us to trigger the recursive PHANTOMCALL multiple times after priming the BTB.

As discussed in Section 5.2, corrupted entries are not always used for return prediction, due to the RSB recovery mechanisms. On Zen 3 and Zen 4 however, we find that our recursive PHANTOMCALL overwrites enough entries to *bypass* the recovery mechanisms, as shown in Figure 6. Specifically, on Zen 3 microarchitectures we hijack a single return instruction by first exhausting 17 uncorrupted RSB entries. On Zen 4, we need to exhaust 8 uncorrupted RSB entries, after which we control the next 16 return target predictions. We find that the number of RSB entries polluted heavily relies on the exact location at which we trigger PHANTOM speculation, the state of the cache, the state of the BTB, and the preceding control flow.

On Zen 1(+) and Zen 2 microarchitectures, however, we do not overwrite enough RSB entries to bypass the recovery mechanisms. Our results in Section 5.2 showed that transiently overwriting *all* 31 RSB entries leads to all corrupted entries being used for prediction on these microarchitectures. We therefore expect that overwriting all RSB entries using a recursive PHANTOMCALL would allow us to bypass the recovery mechanisms on Zen 1(+) and Zen 2 as well.

7.2 Dual-threaded mode

Rather than trying to achieve 31 transient recursions in the transient window of a PHANTOMJMP, we consider whether the capacity of the RSB can be reduced. When two sibling threads are operating in parallel, Zen 1(+) and Zen 2 cores switch to *dual-threaded mode* (2T-mode) [5], reducing the RSB to only 15 entries per thread, instead of 31. As shown in Figure 6, we can poison 18 entries in nested PHANTOM speculation on Zen 1(+) and Zen 2, and we thus potentially control the entire RSB associated to a sibling thread under dual-threaded mode.

We verify that the RSB capacity decreases from 31 to 15 entries for our thread while executing a workload in parallel from the sibling thread. Repeating the experiment shown in Figure 5 reveals that we can indeed overwrite all 15 RSB entries on Zen 1(+) and Zen 2 microarchitectures. Having overwritten all entries, our transiently injected return target is used by all following returns, as shown in Figure 6. This means that we do not rely on deep call stacks on Zen 1(+) and Zen 2: any return can be hijacked in dual-threaded mode by triggering the recursive PHANTOMCALL right before it is executed.

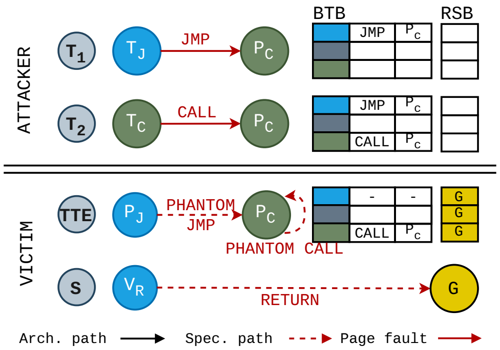

Figure 7: INCEPTION visualized. The BTB and RSB state is shown following steps T_1, T_2, and TTE. Green and blue colors indicate two different BTB mappings.

7.3 Exploit design

We are now able to hijack return instructions by injecting arbitrary return targets using our recursive PHANTOMCALL on all AMD Zen microarchitectures. Using this, we will construct our exploit INCEPTION on Zen 1 (+), 2, and 4. INCEPTION is not fully successful on Zen 3, as discussed later this section.

Figure 7 shows a visualization of INCEPTION together with the resulting state of the BTB and RSB after each training step. In the first training step T_1, the attacker executes a training branch at T_J, which collides with the BTB entry of PHANTOMJMP source P_J. Residing in the kernel address space, P_J is the address that initiates the recursive PHANTOMCALL. The victim return V_R is allocated after P_J in the control flow. The target of the PHANTOMJMP is set to P_C, at which the recursive PHANTOMCALL will be triggered. In training step T_2, the attacker executes a training call at T_C that collides with P_C in the BTB, which will establish the prediction for the PHANTOMCALL. The target of this training call at is set to P_C, establishing a recursive PHANTOMCALL prediction. Upon execution of P_C, the CPU will thus recursively inject RSB predictions to disclosure gadget G, whose location immediately follows the PHANTOMCALL at P_C. As P_C resides in kernel space, the training branches T_J and T_C will trigger page faults, which we recover from.

On Zen 3 and 4, we take the cache line placement of the branches at T_J and T_C into account. Concretely, this means that the PHANTOMCALL in P_C may be preceded by different instructions to ensure that the start of P_C and the PHANTOMCALL fall in different cache lines. Likewise, the PHANTOMJMP in P_J may be preceded by different instructions, depending on the address using which the BTB is indexed before executing P_J.

After steps T_1 and T_2, we invoke the kernel using a system call to trigger the TTE step. Whenever we reach P_J, the BTB provides the prediction to P_C, and the speculative instruction pointer is set to P_C. Since there exists a prediction for a call at P_C, G is pushed to the RSB. Since the call prediction is recursive, we will continue the loop of 1) updating the instruction pointer, 2) consulting the BTB and 3) pushing to the RSB. This recursion continues until the actual instruction at the location of the PHANTOMJMP in P_J is eventually decoded and the CPU corrects the misprediction by resetting the instruction pointer back to P_J. Finally, in step S the victim return at V_R will take a prediction from the RSB. Since we have overwritten RSB entries with return target G during the TTE step, we start executing the disclosure gadget at G, accomplishing a long speculation window in which we control the executed instructions.

7.4 Dueling recursive PHANTOMCALLS

The desired disclosure gadget may not exist in the kernel code, specially if the hijacked return is in a deep call stack (i.e., on Zen 3 and Zen 4). In this case, INCEPTION can execute two separate disclosure gadgets within the same transient window, that together perform the desired operation, similar to [55]. INCEPTION achieves this by introducing two recursive PHANTOMCALLS, or dueling recursive PHANTOMCALLS, establishing a transient Return-Oriented Programming (ROP) chain. The first recursive PHANTOMCALL trains the RSB with the first disclosure gadget, G_1, while the second recursive PHANTOMCALL inserts the address of the second disclosure gadget G_2. As a result, some entries in the RSB contain the address of G_1, while others contain the address of G_2. If G_1 ends with a return instruction, G_2 potentially executes in the same speculation window. However, for this to work, RSB recovery mechanisms must be bypassed without overwriting all entries, which is only possible on Zen 3 and 4.

The end goal of dueling recursive PHANTOMCALLS is to have some (ideally one) of the newer RSB entries contain the address of G_1, and to have the other, older RSB entries contain the address of G_2. Figure 8 shows a possible progression of the RSB state over time. ① shows the unmodified RSB, before the first recursive PHANTOMCALL. ② shows the state after triggering the first recursive PHANTOMCALL, which precedes G_1. ③ shows the state after two returns. ④ shows the state after issuing the second recursive PHANTOMCALL, which precedes G_2. Lastly, step ⑤ shows the RSB state after additionally two returns. The next return will transiently execute G_1, and until its return target has been resolved, subsequent returns will keep taking predictions from the RSB, eventually leading to transient execution of G_2. If G_1 is idempotent with respect to the CPU state relied on by G_2 (e.g. register or memory values), G_1 can be executed more than once transiently. Otherwise, we target the next return instruction, which executes G_1 once before reaching G_2, as shown in step ⑥.

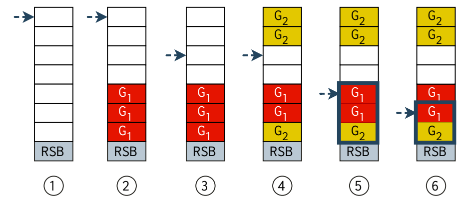

Figure 8: Triggering dueling recursive PHANTOMCALLS to chain two disclosure gadgets G_1 and G_2 together. The arrow is the committed top-of-the-stack pointer.

7.5 Victim return instruction

Having designed INCEPTION, we proceed by searching for an exploitable victim return in the Linux kernel. The first requirement is that upon execution of the victim return, we control the values in two registers or memory locations, V_1 and V_2. We use this to leak arbitrary information through the kernel's physmap area, similarly to previous work [20, 49].

As stated before, we can overwrite all RSB entries on Zen 1 (+) and 2, and all of them will be used for return target prediction. On Zen 3 and 4 we can only reach poisoned RSB entries served for prediction after exhausting a number of uncorrupted RSB entries. Therefore, on Zen 3 and 4, a second requirement is to exhaust these returns after the recursive PHANTOMCALL, before the victim return.

Previous work built an open-source framework to trace register contents in the Linux kernel at the time of executing a return instruction [49]. We use this framework to find vulnerable returns that meet our requirements.

7.6 Derandomizing KASLR

As in previous work [49], we derandomize KASLR in three steps. In all, we prime the BTB with the PHANTOMJMP and the recursive PHANTOMCALL before issuing the system call.

① Finding the kernel text. We use a disclosure gadget that dereferences the attacker-controlled pointer in V_1. When executing the system call, this load triggers only if we guess the kernel text location right, which we infer using Prime+Probe.

② Finding physical address mapping. To find the physical address of our reload buffer, we trigger a transient load of a physmap offset. We achieve this using a gadget that adds V_1 to the physmap base address and dereferences the result.

```
1 leave
2 xor    edx,edx
3 mov    esi,edx
4 mov    edi,edx
5 jmp    return_thunk   ; jumps to return (jmp2ret mitigation)
```

Listing 5: The location where we trigger PHANTOM speculation in *__fdget_pos* at kernel code offset `0x41db94` on Zen 1 (+) and 2. It ends with a direct branch to the return thunk.

```
1 add    r12,QWORD PTR [physmap]
2 mov    rax,QWORD PTR [r12]
```

```
1 movzx  eax,BYTE PTR [r14+0x2]
2 lea    rdx,[r12+rax*2]
3 movzx  r13d,WORD PTR [rdx]
```

Listing 6: Disclosure gadgets used on Zen 1(+) and Zen 2 for derandomizing KASLR (top, at offset `0xf22a44` of kernel text) and arbitrary information leakage (bottom, at offset `0x70c4a6` of kernel text).

Using Flush+Reload, we detect when we guess the physical address correctly.

③ Finding physmap. We derandomize physmap using the same gadget as in ①. We trigger a transient load of V_1, which instead provides the reload buffer physical address added to our physmap base address guess. Using Flush+Reload, we detect when we guess the physmap address correctly.

7.7 Leaking kernel memory

To leak memory, we need to trigger transient execution of a disclosure gadget that performs a secret-dependent access in our reload buffer. We first prime the BTB with the PHANTOMJMP and the recursive PHANTOMCALL. We trigger execution of a disclosure gadget that dereferences V_1 and uses its result to index into the address in V_2, which points to the reload buffer. If V_1 or V_2 are provided by memory locations, they first need to be loaded from memory into registers. Using Flush+Reload, we can deduce the secret residing at address V_1.

7.8 INCEPTION on Zen 1(+) and Zen 2

Vulnerable return. We find that after issuing the system call *readv()*, register **R12** will hold the value we pass in **RSI** (i.e., second argument) and register **R14** will hold the value we pass in **RDX** (i.e., third argument) at the moment we execute the return instruction of function *__fdget_pos()*. Listing 5 shows the last instructions of this function. We trigger the PHANTOMJMP on line 2, poisoning the RSB right before jumping to the return.

Disclosure gadgets. We could find the desired gadgets with simple string matching against the assembly code of the kernel text. Listing 6 shows the disclosure gadgets found. Line 2 of Listing 6-top is used for steps ① and ③ of breaking KASLR. Lines 1 and 2 are together used for step ②. Lastly, Listing 6-bottom presents the disclosure gadget used to leak arbitrary data.

```
1 call   0x8cfbe640
2 test   eax,eax     ; index BTB upon return from call
3 jg     0x8cf9040d  ; trigger PhantomJMP (next cacheline)
```

```
1  call   0x8cf1bbf0
2  test   eax,eax     ; index BTB upon return from call
3  js     0x8cfbbc83
4  pop    rbx
5  mov    eax,r13d
6  pop    r12
7  pop    r13
8  pop    r14
9  pop    rbp
10 xor    edx,edx     ; trigger PhantomJMP (next cacheline)
```

Listing 7: The locations where we trigger PHANTOM speculation in *ip6_protocol_deliver_rcu* (top, at offset `0xd905b9` from the start of the kernel) and *udpv6_queue_rcv_one_skb* (bottom, at offset `0xdbbba7` from the start of the kernel text) on Zen 3 (bottom only) and Zen 4.

Results. We evaluate INCEPTION on an AMD Zen 2 EPYC 7252 with microcode version `0x8301038` and 64GB of RAM, running Linux 5.19.0-28-generic with all mitigations deployed. We run our attack 50 times, each time leaking 4KB of randomized data. We reboot the machine every run to re-randomize KASLR. Of the 50 runs, we successfully break KASLR in 48 cases, in a median time of 5.5 seconds. In those cases, INCEPTION leaks data at a rate of 126 bytes/s, with an accuracy of 89.9%.

We furthermore show that INCEPTION is capable of locating secrets in the physical memory. Specifically, we let INCEPTION search for `/etc/shadow` to leak the root password hash. We run INCEPTION in parallel on all 8 available cores, where each instance starts searching at a different physical address. We try to locate `/etc/shadow` 10 times and reboot the machine after every attempt. Our results show that we are able to successfully leak the root password hash in all 10 runs, in a median of 11 minutes and 38 seconds.

7.9 INCEPTION on Zen 3 and Zen 4

Vulnerable return. We target the *sendto()* system call, controlling memory locations using our message buffer, whose address we pass in **RSI** (i.e. second argument). We find that upon execution of the return in *do_softirq.part.0()*, our message buffer is reachable using the address in **RBX**. Likewise, our message buffer is pointed to by the **R13** register upon execution of the return instructions of *ip6_local_out()* and *ip6_send_skb()*.

Listing 7-top shows the location where we trigger the PHANTOMJMPS to our recursive PHANTOMCALL. Specifically, we trigger the PHANTOMJMP on the **jg** instruction, shown on Line 3. We find that we reliably

```
mov    rax,QWORD PTR [rbx+0x58]
mov    r13,QWORD PTR [rax+0x8]     ; Load kernel text
mov    rax,QWORD PTR [rbx+0x18]
mov    r12d,DWORD PTR [rax+0x20]    ; Load kernel text + 4096
mov    rax,QWORD PTR [rbx+0x30]
shr    r12d,0x8
and    r12d,0xff00
mov    rsi,QWORD PTR [rax+0x10]    ; Load kernel text + 8192
```

```
mov    rax,QWORD PTR [rbx+0x48]    ; Load physmap guess addr
mov    rdi,QWORD PTR [rax+0xc0]    ; Load physmap guess
```

```
mov    rsi,QWORD PTR [rbx+0x48]    ; Load secret addr
lea    eax,[rdx+0x2]             ; rdx == 0
shl    r14d,cl
mov    rcx,QWORD PTR [rbx+0x60]    ; Load reload buffer addr
and    edx,DWORD PTR [rbx+0x40]
movzx  eax,BYTE PTR [rsi+rax*1]    ; Load the secret
xor    eax,r14d                 ; XORs secret
and    eax,DWORD PTR [rbx+0x74]    ; ANDs secret
mov    DWORD PTR [rbx+0x68],eax
movzx  r14d,WORD PTR [rcx+rax*2]    ; Leaks the data
```

Listing 8: Disclosure gadgets used on Zen 4 for derandomizing KASLR (top and mid, at offset `0xb0a720` and `0x97ef01` of kernel text respectively) and arbitrary information leakage (bottom, at offset `0x701d74` of kernel text).

hijack the return in *do_softirq.part.0()* on Zen 4, during which the address of our message buffer is held by **RBX**. On Zen 3, however, we only control this return in a small percentage of the iterations. Therefore, on Zen 3 we trigger the PHANTOMJMP to our recursive PHANTOMCALL in the *udpv6_queue_rcv_one_skb()* function, specifically on the **xor** on Line 10 in Listing 7-bottom. We find that we reliably hijack the return of either *ip6_local_out()* or *ip6_send_skb()*, during which our message buffer address is stored in **R13**.

Disclosure gadgets. On Zen 3, we are unable to find a disclosure gadget that uses the address in **R13**, even using tools from previous work [49], or when considering dueling recursive PHANTOMCALLS. Hence, we leave finding the disclosure gadget for the Zen 3 exploit as future work. We note however, that other kernel versions may include working disclosure gadgets.

On Zen 4, we successfully found the disclosure gadgets shown in Listing 8. To increase the Prime+Probe signal in step ① of breaking KASLR, out gadget loads 3 different offsets of our guessed kernel text region, as shown in Listing 8-top. To find the physmap base (i.e., step ③ of breaking KASLR), we use the disclosure gadget shown in Listing 8-mid. Listing 8-bottom shows the arbitrary data disclosure gadget, found using tools of previous work [49].

Dueling recursive PHANTOMCALLS. We do not find a disclosure gadget that leaks the physical address of our reload buffer using a location in our message buffer, i.e., step ② of breaking KASLR. We therefore leverage dueling recursive PHANTOMCALLS to complete this step, using the two disclosure gadgets shown in Listing 9. The second recursive PHANTOMCALL is triggered by a PHANTOMJMP on the instruction at Line 10 of Listing 7-bottom.

```
mov    rax,QWORD PTR [rbx+rax*8+0x20]   ; Load phys guess addr
mov    rbx,QWORD PTR [rbp-0x8]
leave
xor    edx,edx
mov    ecx,edx
mov    esi,edx
mov    edi,edx
ret                                 ; return into gadget below
```

```
add    rax,QWORD PTR [physmap]    ; Adds physmap base
mov    QWORD PTR [rax],rdx           ; Loads from physmap
```

Listing 9: Disclosure gadgets used on Zen 4 for finding the physical address of the reload buffer by loading the guess from memory (top, at offset `0xbf6dc6` of kernel text) and adding the physmap base to it, and dereferencing it (bottom, at offset `0xc0407` of kernel text).

Results. We evaluate INCEPTION on an AMD Zen 4 (Ryzen 7 7700X), with microcode version `0xa601201` and 16GB of RAM, running Linux 5.19.0-28-generic with all mitigations enabled. We run our attack 50 times, each time leaking 1KB of randomized data after a reboot. Of the 50 runs, we successfully break KASLR in 45 cases, using a median time of 168 seconds. In those cases, INCEPTION leaks data at a rate of 39 bytes/s, with an accuracy of 93.5%.

To find `/etc/shadow`, we again run INCEPTION in parallel on all 8 available cores. We attempt to locate `/etc/shadow` 10 times, each with a timeout of 3 hours, and reboot the machine after every attempt. Our results show that we are able to successfully leak the root password hash in 6 of the 10 runs, in a median of 40 minutes.

8 Alternative TTE variants

We have demonstrated how a variant of TTE can be leveraged on AMD systems to leak arbitrary data. In this section, we will discuss the security impact of other potential variants of TTE. We then systematically explore what TTE variants can be triggered on various Intel and AMD microarchitectures. We expect our exploration to motivate future work that looks for exploitable transient execution gadgets and effective mitigations that cover these new attack surfaces.

8.1 Exposing new attack surfaces with TTE

We lay out three scenarios that would allow arbitrary transient code execution despite mitigations, if the target microarchitecture allows for specific cases of TTE.

Firstly, conditional branches in the kernel may be followed by call instructions. We previously showed that on AMD, transiently executed call instructions triggered by BTB-misprediction can manipulate the state of the RSB. Therefore, being able to skew the direction of a conditional branch, the attacker may be able to inject an existing return site in the kernel (i.e., $\mathrm{TTE}_{\text{PHT-RSB}}$). On AMD Zen 3 and 4, we found that we do not need to overwrite all RSB entries to reliably trigger misprediction to transiently injected return

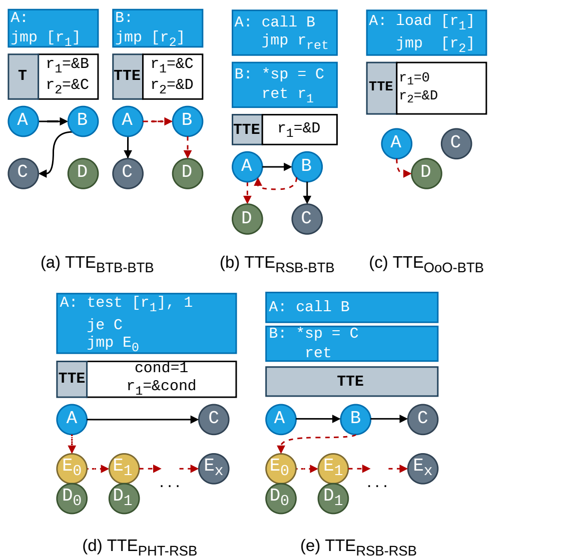

Figure 9: TTE using five different methods. The leak gadgets (D, green) and the training calls in (d) and (e) (E_i, yellow) are never architecturally executed. A barrier (C, gray) is used to stop speculation. Dashed red arrows indicate transiently executed paths.

targets. If a return target contains a disclosure gadget, this would allow an adversary to leak arbitrary data.

Secondly, on newer Intel microarchitectures with eIBRS, BHI [9] has shown that, although kernel branch predictions are isolated from user mode, an attacker can still influence the selection of previously used branch targets in privileged mode. Hence, to exploit BHI, the disclosure gadget must be a previously used branch target. TTE loosens this requirement. Instead of needing a disclosure gadget, we can use an indirect branch at a previously used branch target, which we in turn leverage to inject a disclosure gadget (i.e., TTE$_{\text{BTB-BTB}}$).

Lastly, conditional branch targets in the kernel may contain indirect branches when retpolines are disabled (e.g., on Intel CPUs that support eIBRS). While transient out-of-bound memory accesses are prevented by index masking [52], speculative type confusions may bypass such checks [29]. That is, if an attacker can skew the direction of a conditional branch, it may allow them to execute an indirect branch transiently. If the destination of the indirect branch is attacker-controlled, this would result in arbitrary transient execution whenever the indirect branch is executed architecturally as also discussed in Section 5.1 (i.e., TTE$_{\text{PHT-BTB}}$). Note that through BHI, the injected branch target may be reused by executing a different indirect branch in the kernel. Having established the scenarios in which TTE would bypass existing mitigations, we now evaluate different variants of TTE.

8.2 Testing for TTE variants

We discussed TTE$_{\text{PHT-BTB}}$ and TTE$_{\text{BTB-RSB}}$ in Section 5. We now explore other possible variants of TTE. Figure 9 describes the experiments that we designed to test for these other variants. Similar to experiments in Section 5, code locations A and B are used to manipulate the branch predictors, C is a *barrier target* and D is a *disclosure gadget*.

Model	Microarch.	Year	TTE$_{\text{BTB-BTB}}$	TTE$_{\text{PHT-BTB}}$	TTE$_{\text{RSB-BTB}}$	TTE$_{\text{OOO-BTB}}$	TTE$_{\text{BTB-RSB}}$	TTE$_{\text{PHT-RSB}}$	TTE$_{\text{RSB-RSB}}$
Ryzen 5 1600X	Zen 1	2017	✓	✓	✓	✓	✓	✓	✓
Ryzen 5 2600X	Zen+	2018	✓	✓	✓	✓	✓	✓	-
EPYC 7252	Zen 2	2019	✓	✓	✓	✓	✓	✓	✓
Ryzen 5 5600G	Zen 3	2019	✓	✓	✓	-	✓	✓	✓
EPYC 7413	Zen 3	2021	✓	✓	✓	-	✓	✓	✓
Ryzen 7 7700X	Zen 4	2022	✓	✓	✓	-	✓	✓	✓
i7-8700K	Coffee Lake	2017	✓	✓	✓	✓	-	-	-
i9-9900K	Coffee Lake R	2018	✓	✓	✓	✓	-	-	-
Xeon Silver 4314	Ice Lake	2021	✓	✓	✓	✓	-	-	-
i7-10700K	Comet Lake	2020	✓	✓	✓	✓	-	-	-
i7-11700K	Rocket Lake	2021	✓	✓	✓	✓	-	-	-
i7-12700K (P-core)	Golden Cove	2022	✓	✓	✓	✓	-	-	-
i7-12700K (E-core)	Gracemont	2022	-	-	-	-	-	-	-
i7-13700K (P-core)	Raptor Cove	2022	✓	✓	✓	✓	-	-	-
i7-13700K (E-core)	Gracemont	2022	-	-	-	-	-	-	-

Table 1: CPUs that are vulnerable speculative training of the BTB, i.e., TTE$_{\text{*-BTB}}$, and of the RSB, i.e., TTE$_{\text{*-RSB}}$.

TTE$_{\text{BTB}}$ experiments. To test for TTE$_{\text{BTB-BTB}}$, we first train the branch predictor to branch from A to B in a preparatory training step T. In the training in transient execution step TTE, we transiently execute B by changing the architectural branch target in A to the barrier target C. However, the previously injected B will be predicted, and we provide it with D as branch target. To test for TTE$_{\text{RSB-BTB}}$, we train the RSB to return to the instruction immediately following the call in A. We prevent this from architecturally executing by overwriting the actual return target on the stack (`sp`) with C. We expect the *jmp* r_{ret} in A to transiently execute with the branch target D. Finally, to test for TTE$_{\text{OOO-BTB}}$, the training branch source follows a load instruction that faults in the TTE step, because we pass it a null pointer. Because of OoO, the branch is transiently executed regardless. The CPU defers handling the fault to a later stage in the pipeline.

TTE$_{\text{RSB}}$ experiments. To test for TTE$_{\text{PHT-RSB}}$, we rely on conditional forward branches being predicted as non-taken by default. Therefore, in the TTE step we speculatively execute *jmp* E_0, which starts executing calls transiently. Likewise, to test for TTE$_{\text{RSB-RSB}}$, we again overwrite the architectural return target on the stack (`sp`) with C. We expect the gadgets E_i following A to transiently execute, pushing their return targets (D_i) to the RSB. Testing TTE$_{\text{OoO-RSB}}$ is challenging, since an invalid memory access as done for TTE$_{\text{OoO-BTB}}$ would require kernel-level page-fault handling, which trashes the RSB state. We therefore excluded this experiment for the RSB.

Results. Table 1 shows the results of running all TTE variants on the CPUs we have available in our lab. We note that certain experiments show weaker (yet distinct) signal on certain microarchitectures. This is a common artifact of constructing generic experiments, which can be overcome by fine-tuning them for the given microarchitecture. The results show that training of BTB in transient execution is feasible in most scenarios and microarchitectures. Exceptions are $\text{TTE}_{\text{OOO-BTB}}$ on the more recent AMD CPUs and $\text{TTE}_{*\text{-BTB}}$ on energy-efficient Intel cores embedded next to the high-performance cores in recent Intel processors. The results further show that all AMD CPUs in our lab are susceptible to the training of the RSB in transient execution, although the injected entry is not always used, as discussed in Section 5.2. On Intel, we are unable to transiently train the RSB on any of the considered microarchitectures.

Discussion. Our results show that the previously described attack scenarios ($\text{TTE}_{\text{PHT-RSB}}$ on AMD, $\text{TTE}_{\text{BTB-BTB}}$ on Intel, and $\text{TTE}_{\text{PHT-BTB}}$ on both) are realistic on the microarchitectures that we considered, and future mitigations should consider their attack surfaces.

9 Mitigation

We discuss mitigation strategies against INCEPTION in this section. Our analysis shows that a complete mitigation of INCEPTION requires hardware modification for stopping TTE, but the attack surface can be reduced by flushing the branch predictor state on privilege transitions on certain microarchitectures. Unfortunately, this introduces a significant performance penalty as shown by our evaluation. We first present our analysis of possible mitigations before discussing flushing branch prediction state and its performance impact.

9.1 Analysis of possible mitigations

Synchronization. CPU vendors recommend serializing instructions, such as `lfence`, to stop transient execution of malicious control flow. While synchronization can stop the $\text{TTE}_{\text{PHT-}*}$ variants, it is insufficient against INCEPTION since PHANTOM speculation enables hijacks of arbitrary instructions.

Address Space Isolation (ASI). There is an ongoing effort to prevent secrets from being present in the kernel address space as part of a broader industry effort for a more principled mitigation of transient execution attacks. However, currently ASI may only reduce the attack surface; the entire address space must still be mapped in to handle many interactions, where microarchitectural buffer flushes are necessary [44]. Unfortunately, adequate flushing mechanisms are at the time of writing not available, as we find in Section 9.2.

Avoid transient training. Updating branch predictors using transient control flow is the root cause of TTE. If branch predictors were only updated at retirement, our INCEPTION would be unsuccessful. There might exist undocumented MSR registers that control the behavior of the CPU frontend, such that early branch predictor updates are prevented from being dispatched. This would mitigate all TTE attacks. At time of writing, we are unaware whether such a functionality exists on affected CPUs. AMD has previously disclosed such undocumented MSR registers that can be accessed to toggle features or reconfigure CPU properties. *SuppressBPOnNonBr* MSR bit in AMD documents, for mitigating PHANTOMJMPS [50], is one such example [6]. Counter-intuitively and unfortunately, INCEPTION works using PHANTOMJMP, regardless setting this bit, as we discussed in Section 6.

Speculative BPU structures. By designing dedicated speculative variants of branch predictor structures, predictions do not become visible outside of the transient window in which they were inserted. As an example, our results on Intel microarchitectures suggest that they implement a speculative RSB. By using speculative variants of all branch predictor structures, TTE attacks can be prevented by discarding the transiently updated structure. However, while the RSB typically contains only 16 or 32 entries, the BTB typically contain thousands of entries. Creating a speculative counterpart for every predictor structure is thus a costly operation, and unlikely to be implemented in practice.

Isolating the branch predictor state. Some existing hardware mitigations, such as Intel eIBRS, stop the branch predictions learned in a lower privilege mode from being used in a higher one. While this reduces the attack surface of TTE, and in particular INCEPTION, other TTE variants remain possible in principle, as discussed in Section 8.1. Investigating the feasibility of such attacks is an interesting direction for future research. For mitigating INCEPTION on affected AMD CPUs without eIBRS, a complete flushing of the branch predictor state is an alternative option.

9.2 Full predictor buffer sanitization

Creative spot mitigations continue to fall short in face of newer attacks. For example, retpoline [47] is bypassed by Retbleed [49], and now jmp2ret [6] by INCEPTION. Furthermore, these mitigations introduce ubiquitous source code changes and configuration parameters, which affect the maintainability of the OS kernel. Instead of additional spot mitigations, issuing IBPB on privilege level elevation may provide an in-depth mitigation against PHANTOM speculation and INCEPTION, however with a high performance cost. The Xen Project Security Team anticipated that *jmp*2*ret* was inadequate and enable IBPB-on-entry by default for the Xen hypervisor to mitigate PHANTOM vulnerabilities [46]. Using IBPB-on-entry rests on the assumption that all potentially harmful branch prediction state is sanitized.

We evaluate the performance impact of IBPB-on-entry on Linux using the UnixBench test suite[1]. We run the test suite 5 times with and without the IBPB-on-entry enabled.

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

Micro-architecture	Model	Microcode	Performance overhead		IBPB effect		
			single-core	*multi-core*	*cycles (median)*	*flush direct*	*flush indir*
Zen 1	Ryzen 5 1600X	`0x8001137`	239.2 % / 234.3 % *	198.4 % / 216.9 % *	8,803 cycles	✓	✓
Zen +	Ryzen 5 2600X	`0x800820d`	226.6 % / 205.0 % *	183.1 % / 204.0 % *	8,196 cycles	✓	✓
Zen 2	Ryzen 5 3600X	`0x8701021`	130.1 %	95.2 %	1,306 cycles	✓	✓
Zen 2	EPYC 7252	`0x8301038`	128.6 %	93.1 %	1,306 cycles	✓	✓
Zen 3	Ryzen 5 5600G	`0xa50000c`	35.05 %	29.35 %	738 cycles	✗	✓
Zen 4	Ryzen 7 7700X	`0xa601201`	59.90 %	87.33 %	962 cycles	✗	✓

* : Simultaneous Multi-Threading (SMT) disabled

Table 2: Performance overhead of single- and multi-core benchmarks with the IBPB-on-entry mitigation, including the cost of issuing one IBPB. We benchmark with and without SMT enabled where relevant. IBPB only concerns indirect branches on Zen 3 and 4.

We compute median results for each of the 12 tests in the test suite, from which we then derive a cumulative geometric mean. The final result is a score analogous to number of operations per time unit. Hence, we denote *performance overhead* as $score_{baseline}/score_{ibpb} - 1$. Furthermore, we measure the median number of clock cycles for issuing IBPB (using the precise APREF clock cycle counter [7]) over 1 M samples.

Table 2 shows the results of our benchmarks. Because Zen 1 and Zen + do not have STIBP support, for a complete mitigation, we also benchmark with SMT disabled. An attacker could otherwise poison the BTB from a sibling thread after the IBPB has been issued. Without disabling SMT, these parts are also vulnerable to existing attacks, like Retbleed. However, SMT is left enabled on Linux regardless.

IBPB is an expensive operation, most particular for Zen 1(+), but it is necessary for a complete mitigation of INCEPTION. Surprisingly, for Zen 3 and Zen 4, IBPB is substantially cheaper, and has suspiciously low performance impact. This observation led us to furthermore check the scope of IBPB on the evaluated systems. We discover that IBPB, which sanitized branches of all types on Zen 1(+) and Zen 2, no longer does so for Zen 3 and Zen 4. To determine the impact of this for mitigating INCEPTION, we execute an architectural direct recursive call, while catching the stack overflow fault. Our results show that this primes the BTB with a direct recursive PHANTOMCALL which is not flushed by IBPB. Consequently, we conclude that INCEPTION can *circumvent IBPB-on-entry on Zen 3 and Zen 4 systems* by injecting all PHANTOMJMP and PHANTOMCALL predictions using direct branches instead of indirect ones. AMD does not recommended IBPB-on-entry as a mitigation against INCEPTION, which likely is for this reason.

10 Related work

The *confinement problem* is a known issue in computer security since the 1970s [34]. 20 years later, Kocher was among the first to consider this problem in the context of microarchitectural timing attacks [31]. Over the past few decades, microarchitectural side channels have been studied on the CPU caches [16, 22, 39, 41, 53], μop caches [42], execution units [8, 10, 21, 32] and branch predictors [2, 17, 18].

Transient execution attacks. Kocher et al. [30] showed that a mispredicted branch combined with a CPU cache side channel can be used to leak arbitrary memory, resulting in Spectre attacks. Spectre variant 1 (also known as Spectre-PHT) bypasses array bounds checks through mispredicted conditional branches and Spectre variant 2 (also known as Spectre-BTB) injects malicious branch targets into the BTB for indirect branch instructions. In addition, Lipp et al. [35] showed that OoO pipelines in modern CPUs execute instructions following a faulting instruction, which could result in transient use of unauthorized memory if present in the L1d cache. These are the two main categories transient execution attacks as Canella et al. categorized them, and they furthermore categorized branch predictor training methods as *in-place* and *out-of-place* [11]. Following Spectre, Maisuradze et al. [36], Koruyeh et al [33], and Wikner et al. [48] demonstrated Spectre attacks via the RSB. Similar to [48] and [36], INCEPTION works by overflowing the RSB. However, unlike previous attacks, INCEPTION does so in a transient execution window which expands the attack surface of Spectre. RSB training in transient execution was explained however not used in [33]. BTB training in transient execution was used by [49], but the purpose was to suppress page faults, and not to increase the attack surface.

Spectre attack arms-race. To mitigate Spectre-BTB attacks against the kernel, Turner et al. invented retpolines [47], and AMD invented lfence-retpoline [3]. Intel invented IBRS, and enhanced IBRS for newer CPUs, to prevent use of branches injected from a lower privilege level to be used at a higher one [12, 40]. These defenses have all been broken. Milburn et al. [37] showed that lfence-retpolines are vulnerable to Spectre-BTB, because certain workloads on the sibling thread could extend the speculation window of the victim. Barberis et al. [9] presented BHI, a confused-deputy attack against the BTB, forcing branch target injection within the privileged context to bypass the eIBRS mitigation. Wikner and Razavi [49] showed that the RSB-backed return target predictor could be bypassed to use the branch target predictor instead, bypassing the deployed retpoline mitigations with their Retbleed attack. Moreover, they showed that AMD CPUs vulnerable to Retbleed are also vulnerable to PHANTOMJMPS [51]. In this paper we introduced PHANTOMCALLS, and showed that they can be transiently executed inside a PHANTOMJMP to manipulate the RSB.

To mitigate Retbleed and BHI on Intel CPUs, Linux combines (enhanced) IBRS and retpolines. AMD, who is unaffected by BHI, proposed *jmp2ret* to mitigate Retbleed [6]. This paper shows that jmp2ret can be broken with recursive PHANTOM speculations that cause the RSB to overflow.

11 Conclusion

We introduced Training in Transient Execution (TTE) in this paper. TTE expands the attack surface of transient control-flow hijacks by enabling the attacker to train the BTB and RSB in the kernel context. We further introduced PHANTOMCALL, a new PHANTOM primitive that enables TTE without relying on complex gadgets in the kernel, by using the CPU as a confused deputy. Our end-to-end exploit, called INCEPTION, uses a recursive PHANTOMCALL to create an infinite hardware loop in transient execution that poisons many RSB entries with an attacker-controlled return address. In 6 out of 10 trials, INCEPTION can leak the contents of `/etc/shadow` in 40 minutes on AMD Zen 4 despite all existing and recent hardware and software mitigations against speculative control-flow hijacks. We expect our insights to motivate future work to explore TTE's new attack surfaces further and consider efficient mitigations that protect against them.

Acknowledgments

We thank the anonymous reviewers for their feedback. We would also like to thank Andrew Cooper of XenServer for the discussions around PHANTOMJMP and Michele Marazzi for proofreading the paper. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] The Linux kernel user's and administrator's guide: Spectre Side Channels. https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst. Accessed on 8.2.2023.

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *CCS*, 2007.

[3] AMD. Amd64 technology indirect branch control extension. https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf, 2018. Accessed on 8.2.2023.

[4] AMD. Software optimization guide for amd family 17h models 30h and greater processors, March 2020.

[5] AMD. Software optimization guide for amd family 17h models 30h and greater processors, March 2020.

[6] AMD. Technical guidance for mitigating branchtype confusion. revision 3.0. https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion.pdf, 2022. Accessed on 8.2.2023.

[7] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf, January 2023. accessed on 1.2.2023.

[8] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P*, pages 623–639. IEEE, 2015.

[9] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *SEC*. USENIX, 2022.

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *CCS*. ACM, 2019.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *SEC*. USENIX, 2019.

[12] Intel Corp. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2018. Accessed on 8.6.2023.

[13] Intel Corp. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf, 2018. Accessed on 8.2.2023.

[14] Intel Corp. Speculative Execution Side Channel Mitigations. https://www.intel.com/content/ww

w/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, 2018. Accessed on 8.2.2023.

[15] Intel Corp. Post-barrier Return Stack Buffer Predictions / CVE-2022-26373 / INTEL-SA-00706. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/post-barrier-return-stack-buffer-predictions.html, 2022. Accessed on 8.2.2023.

[16] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *SEC*. USENIX, 2017.

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

[18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*. ACM, 2018.

[19] Simcha Gochman, Nicolas Kacevas, and Farah Jubran. Method and apparatus for implementing a speculative return stack buffer, October 12 1999. US Patent 5,964,868.

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

[21] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[22] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[23] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, 2017.

[24] Jann Horn. Issue 1528: Speculative Execution, Variant 4: Speculative Store Bypass. https://bugs.chromium.org/p/project-zerot/issues/detail?id=1528, 2018. Accessed on 11.6.2023.

[25] Intel Corp. Return Stack Buffer Underflow / Return Stack Buffer Underflow / CVE-2022-29901, CVE-2022-28693 / INTEL-SA-00702. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/return-stack-buffer-underflow.html, 2022. Accessed on 11.6.2023.

[26] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. In *NDSS*, 2022.

[27] Stephan J Jourdan, John Alan Miller, and Namratha Jaisimha. Return address stack including speculative return address buffer with back pointers, May 24 2005. US Patent 6,898,699.

[28] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[29] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *SEC*. USENIX, 2021.

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

[31] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[32] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *SEC*, pages 69–81. USENIX, 2017.

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

[34] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

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

[36] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *CCS*. ACM, 2018.

[37] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You cannot always win the race: Analyzing the lfence/jmp

mitigation for branch target injection. *arXiv preprint arXiv:2203.04277*, 2022.

[38] Alex Murray. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047. Accessed on 8.2.2023.

[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*. Springer, 2006.

[40] Kim Phillips. LKML: [PATCH 0/3] x86/speculation: Support Automatic IBRS, 2022.

[41] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *CCS*. ACM, 2021.

[42] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. In *ISCA*, pages 361–374. IEEE, ACM, 2021.

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

[44] Junaid Shahid and Ofir Weisse. [RFC PATCH 00/47] Address Space Isolation for KVM. https://lkml.org/lkml/2022/2/23/14, 2022. accessed on 12.06.2023.

[45] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core. In *ISSCC*. IEEE, 2020.

[46] Xenproject.org Security Team. Retbleed - arbitrary speculative code execution with return instructions. https://xenbits.xen.org/xsa/advisory-407.html, 2022. Accessed on 8.2.2023.

[47] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. Accessed on 8.2.2023.

[48] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks. In *WOOT*. IEEE, 2022.

[49] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary Speculative Code Execution with Return Instructions. In *SEC*. USENIX, August 2022.

[50] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Addendum to Retbleed: Arbitrary Speculative Code Execution with Return Instructions. August 2022.

[51] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting Decoder-detectable Mispredictions. August 2023.

[52] Dan Williams. LKML: [PATCH v6 02/13] array_index_nospec: sanitize speculative array dereferences. https://lkml.org/lkml/2018/2/16/65, 2018. Accessed on 8.2.2023.

[53] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *SEC*. USENIX, 2014.

[54] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991.

[55] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*, pages 667–682. ACM, 2020.

[56] Peter Zijlstra and Thomas Gleixner. [PATCH v3 00/59] x86/retbleed: Call depth tracking mitigation. https://lkml.org/lkml/2022/9/15/427, 2022. Accessed on 8.2.2023.

[57] Peter Zijlstra and Borislav Petkov. Re: [PATCH] x86/CPU/AMD: Rename the spectral chicken. https://lkml.org/lkml/2023/4/25/875, 2023. Accessed on 11.6.2023.