ETH zürich

ETH Library

Racing Against the Decoder

Leaking Arbitrary Memory with Phantom Speculation and Training in Transient Execution

Master Thesis

Author(s):
Trujillo, Daniel

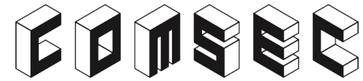
Publication date:
2023

Permanent link:
https://doi.org/10.3929/ethz-b-000629949

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Racing Against the Decoder

Leaking Arbitrary Memory with Phantom Speculation and Training in Transient Execution

Master Thesis

Author: Daniël Trujillo

Tutor: Johannes Wikner

Supervisor: Prof. Dr. Kaveh Razavi

November 2022 to May 2023

Abstract

To avoid pipeline stalls, microarchitectures perform various types of speculative execution, improving performance considerably. However, speculative execution may lead to information disclosure when microarchitectural buffers are not sufficiently isolated between security domains. This thesis investigates two classes of transient execution attacks mostly unexplored by previous work, and shows that the consequences of overlooking them are severe. Our findings enable real-world information leakage, bypassing state-of-the-art mitigations deployed in the Linux kernel.

First, we systematically explore PHANTOM, a class of transient execution attacks that rely on transient execution arising from arbitrary instructions, i.e. decoder-detectable mispredictions, bypassing mitigations that selectively protect instructions of certain branch types. We derandomize physmap KASLR using PHANTOM on Zen 1 and Zen 2, allowing us to determine virtual-to-physical address mappings of our unprivileged user space program. Furthermore, we prove it to be practical to leak arbitrary kernel memory with PHANTOM, as suggested in prior work, by presenting a PoC that abuses a realistic dummy MDS gadget.

Second, we introduce a class of transient execution attacks that do their **T**raining in **T**ransient **E**xecution (TTE), bypassing sanitization techniques. We investigate different TTE variants, and find some which are of concern. Particularly, we discover that combining TTE with PHANTOM allows us to abuse the CPU as a confused deputy that poisons the RSB transiently, letting us hijack the transient control flow of return instructions on all Zen microarchitectures. We construct our end-to-end exploit INCEPTION, which leaks arbitrary kernel memory at a rate of 39 bytes/s with an accuracy of 93.5% on AMD's newest flagship Zen 4, despite all deployed mitigations against transient execution attacks, including AutoIBRS. Furthermore, we show that INCEPTION is capable of leaking the Linux root password hash by locating `/etc/shadow` in a median of 40 minutes, in 6 of our 10 attempts.

This thesis proves that current mitigations against transient execution attacks are insufficient. To prevent INCEPTION, we propose to fully flush the BTB state on privilege switches, which comes with a substantial performance penalty. Future microarchitectural designs should consider PHANTOM and TTE to allow more efficient mitigations against these type of attacks.

Acknowledgments

I would first like to thank my professor, Prof. Dr. Kaveh Razavi, and my supervisor, Johannes Wikner. Being your student is like winning the lottery, and I am incredibly grateful for being one of the lucky ones. Kaveh, I do not believe it is possible for a professor to be more supportive than you. Johannes, your work and code has taught me a lot. Working with both of you was always fun, interesting, exciting. Without you, pursuing this degree would not nearly have been as fruitful as it was now. I am thinking back with happy memories to our brainstorm sessions, during which we discussed crazy exploitation ideas. And to the two times we worked insane hours, fueled by sugar, to get the papers in good shape for submission. I could not be more happy with – and proud of – the results that came out of this. Thank you, Kaveh and Johannes.

Second, I would like to thank Dr. Ben Gras. I feel very lucky to have worked with you at Intel during my studies; I could not have asked for a better mentor during my internship. You are always supportive of me, and I can always ask you any question, whether it is related to work or not. Thank you, Ben.

Lastly, I thank my family. Your support and generosity have made it possible for me to study at ETH Zürich. Thank you mom, dad, brother, sisters and grandparents. I am very grateful.

Daniël Trujillo
May 2023

Contents

1 Introduction **1**

2 Background **4**

2.1 Pipelining . . . 4
2.2 Speculative execution . . . 4
2.3 Modern branch prediction . . . 5
2.4 Transient execution attacks . . . 6
2.5 Attack mitigations . . . 6

3 Phantom: Exploiting Decoder-detectable Mispredictions **7**

3.1 Threat model . . . 7
3.2 Overview . . . 7
3.3 Phantom speculation . . . 9
3.3.1 Observation Channels . . . 9
3.3.2 Triggering mispredictions . . . 10
3.4 Exploitation Primitives . . . 12
3.4.1 Attacker primitives . . . 12
3.4.2 SuppressBPOnNonBr and AutoIBRS . . . 14
3.4.3 Covert Channel . . . 14
3.5 Exploitation . . . 15
3.5.1 Breaking physmap KASLR . . . 15
3.5.2 Overcoming noise . . . 16
3.5.3 Leaking kernel memory . . . 16
3.6 Mitigation . . . 18
3.6.1 Hardware mitigations . . . 18
3.6.2 Software mitigations . . . 18

4 Inception: Exposing New Attack Surfaces with Training in Transient Execution **20**

4.1 Threat Model . . . 20
4.2 Overview . . . 20
4.3 Training in Transient Execution . . . 21
4.3.1 Training BTB in transient execution . . . 22
4.3.2 Training RSB in transient execution . . . 23
4.4 TTE and Phantom . . . 26
4.4.1 Chicken out from Phantom . . . 26
4.4.2 Exploring the limits of Phantom . . . 26
4.4.3 PhantomCall . . . 28
4.5 Inception . . . 29
4.5.1 Recursive PhantomCall . . . 29
4.5.2 Dual-threaded mode . . . 32

4.5.3 Exploit design 32
4.5.4 Dueling recursive PhantomCalls 33
4.5.5 Victim return instruction 34
4.5.6 Derandomizing KASLR 34
4.5.7 Leaking kernel memory 35
4.5.8 Inception on Zen 1(+)/2 35
4.5.9 Inception on Zen 3 35
4.5.10 Inception on Zen 4 36
4.5.11 Leaking root password hash 38
4.5.12 AutoIBRS 38
4.5.13 Optimizations 38
4.6 Alternative TTE variants 39
4.6.1 Exposing new attack surfaces with TTE 39
4.6.2 Testing for TTE variants 39
4.7 Mitigation 41
4.7.1 Analysis of possible mitigations 41
4.7.2 IBPB-on-entry 42

5 Related Work 44

6 Conclusion 46

Bibliography 47

A Phantom: Collision with kernel addresses I

B Phantom: Breaking code KASLR III

Paper Submissions

Results in this thesis have been submitted to conferences for publication. The contents of the below papers form this thesis:

Wikner J., Trujillo, D. and Razavi, K. **Phantom: Exploiting Decoder-detectable Mispredictions**. 56th IEEE/ACM International Symposium on Microarchitecture (MICRO '23). [*Under submssion*]

Trujillo D., Wikner J. and Razavi, K. **Inception: Exposing New Attack Surfaces with Training in Transient Execution**. 32nd USENIX Security Symposium (USENIX Security '23). [*Accepted*]

The results of these papers appear in Chapter 3 and Chapter 4 respectively. My contributions are listed at the beginning of each chapter.

To my family

Chapter 1

Introduction

A particularly dangerous class of microarchitectural attacks are transient execution attacks, which enable arbitrary information disclosure in many cases of interest [9, 10, 30, 19, 31, 58, 36, 56]. Over the past years, variants of these so-called Spectre attacks have called for a plethora of mitigations, both in software and hardware [51, 7, 25, 27, 59]. In this thesis we wish to investigate whether there exists remaining transient execution attack surface that previous work has overlooked. To do so, we look in detail at classes of transient execution attacks mostly unexplored by previous work.

Our results are alarming, showing that current mitigations are fundamentally insufficient. First, we explore a class of transient execution attacks that rely on decoder-detectable misprediction, to which we refer as Phantom attacks, and show that these attacks allow us to leak address space information. We furthermore introduce a new class of transient execution attacks that do their **T**raining in **T**ransient **E**xecution (TTE), enabling an attacker to leak data with the help of gadgets in the victim code, despite current mitigations against transient execution attacks. However, these gadgets are nontrivial to find. Instead, we show how Phantom enables TTE with few requirements on the victim code, using the CPU as a confused deputy. We build our end-to-end exploit Inception that leaks arbitrary kernel memory on fully-patched AMD systems.

Transient execution attacks and mitigations. Transient execution attacks trigger speculative execution of code that leaves secret-dependent traces in microarchitectural buffers such as the cache. They do so by manipulating Branch Prediction Unit (BPU) structures, which are shared between mutually untrusted parties. For example, an attacker may inject a branch target into the BPU's Branch Target Buffer (BTB) [30] or manipulate the state of the Return Stack Buffer (RSB) [31]. Mitigations in many shapes and forms have been deployed over the past years to protect against these attacks, focused on properly isolating BPU states while minimizing performance penalties at the same time. Sanitization of BPU structures is one class of mitigations, protecting privileged software from transient execution attacks. For example, RSB stuffing protects privileged code such as the kernel from having return target predictions being influenced by an unprivileged attacker [12]. In its first chapter, this thesis investigates a class of transient execution attacks that rely on speculation very early in the pipeline, before decoding has finished.

Decoder-detectable mispredictions. The Instruction Fetch (IF) stage of a CPU fetches instructions from the instruction cache, feeding them to the Instruction Decode (ID) stage. Until the ID stage finished decoding potential branches, it is unclear which address the IF stage should fetch next. Trivially, the IF stage may stall until the next instruction pointer is nominal. Instead, AMD CPUs speculate on the *existence* of branches by consulting the BTB before decode, as shown in our work prior to this thesis [57]. This means that AMD CPUs are susceptible to decoder-detectable mispredictions, as later confirmed by AMD in an

advisory [7]. Since the ID stage is positioned in the frontend, this thus gives rise to *frontend resteers*, deviating from the majority of literature on speculative execution, which describe backend resteers exclusively [30, 31, 9, 36].

Phantom. In this thesis, we refer to the class of transient execution attacks that rely on decoder-detectable mispredictions as PHANTOM. Whereas most previous transient execution attacks give rise to long transient windows, PHANTOM results in very short transient windows. Specifically, PHANTOM transient windows are just long enough to fit execution of only one or a few instructions, or in some cases only instruction fetches. The first concrete research aim of this thesis is to systematically explore this class of attacks, and to investigate to what extent PHANTOM attacks could lead to information leakage, despite their short transient windows:

> Research Question **RQ1.**
>
> How can we systematically explore PHANTOM, and what information leaks does this class of attacks enable?

We introduce observation channels that determine very precisely whether PHANTOM speculation has occurred, using which we construct experiments to establish when PHANTOM is triggered. We measure the covert-channel characteristics of PHANTOM, and we reveal that the short transient windows it triggers are sufficient to efficiently derandomize physmap KASLR on certain AMD microarchitectures. Furthermore, our work prior to thesis suggests that PHANTOM attacks can leak arbitrary kernel memory, if MDS gadgets are present in the victim code [57, 7]. This thesis proves such attacks to be practical, by presenting a PoC with a realistic dummy MDS gadget that leaks arbitrary memory from the Linux kernel. Although these gadgets are ubiquitous, they are actively searched for, given the known attack surface they introduce on Intel microarchitectures, which are vulnerable to MDS. Our second research question therefore is:

> Research Question **RQ2.**
>
> Is there unexplored attack surface of transient execution attacks that does not rely on gadgets which are actively being patched?

Training in Transient Execution. We introduce a new class of attacks that do their **T**raining in **T**ransient **E**xecution (TTE). With TTE, we show that current approaches of isolating BPU states through sanitization are fundamentally flawed, because we enable training *after* sanitization has happened. By forcing transient execution of branches during victim execution, BPU structures are manipulated in the victim security context using TTE. We evaluate TTE variants that transiently train the BTB and RSB, i.e. TTE_{BTB} and TTE_{RSB}, on both AMD and Intel microarchitectures. We find that TTE increases the known attack surface in the presence of certain gadgets in victim code, effectively abusing the victim code as a confused deputy. Although some of these gadgets were previously not seen as a security threat, they are also not necessarily trivial to find. Our third research question is therefore whether we can loosen the requirements of these gadgets by combining TTE with PHANTOM, and if so, to which extent:

> Research Question **RQ3.**
>
> To what extent can we reduce TTE requirements on the victim code using PHANTOM, if at all?

TTE and PHANTOM. Our research efforts reveal that PHANTOM allows us to abuse AMD CPUs as a confused deputy instead of the victim code, performing TTE on our behalf. Specifically, we discover that executing a PHANTOMCALL, i.e. a predicted call for an arbitrary instruction, transiently manipulates the RSB. By furthermore executing this PHANTOMCALL inside a PHANTOMJMP, we can transiently inject an arbitrary address into the RSB. We thus enable

TTE_{RSB} with barely any requirements on the victim code. Counterintuitively, the victim code may even be free of call instructions.

Inception. To show the immediate threat of combining TTE_{RSB} with Phantom, we construct Inception, a PoC exploit that leaks arbitrary data from the kernel on AMD Zen 1(+), Zen 2 and Zen 4 microarchitectures. Developing an exploit with a single PhantomCall is challenging, due to the exact behavior of the RSB and the need for deep return stacks to reach the poisoned entry. Instead, Inception triggers an infinite transient hardware loop that pushes onto the RSB by constructing a recursive PhantomCall, enabling reliable exploitation. Concerningly, we discover that Inception is not stopped by any of the currently deployed software- and hardware mitigations. Even AutoIBRS, a brand-new feature available on AMD Zen 4, fails to address Inception. Our research shows that a full flush of the BTB state on kernel entry is necessary to prevent our attack, and we show that this unfortunately comes with a hefty performance penalty.

Contributions. In summary, the contributions of this thesis are as follows:

- Exploring decoder-detectable mispredictions, or Phantom attacks, on AMD Zen microarchitectures. We show that mitigations are only partially effective, and we measure covert channel characteristics of Phantom.
- Derandomizing physmap KASLR on AMD Zen 1 and Zen 2, and implementing a PoC that leaks arbitrary kernel memory using a realistic dummy MDS gadget.
- Introducing Training in Transient Execution (TTE) and the evaluation of two variants, TTE_{BTB} and TTE_{RSB}, on both Intel and AMD microarchitectures.
- Discovery of PhantomCall, and an analysis of how it can corrupt a large part of the RSB with an arbitrary attacker-controlled address by establishing an infinite transient hardware loop on AMD Zen microarchitectures.
- Construction of Inception, an end-to-end exploit that leaks arbitrary data on AMD Zen 1(+), Zen 2 and Zen 4, despite all latest mitigations. Inception continues to be effective even with AMD's brand-new AutoIBRS feature enabled.

Responsible disclosure. In February 2023, we disclosed Inception to Intel, AMD and Greg KH of the Linux foundation. AMD requested an embargo due to the need for developing microcode updates and software patches. This embargo ends on August 8, 2023.

We also informed AMD of the new findings regarding Phantom in May 2023, but at the time of writing no embargo has been requested.

Chapter 2

Background

2.1 Pipelining

To prevent underutilization of the resources of a CPU, modern processors implement pipelining, a technique where instruction execution is split up in several stages that execute concurrently. Stages include Instruction Fetch (IF), Instruction Decode (ID) and Execute (EX). After an instruction has been fully processed by each of the stages, it is committed to the architectural state, an event to which one refers as *instruction retirement*. Without pipelining, a CPU's clock cycle needs to be sufficiently long to allow an instruction to complete in its entirety. The clock cycle would thus be lower bounded by the instruction that takes the longest. Using pipelined execution, the clock cycle can be decreased as long as it accommodates the longest-taking pipeline stage. Since stages operate concurrently, the CPU can execute multiple instructions simultaneously, significantly increasing instruction throughput. Figure 2.1 shows an example of a pipelined CPU, where multiple instructions execute concurrently.

2.2 Speculative execution

Instructions executed concurrently are not necessarily independent. An instruction which has not yet retired may provide crucial information for the execution of the next instruction. For example, a branch instruction's target may be dependent on a preceding compare instruction. To make things worse, this compare instruction may depend on slow memory loads.

A simple option is to wait until the instructions have been retired, i.e. we can simply *stall*

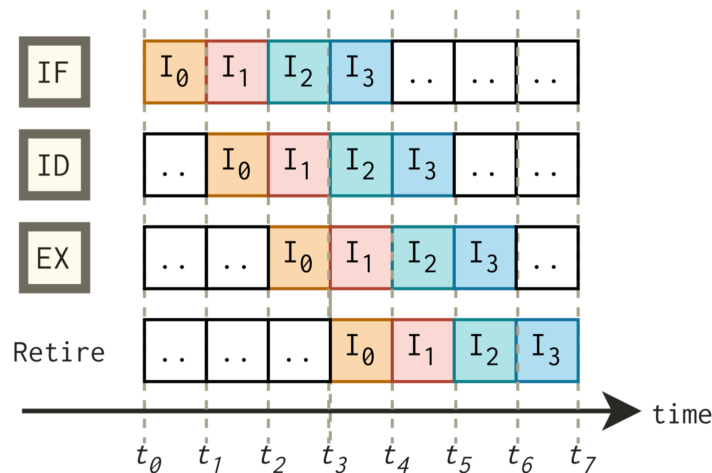


Figure 2.1: A pipelined processor. Instructions I_0, I_1, I_2 and I_3 execute concurrently.

the pipeline. However, this would considerably harm instruction throughput. Instead, modern processors *speculate* on instruction behavior, through most notably branch prediction. Modern processors also speculative on the outcome of non-branch instructions, with techniques such as predictive store forwarding [3]. If the speculation turns out to be correct, we have avoided unnecessary stalling. If the speculation was wrong, we can simply discard the results and *resteer* to the correct execution path.

For example, instruction I_1 in Figure 2.1 may be a branch whose target depends on the outcome of instruction I_0. Instead of stalling the pipeline from t_2 until instruction I_0 retires at t_3, we speculate on the outcome of the branch and start fetching I_2. If it turns out that instruction I_2 was incorrectly brought into the pipeline, we need to resteer.

Modern processors have complex pipelines with many more stages than the example in Figure 2.1. Because of these *deep* and complex pipelines, it may take hundreds of clock cycles before we can judge whether speculation was correct or not. This time frame is commonly referred to as the *speculation window*. Incorrect speculative execution is referred to as *transient execution*, and its speculation window is sometimes called a *transient window*.

2.3 Modern branch prediction

Modern CPUs embody a Branch Prediction Unit (BPU) residing in the frontend of the CPU to help perform accurate branch prediction [4, 5, 24]. The task of the BPU is to inform the IF stage about the most likely path of execution in the near future. Inside the BPU, several subcomponents contribute to a prediction, as shown in Figure 2.2. A Branch Target Buffer (BTB) contains entries with previously seen branch targets, to be used for branch prediction. BTB entries are selected using an indexing function, computed on the virtual address of the branch. Since branches may behave differently depending on program execution context, the BPU could keep track of multiple targets for a single entry. Program execution history, captured by the Branch History Buffer (BHB), may be used to select the most probable branch target. To update the BHB upon execution of a branch, a hash of the branch source address and branch target address is used. Modern processors make use of complex algorithms to reach an accurate prediction. For example, AMD Zen CPUs make use of the TAGE algorithm [48].

Since return instructions are typically paired with a preceding call instruction, BPUs implement a dedicated Return Stack Buffer (RSB) that mimics the program stack to accommodate more accurate return target predictions. Upon the execution of a call instruction, the address of the next instruction is pushed onto the RSB. When a return instruction is encountered, the BPU pops from the RSB to obtain a target address for speculative execution.

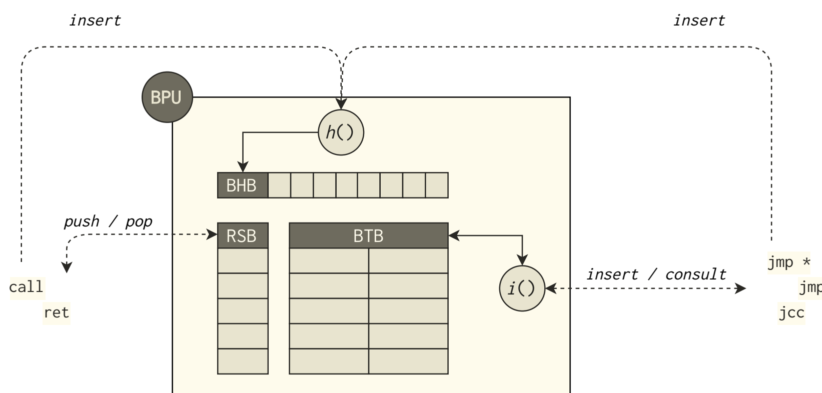


Figure 2.2: An overview of a modern BPU. Call and return instructions interact with the RSB, while other branches use the BTB. All branches potentially update the BHB.

2.4 Transient execution attacks

Although speculative execution has been ubiquitous for decades in commercial processors, its security risks were poorly understood until very recently. In 2018, researchers discovered Spectre, an attack that leaks sensitive data by triggering calculated branch predictions across security contexts [30]. Spectre forces transient execution of a so-called *disclosure gadget*, a snippet of code that leaves secret-dependent traces in the cache or other microarchitectural buffers. An attacker can infer these secrets using side channels such as Flush+Reload [60] and Prime+Probe [39]. Both Intel and AMD microarchitectures have been deemed vulnerable to various Spectre variants. Spectre-BTB injects a target in the BTB for an indirect branch. Spectre-PHT forces misprediction of the direction of a conditional jump. Spectre-RSB triggers misprediction of a return instruction by desynchronizing the RSB and the actual return sites on the stack [31].

Even recently, new transient execution attack surface has been discovered. Branch History Injection (BHI) allows an attacker to influence branch predictions by manipulating the BHB state only [9]. Retbleed shows that BTB predictions are also served for return instructions under certain circumstances [56]. On Intel, this occurs when there are no entries left in the RSB, i.e. upon an *RSB underflow*. Retbleed on AMD abuses confusion in the BPU where the BTB provides a prediction for a return instruction instead of the RSB. A class of transient execution attacks that trigger branch predictions for arbitrary instructions, commonly referred to as PhantomJmps, was discovered shortly after on AMD microarchitectures [57]. Retbleed is part of this class, to which AMD refers as Branch Type Confusion [7].

2.5 Attack mitigations

Transient execution attacks have triggered a plethora of mitigations, implemented in both software and hardware.

Software mitigations. *Retpolines* replace indirect branches with returns, to prevent indirect branch predictions, i.e. Spectre-BTB [2, 26, 51]. *RSB stuffing* overwrites the contents of the RSB upon a security context switch to prevent predictions from being influenced by an attacker, mitigating Spectre-RSB [49]. To protect against Spectre-PHT, *index masking* prevents speculative out-of-bounds accesses [59]. Barrier instructions such as *lfence* are also used to prevent Spectre-PHT [13, 6]. On a number of AMD microarchitectures, *jmp2ret* replaces all returns with a direct jump to a return sanitized on privilege transitions, mitigating Retbleed [7]. On certain Intel microarchitectures, call-depth tracking is used to avoid BTB predictions for return instructions upon an RSB underflow, i.e. Retbleed [63].

Hardware mitigations. Both Intel and AMD support Indirect Branch Restricted Speculation (IBRS) which limits target predictions to a privilege level [25]. After a Model Specific Register (MSR) write, targets for indirect branches inserted while running in a privilege level (e.g. user mode) are prevented from being used as predicted branch targets in higher privilege levels (e.g. kernel mode). Enhanced IBRS (eIBRS) is supported by newer Intel microarchitectures, and does not require MSR writes on privilege transitions. Newer AMD microarchitectures also support such variant, called Automatic IBRS (AutoIBRS). Since it was introduced recently, only recent Linux kernel versions support its usage, and it is thus not widely deployed yet [41].

Likewise, Single Thread Indirect Branch Predictors (STIBP) prevents branch predictions from being used across sibling hardware threads [27]. Intel and AMD microprocessors also support Indirect Branch Prediction Barrier (IBPB), which stops indirect branch targets inserted prior to issuing the command from being used in the future. However, due to the prohibitive performance cost associated with IBPB, it has not been widely adopted in software.

To partially mitigate PhantomJmps, an MSR configuration *SuppressBPOnNonBr* can be enabled to ignore branch predictions for non-branch instructions on AMD Zen 2 [7].

Chapter 3

Phantom: Exploiting Decoder-detectable Mispredictions

Publication details

The results of this chapter are part of a paper that has been submitted for publication:

Wikner J., Trujillo, D. and Razavi, K. **Phantom: Exploiting Decoder-detectable Mispredictions**. 56th IEEE/ACM International Symposium on Microarchitecture (MICRO '23). [*Under submission*]

Contributions

Johannes Wikner and I share first authorship of this paper. A part of the work for the paper listed above was completed prior to the start of this thesis. Some of these results are public [57], while others are presented in Appendix A and Appendix B for completeness. Furthermore, observations **O1** and **O3**, and primtives **P1** and **P3** originate from results obtained prior to starting this thesis. However, since the work done during this thesis yields new insights regarding these observations and primitives, they are still included in this thesis.

For this Master thesis, my contributions in this chapter were the implementation of the covert channel measurements, the implementation of the physmap KASLR and physical address break, and the implementation of a PoC that leaks kernel memory using an MDS gadget.

3.1 Threat model

We consider a realistic attacker that is able to run unprivileged software on the victim machine. The goal of the attacker is to infer secrets using a transient execution attack. The victim machine is equipped with an AMD Zen microprocessor, and runs Linux kernel 5.19.0-28-generic. All state-of-the-art Spectre defenses are deployed, both in hardware and software. This includes retpoline [2, 26], call-depth tracking [63], jmp2ret, user pointer sanitization [50], KPTI [22], and disabling of unprivileged eBPF [37].

3.2 Overview

The majority of current literature on the topic of transient execution attacks focuses on misprediction of branches that were trained using a branch type that matches the victim branch type. Such mispredictions can usually not be detected by the decoder, and thus they yield long

speculation windows in which secret-dependent accesses can be made. The EX stage in the backend of the CPU pipeline will eventually issue a resteer.

Intuitively, speculative type conflicts between a branch prediction and the architectural branch may not occur, assuming the BPU is consulted after the ID stage has confirmed the existence of a branch instruction. Even if the BPU is consulted prior to decode, it would not yield a speculation window long enough for exploitation, since the decoder in the frontend of the CPU can already detect misprediction. However, recent work by Wikner and Razavi has proven this not to be the case, by showing that architectural return instructions may be speculatively confused with indirect branches [56]. Despite the fact that the prediction does not match the architectural branch type, long speculation windows are enabled in which data can be leaked. Likewise, prior to this thesis, we show that short speculation windows can arise when arbitrary instructions are mistaken for branch instructions, a phenomenon to which we referred as PhantomJmp [57]. AMD later released an advisory that refers to these cases as Branch Type Confusion [7].

In this chapter, we study decoder-detectable speculation in more detail. First, we explore previously untested asymmetric combinations of the training branch type and victim branch type. In addition, we determine to which extent AMD's mitigations protect against decoder-detectable misprediction. Lastly, we present two exploits based on Phantom speculation.

We classify all cases of decoder-detectable mispredictions as Phantom speculation, since we speculate using predicted branch instructions that do not match reality. As a first research goal, we wish to determine which asymmetric combinations trigger speculation, how far the speculation reaches in the pipeline, and which AMD CPUs are susceptible:

> Research Question **RQ1.**
>
> What asymmetric combinations of training- and victim instruction types trigger Phantom, how far do they reach in the pipeline, and on which AMD CPUs?

Since Phantom speculation results in a frontend resteer, we hypothesize that we are only able to observe very short speculation windows for most asymmetric combinations, in line with previous work [57, 7]. Therefore, we need methods that are capable of detecting very short speculation windows to accurately answer this research question.

In Section 3.3.1, we design three observation channels to detect short speculation windows. First, by triggering transient execution of code that issues a data load, we can determine whether we have reached the EX stage by observing the state of the data cache using Flush+Reload. If we observe that *transient execute* of the data load has happened, we deduce that the mispredicted control-flow has advanced through IF, ID and EX. Second, by examining the state of the µop-cache we can determine whether the instructions at the mispredicted target have reached the ID stage, i.e. whether *transient decode* has occurred. Lastly, if we observe using Flush+Reload that the code of the predicted branch target has been fetched into the instruction cache, we conclude that the decoder-detectable misprediction has reached the IF stage. That is, we can detect *transient fetch.* In Section 3.3.2 we use these observation channels to detect which asymmetric instruction type combinations trigger Phantom speculation.

Having established this, our second research question is:

> Research Question **RQ2.**
>
> What exploitation primitives can we build using Phantom speculation?

In Section 3.4 we describe three exploitation primitives that are enabled by Phantom mispredictions, that detect mapped memory (**P1** and **P2**) and leak register values (**P3**). One of our primitives, **P1**, is *execute-free*, i.e. it only requires *transient fetch.* In addition, we measure

the covert-channel characteristics of our primitives on all AMD Zen microarchitectures. Since our end goal is information disclosure, our last research question reads:

> Research Question **RQ3.**
>
> What information can we leak with these exploitation primitives?

In Section 3.5, we present a KASLR derandomization attack on AMD Zen 1 and Zen 2 microarchitectures that can determine the location of the kernel's direct mapping of physical memory, i.e. physmap. We also demonstrate how this enables an attacker to recover the virtual to physical address mapping of its own address space. Lastly, having broken KASLR, we evaluate the feasibility of using this information to leak arbitrary kernel memory using an MDS gadget as suggested by previous work [57, 7]. We show that leaking data is practical by developing a PoC that makes use of a realistic dummy MDS gadget, and we measure its bandwidth and error rate on AMD Zen 2.

3.3 PHANTOM speculation

In this section, we discuss methods that we use to determine whether a microarchitecture exhibits PHANTOM speculation. Specifically, we introduce a number of observation channels that allow us to determine precisely until what pipeline stage PHANTOM speculation has advanced, and discuss which combinations of asymmetric instruction types we use as training- and victim code.

3.3.1 Observation Channels

Figure 3.1 shows a high-level overview of our setup, which fits most experiments. In step ❶ we execute function A, which injects a BTB prediction by executing a branch to target C. Step ❷ executes function B, which contains code that does not match the instruction type used to train in A. If B can mispredict to C, regardless of having a non-matching instruction at B, the target microarchitecture exhibits PHANTOM speculation.

In order to cause misprediction, we allocate A and B such that they map to the same BTB entry, i.e. $i(A) = i(B)$, where i is the BTB indexing function. Prior work reverse engineered same-privilege BTB indexing functions that work on all Zen microarchitectures [56].

Figure 3.2 presents a high-level overview of which pipeline stages we distinguish between, and what observation channels we use to determine that PHANTOM speculation has advanced to a pipeline stage. For each observation channel, we prime the respective cache directly after

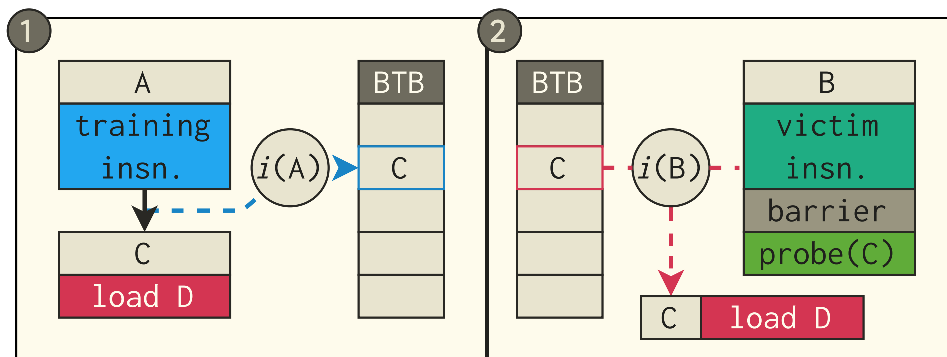


Figure 3.1: In step ❶, A injects C as a target in a BTB entry. In step ❷, the victim instruction of B may use the injected target as a prediction. C contains a data load of D.

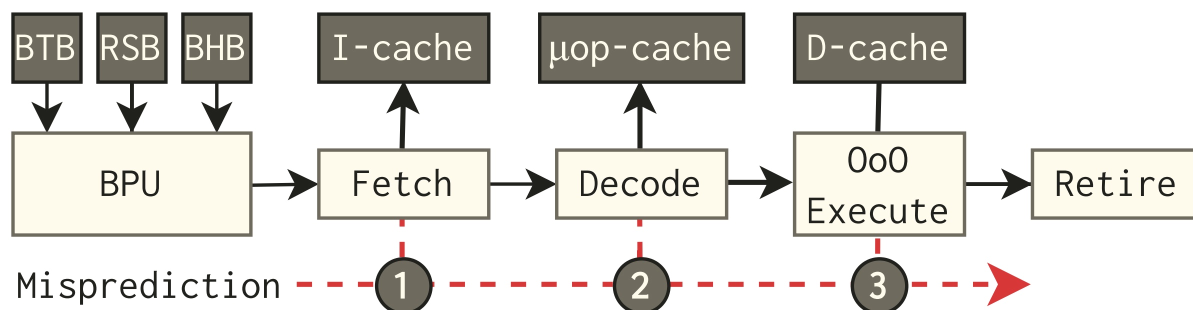


Figure 3.2: We can determine PHANTOM speculation at any of the pipeline stages by probing the: ❶ instruction cache, ❷ µop-Cache, and ❸ data cache.

step ❶, and probe the same cache during step ❷. We will now discuss each pipeline stage and its observation channel in detail.

Instruction Fetch (IF). We use the instruction cache as an observation channel for detecting IF due to PHANTOM speculation. For this, we use Flush+Reload on C. An example of how we detect IF due to PHANTOM speculation is shown in Figure 3.3-1. In step ❶, we execute an indirect jump to target C. We then flush C from the instruction cache. Lastly, we perform step ❷, which executes non-branch instructions, and finally, after a memory barrier, times the access time to C. If the access to C was fast, we conclude that IF happened because of PHANTOM speculation.

Instruction Decode (ID). Using the µop-cache as an observation channel, we determine whether PHANTOM speculation reaches the ID stage of the pipeline. However, to perform the probe- and prime step we require details of the µop-cache, such as its wayness and set indexing function. We therefore reverse engineer AMD's µop-cache by sampling performance counters. Specifically, we sample *de_dis_uops_from_decoder.opcache_dispatched* on Zen 2, and *op_cache_hit_miss.op_cache_hit* on Zen 3 and Zen 4. The results show that the µop-cache consists of 64 sets, each having 8 ways. The set is indexed using the lower 12 bits of the virtual address.

Figure 3.3-2 shows an example of how we detect ID due to PHANTOM speculation with an indirect jump. We perform step ❶, which trains the BTB by executing the indirect jump in function A, branching into C. This code location performs a number of jumps that all map to the same µop-cache. In step ❷, we execute jumps that map to the same µop-cache set as the jumps in C, and proceed to execute B. While executing B, we keep track of the µop-cache related performance counters. If B transiently executes C due to PHANTOM, we will observe µop-cache misses.

Execute (EX). To detect EX due to PHANTOM speculation, we rely on the memory access to address D, triggered by the code in C as shown in Figure 3.1. We are unaware of any mechanism in modern CPUs that are used to abort data accesses after they have been issued, and thus we assume the result of transiently performing of a data load to be visible in the data cache, even if misprediction was detected before the data was served back from the memory. We first execute A, which trains the BTB. Then, we flush D from the cache hierarchy. Lastly, we invoke function B and consequently determine the access time of D. We assume that PHANTOM speculation reached the EX stage if D was cached.

3.3.2 Triggering mispredictions

Having established the general experiment and our observation channels, we now proceed to determine which instruction types in A and B we will explore to trigger PHANTOM speculation.

We consider ① indirect branches (*jmp**), ② direct branches (*jmp*), ③ conditional branches

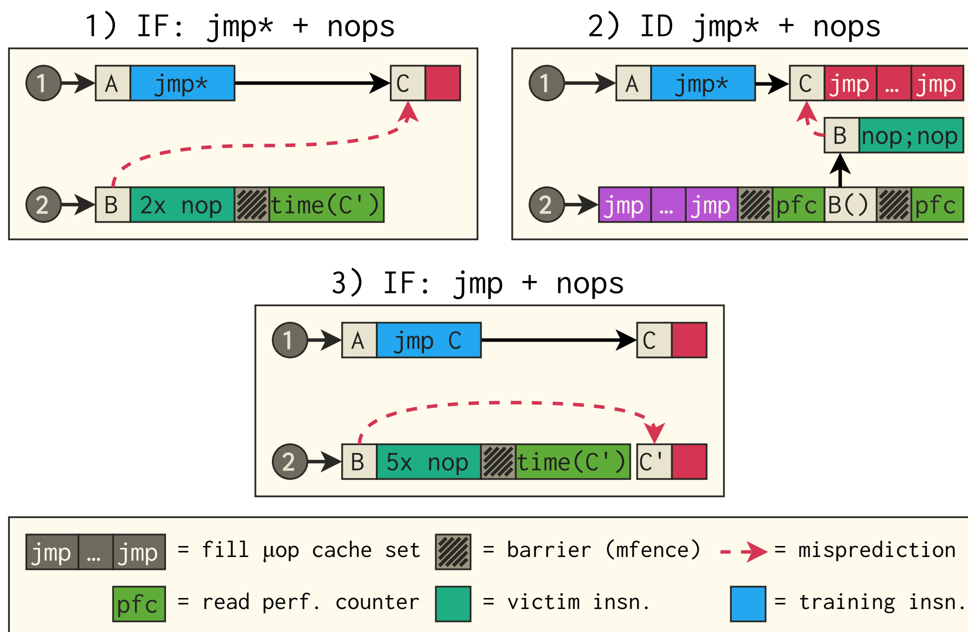


Figure 3.3: 1) Training *non branch* using an (indirect) *jmp**, measuring IF: C will be cached in the instruction cache. 2) Training using *non branch* using an (indirect) *jmp**, measuring ID: The *jmp*-series in C will evict the series in B, resulting in µop-cache misses. 3) Training *non branch* using a (direct) *jmp*, measuring IF: C' is at an address at the same relative offset from B as C from A.

(*jcc*) and non-branch instructions (*non branch*). We wish to explore all asymmetric combinations of these instruction types when it comes to the *predicted type*, i.e. the type of the training branch, and the *architectural type*, i.e. the type of the victim instruction. We will now discuss each instruction type and its possible semantics when used as a prediction.

Training with non branch. Executing *non branch* instructions at A invalidates any branch predictions for the associated BTB entry. Therefore, upon executing B, we expect the BPU to not provide any prediction. In response, the IF stage will continue to fetch instructions as if there was no branch. Previous work has discussed this phenomenon on AMD CPUs, and it is referred to as Straight-Line-Speculation (SLS) [8, 55].

Training with direct branches. To to best our knowledge, we are the first to explore transient execution attacks that train using a direct jump, i.e. *jmp*. Since a direct jump contains a branch target relative to the current instruction pointer, we expect to observe PHANTOM speculation at a matching offset from B. In other words, we expect the BTB to provide a branch target relative to the code location at which the branch is predicted. We therefore allocate C', a code location that has the same distance to B as C has to A, i.e. $|A - C| = |B - C'|$. Figure 3.3-3 presents this scenario.

Training with indirect branches. The original Spectre work [30] trained with indirect branches, i.e. *jmp**, but did not investigate cases where the architectural instruction does not match the predicted branch type. Whereas Retbleed [56] also trained with *jmp** for a non-matching instructions, they limited their exploration to architectural *ret* instructions only. In this chapter, we investigate other victim instruction types as well.

Training with returns. Return instructions are provided with a predicted target by the Return Stack Buffer (RSB). Therefore, we hypothesize that a predicted return instruction triggers PHANTOM speculation at the location of the address most-recently pushed to the RSB, instead of the location to which we returned during the training phase. We therefore prepare the RSB in a known state before executing B, by executing a *call* instruction.

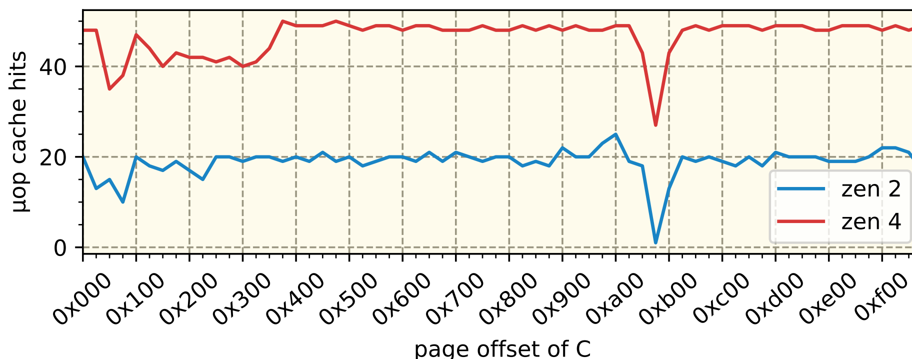


Figure 3.4: Detecting transient decode. Mispredictions caused by training *non branch* using *jmp** is observable in the µop-cache. Only when we place C at the page offset that matches the *jmp-series* in B (here *0xac0*), we see µop-cache misses.

3.4 Exploitation Primitives

Table 3.1 shows the results of our experiments. We draw a number of interesting observations from these results. First, for all tested combinations, fetch and decode of the predicted target occurs. This happens even in the absence of an architectural branch at that location (e.g., when the victim instructions are *nops*, used for the *non branch* case). We can thus conclude that the frontend fetches branch targets *before* it has even determined whether a branch exists. This leads to our first observation:

> Observation **O1.**
>
> On all Zen CPUs, speculative branch targets are fetched before the branch source is decoded.

Moreover, our results show that instructions at speculative branch targets are decoded as well, even in the absence of any branch source. As an example, Figure 3.4 shows the results of the ID observation channel, when a *nop* (i.e., *non branch*) is confused with a predicted *jmp**. Thus, our second observation is:

> Observation **O2.**
>
> On all Zen CPUs, the speculative branch target fetches we observe are not due to instruction prefetching, but with *intent to execute.*

Furthermore, on AMD Zen 1 and Zen 2, instructions at the target even reach the execute stage. On these microarchitectures, we measure a cache hit on the address loaded from memory by the instructions at the speculative target. Our third observation therefore is:

> Observation **O3.**
>
> On AMD Zen 1 and Zen 2, decoder-detectable speculations yield windows long enough to execute code.

Now that we have an overview of decoder-detectable speculation on AMD Zen CPUs, we will discuss the primitives that this enable for an attacker.

3.4.1 Attacker primitives

Understanding Phantom better, we can now use the observation channels for exploitation instead. An attacker can trigger Phantom speculation on arbitrary instructions, changing

Training	Victim instruction: *jmp**	*jmp*	*jcc*	*ret*	*non branch*
*jmp**	— [a]	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕ [b]	● ● ◕ ◕
jmp	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕
jcc	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕
ret	● ● ◕ ◕	● ● ◕ ◕	● ● ◕ ◕	— [a]	● ● ◕ ◕
non branch	● ● ◕ ◕ [c]	● ● ◕ ◕ [c]	● ● ◕ ◕ [c]	● ● ◕ ◕ [c]	— [a]

◔ : IF ◕ : ID ● : EX ◑ : Zen 1 ◑ : Zen 2 ◑ : Zen 3 ◑ : Zen 4

[a] Not Phantom [b] Retbleed [56]. [c] Spectre-SLS [55, 8].

Table 3.1: Various combination of training and victim instructions and how far they reach in the pipeline. The asymmetric combinations here, we refer to as Phantom speculation.

the state of the µop- and instruction cache. Likewise, on certain AMD microarchitectures, an attacker can trigger Phantom speculations that fit a memory load, even on non-branch instructions, changing the state of the data cache.

We identify the observation channels to give rise to three adversarial exploitation primitives. This section describes these primitives in detail. In Section 3.5, we discuss how we build exploits using these primitives.

P1: Detecting mapped executable memory. An instruction fetch only populates the instruction cache if the target of the fetch was executable and backed by physical memory. Combining this insight with Phantom, we can detect whether a virtual address T is mapped and executable in a victim's address space by using Prime+Probe on the instruction cache. The attacker would ① prime the instruction cache, ②, train the BTB with branches to T, ③ execute the victim and ④ infer whether T was fetched by probing the instruction cache.

P2: Detecting mapped non-executable memory. If our target T is mapped but not executable, the fetch fails and would leave the state of the instruction cache unaffected. Using Phantom on Zen 1 and Zen 2, however, an attacker can trigger a data load of target T instead. To detect mapped but non-executable memory, the victim's address space needs to contain a disclosure gadget G that loads the address in some register $\mathbf{R}$ from memory. Then, an attacker ① primes the data cache, ② trains the BTB with branches to G, ③ executes the victim with value T in register $\mathbf{R}$ and ④ infers whether T was loaded from main memory by probing the data cache.

P3: Leaking register values. Lastly, instead of detecting mapped memory, an attacker can use Phantom speculation windows to leak the victim's register values on AMD Zen 1 and Zen 2, using Prime+Probe. In this case, the disclosure gadget G would need to add the shifted value in the victim register to an address in a mapped area of the victim's address space, and issue a load on the resulting address. An attacker would ① prime the data cache, ② train the BTB with a branch to G, ③ execute the victim and ④ infer which address was loaded by probing the data cache.

Alternatively, the attacker can enable a more robust variant using Flush+Reload. First, they need to allocate a reload buffer RB which is shared memory with the victim, for example using the kernel's physmap area. The disclosure gadget G is chosen such that it issues a data load on the address obtained by adding the shifted victim register value to the address in some register $\mathbf{R}$. An attacker would ① flush the entire RB from the cache hierarchy, ② train the BTB with a branch to G, ③ execute the victim so that the address of RB ends up in register $\mathbf{R}$, and ④ infer which offset in the reload buffer was loaded from memory using Flush+Reload.

3.4.2 SuppressBPOnNonBr and AutoIBRS

AMD Zen CPUs support hardware mitigations that may impact the results of our PHANTOM experiments. First, as a response to PHANTOMJMPS [57], AMD disclosed a configuration of Zen 2 CPUs that should prevent speculation arising from non-branch instructions. By setting the SuppressBPOnNonBr bit in MSR `0xC00110E3`, PHANTOM speculations should be prevented. In addition, Zen 4 CPUs support AutoIBRS, which prevents branch predictions from being influenced across privilege levels. In this section, we discuss the implication of setting the SuppressBPOnNonBr bit and enabling AutoIBRS on our results.

SuppressBPOnNonBr. We repeat the experiments described in Section 3.3.1 and Section 3.3.2, but then with this bit enabled. As expected, our results show that whenever the victim instruction is of type *non branch*, we do not observe execution at the predicted target anymore. However, we find that this bit *does not* prevent IF when the victim instruction is of type *non branch*.

Observation **O4.**

SuppressBPOnNonBr does not prevent IF due to a PHANTOMJMP.

AutoIBRS. We again repeat the experiments in Section 3.3.1 and Section 3.3.2 on Zen 4. However, unlike before, we train in user space while we try to trigger PHANTOM speculation in kernel space. For this, we use the cross-privilege functions shown in Appendix A, which were found by us prior to the start of this thesis. Interestingly, our results show that IF is still triggered, despite AutoIBRS being enabled.

Observation **O5.**

AMD AutoIBRS does not prevent IF of cross privilege mode branch targets.

Our previously described primitive **P1** is thus unaffected on all AMD Zen microarchitectures. Primitives **P2** and **P3**, however, are now restricted to speculation on arbitrary branch instructions on AMD Zen 2, thanks to the SuppressBPOnNonBr mitigation. However, given that branches are frequently occurring in software, the impact of this mitigation is negligible. In addition, all primitives still work unrestricted on Zen 1.

3.4.3 Covert Channel

Our primitives **P1** and **P2** enable an attacker to detect mapped memory by triggering a fetch and, on some microarchitectures, a data load. In this section, we measure the accuracy and leakage rate of these primitives. In order to do this, we build a custom kernel module that performs a number of jumps. From user space, we aim to hijack one of these jumps to trigger a fetch or data load in the kernel to either area T_0 or area T_1. In the kernel address space, T_0 is not mapped while T_1 is mapped. Furthermore, address T is mapped, executable and contains a memory load instruction that fetches the address in a register **R**.

Fetch. We randomly generate 4096 bits. For each random bit b, we ① probe a chosen instruction cache set S, ② prime the BTB with a target in area T_b such that it falls in cache set S, ③ invoke

μarch	Model	Accuracy	Rate
Zen 1	AMD Ryzen 5 1600X	96.30%	204 bits/s
Zen 2	AMD EPYC 7252	93.04%	215 bits/s
Zen 3	Ryzen 5 5600G	100%	256 bits/s
Zen 4	Ryzen 7 7700X	90.67%	341 bits/s

Table 3.2: Accuracy and leakage rate of **P1** when leaking 4096 bits (median of 10 runs).

the kernel module and ④ probe instruction set S. If our probe step indicates a higher latency than probing the BTB with T_0, we deduce that we primed the BTB with a target in area T_1 and thus we output a 1. Likewise, if our probe step indicates a lower latency than probing the BTB with T_1, we output a 0. To improve performance, mostly on AMD Zen 3 and Zen 4, we prime the BTB so that the location where the speculative branch is triggered straddles a page boundary. On the co-resident hyperthread, we run `stress -c 10`. The results can be seen in Table 3.2.

Execute. Again, we randomly generate 4096 bits. For each random bit b, we ① we probe a chosen data cache set S, ② prime the BTB with target T, ③ invoke the kernel module such that an address in T_b that falls in set S ends up in $\mathbf{R}$ and ④ probe data cache set S. From our probe step we deduce whether we provided the kernel module with a target in area T_0 or in T_1. Table 3.3 shows our results.

μ**arch**	**Model**	**Accuracy**	**Rate**
Zen 1	AMD Ryzen 5 1600X	100%	256 bits/s
Zen 2	AMD EPYC 7252	99.28%	292 bits/s

Table 3.3: Accuracy and leakage rate of **P2** when leaking 4096 bits (median of 10 runs).

3.5 Exploitation

We evaluate two exploits using the primitives we presented in Section 3.4. First, we discuss how we use **P2** to leak the kernel's physmap location in Section 3.5.1. This attack uses Prime+Probe which turns out to be noisy. We explain how we can overcome this in Section 3.5.2. Finally, in Section 3.5.3 we evaluate an arbitrary kernel leak using a Flush+Reload attack. Appendix B presents our previous work on leaking the kernel code location using **P1**, completed prior to the start of this thesis.

3.5.1 Breaking physmap KASLR

Physmap is the direct mapping of physical memory in the address space of the kernel, and has, depending on configuration, 25600 possible locations [32]. As mentioned in Section 3.4.1, we can only detect mapped memory using a Phantom-induced transient fetch if the target is executable. However, physmap is an non-executable memory area. To derandomize physmap, we therefore use **P2**, which detects mapped non-executable memory by measuring a load triggered in the short speculation window using Prime+Probe on the L2 data cache. For constructing eviction sets more easily, we rely on transparent huge pages being enabled.

```
nop     DWORD PTR [rax+rax*1+0x0]
push    rbp
mov     esi,0x4000
mov     rbp,rsp
sub     rsp,0x8
call    0x9341c7b0
```

Listing 3.1: We trigger speculation at the call instruction, upon entering `__fdget_pos()`. Found at kernel offset 0x41db60.

```
mov     r12,QWORD PTR [r12+0xbe0]
```

Listing 3.2: Our disclosure gadget to leak the physmap location. Found at kernel offset 0x41da52.

μarch	Model	Accuracy	Median time
Zen 1	AMD Ryzen 5 1600X	100%	101 s
Zen 2	AMD EPYC 7252	90%	106.5 s

Table 3.4: Accuracy and median time needed to find physmap on a AMD Zen 2 microarchitecture using **P2**, over 10 runs.

μarch	Model	Memory	Accuracy	Median time
Zen	AMD Ryzen 5 1600X	8 GB	99%	1 s
Zen 2	AMD EPYC 7252	64 GB	100%	16 s

Table 3.5: Accuracy and median time needed to find a physical address on a AMD Zen 1/2 microarchitectures, over 100 runs.

Using tooling made by previous work [56], we find that upon executing the `readv()` system call, we control the value of **R12** using the second argument to the system call (i.e. **RSI**) when `__fdget_pos()` is called. We trigger speculation by confusing the call instruction shown in Listing 3.1 with an indirect jmp. We train the BTB with a disclosure gadget as a target, shown in Listing 3.2.

Results. We run our physmap derandomization exploit 10 times on our vulnerable AMD machines, each time rebooting the system. Table 3.4 shows the success rate and median time needed to derandomize the physmap location.

3.5.2 Overcoming noise

Prime+Probe proves to be very noisy. This may be because of the cache replacement policy or because the system call trashes the chosen cache set. To improve the results, we repeat our exploit for multiple cache sets. For each set we also measure the latency of the probe step when triggering a transient load to an address falling in a different cache set, giving us a baseline B_s for the monitored set S. We score each possible physmap guess using the bounded probe timing difference between transiently loading an address in the guessed physmap area falling into the primed cache set (i.e. P_s) and the baseline B_s, accumulated for all 1024 L2 cache sets. That is, $score_{guess} = \sum_{S=0}^{S \leq 1024} MIN(MAX(P_S - B_S, -1), 1)$.

3.5.3 Leaking kernel memory

We now discuss how our **P3** extends the attack surface of Spectre with new gadgets. First, to leak kernel memory, we need to find the location of a reload buffer in physmap.

Enabling Flush+Reload. We use the attacks in Appendix B and Section 3.5.1 to leak the kernel image and physmap locations respectively. To enable Flush+Reload, however, we need to be able to determine physical addresses belonging to our user space program. In particular, we must know the physical address of a reload buffer to enable a Flush+Reload type of attack through the kernel's physmap area, as discussed in Section 3.4.1. To determine physical addresses allocated to our program, we re-use the same setup as described in Section 3.5.1, i.e. we trigger speculation during the `readv()` system call. For an address A in our user space program, we make a guess P_g of its physical address, and we pass physmap + P_g in **RSI** upon calling the system call. If our guess is correct, we can detect this using Flush+Reload on address A. To reduce entropy, we allocate A as a 2 MB huge page.

We attempt to determine the physical address of A a 100 times. To randomize the physical address of A, we allocate N huge pages before allocating A, where N is randomly sampled each run such that $0 \leq N \leq 99$. Table 3.5 presents the accuracy and median time observed.

Leaking memory with MDS gadgets. In this section, we build a PoC that makes use of reduced Spectre gadgets, referred to as MDS gadgets in previous literature [28], to leak arbitrary kernel memory by combining them with **P3**. The high-level idea has been described by us in prior work [57], and was later independently reported by AMD [7].

A conventional disclosure gadget performs two loads: one that fetches the secret from memory and one which uses the result of this load to access a slot in a reload buffer. With **P3**, however, we are able to introduce the secret-dependent load ourselves. A gadget that only performs one out-of-bound load would thus be enough to enable arbitrary read capabilities.

```
void read_data(uint64_t user_index) {
    if (user_index < *array_length) {
        uint8_t data = array[user_index]
        parse_data(data);
    }
}
```

Listing 3.3: A sample MDS gadget.

To prove that such an exploit is practical, we build a kernel module that contains a realistic MDS gadget. Listing 3.3 shows a high-level code explanation of the gadget that we use. When the user provides an out-of-bounds value to `read_data()`, the conditional branch may be incorrectly predicted as taken, causing an attacker-controlled address to be fetched from memory. A conventional Spectre attack would not succeed, however, since there is no data load that depends on the value of `data`. Our goal is to introduce this secret-dependent load using **P3**.

We presume to know where our MDS gadget resides in the kernel address space. We also know the location of the kernels' physmap area and the physical address of our reload buffer. Furthermore, we have the address of a disclosure gadget in the kernel that performs the secret-dependent load. All this information can be leaked with our previous steps. The user provides the kernel module with `user_index` and the location of our reload buffer in the kernel's virtual address space.

Relying on BTB aliasing, we train the conditional branch to be predicted as taken. Additionally, we train the BTB to believe there exists a branch to the disclosure gadget at the location of the (direct) call to `parse_data()`. Our disclosure gadget indexes into our reload buffer using the (shifted) value of `data`.

Results. We run our proof-of-concept on an AMD Zen 2 EPYC 7252. Our results show that we can reliably leak 4096 bytes of randomized data from the kernel using an MDS gadget. We repeat our experiment 10 times, each time after a reboot. In 8 of these attempts, we measure a median bandwidth of 84 bytes/s, achieving a perfect accuracy of 100%. In the remaining 2 attempts, no signal is observed. One possible explanation could be an undesired BTB aliasing.

Finding MDS gadgets. This work focuses on the analysis of frontend speculation and not on the discovery of gadgets. Previous work shows how one can find MDS-like gadgets in the kernel [28]. Furthermore, new gadgets are continuously discovered and patched as shown recently by Google [64].

3.6 Mitigation

In this section we review mitigations that can be used against PHANTOM attacks, including some that were proposed by AMD [7].

3.6.1 Hardware mitigations

SuppressBPOnNonBranch. In response to PHANTOMJMPS [57], AMD revealed a previously undocumented MSR configuration bit to which they refer as SuppressBPOnNonBr. Upon setting this bit, it should not be possible to have non-branch instructions be mispredicted as a branch. Effectively, the CPU becomes less aggressive when it comes to branch predictions. With our results, however, we show that SuppressBPOnNonBr has only limited impact. First, as discussed in Section 3.4.2, the configuration does not prevent *transient fetch* and *transient decode*. Second, branch instructions, which occur frequently in almost all software, are still susceptible to decoder-detectable misprediction. Additionally, the configuration is only supported on Zen 2, leaving older models vulnerable to unrestricted PHANTOM speculation. In conclusion, the impact of this mitigation is negligible.

AutoIBRS. AutoIBRS is only supported by Zen 4, and does not prevent *transient fetch*, as shown in Section 3.4.2. We suspect that AutoIBRS operates too deep in the pipeline to prevent decoder-detectable misprediction. This hypothesis is in line with AMD's advisory [7], which states that IBRS prevents speculation for architectural indirect branches, suggesting it operates after the instruction has been decoded. An improvement to AutoIBRS on Zen 4 that would prevent *transient fetch* could be one that refuses to serve any BTB prediction to the IF stage that does not match the current privilege level. This requires AutoIBRS to perform checks in the very beginning of the pipeline.

Stall till decode. An hardware mitigation that tackles the root cause of PHANTOM attacks would be to stall until decode has finished. This would prevent any decoder-detectable misprediction. However, this approach has two issues. First, it would make Zen CPUs less performant, as the IF stage has to sit idle until its most-recently fetched instructions are decoded. Second, it would likely require an deep overhaul of AMD's CPU design, making it unlikely to be implemented in practice.

3.6.2 Software mitigations

lfence. In their advisory, AMD refers to previous recommendations of including *lfence* barriers after conditional branches, to also avoid arbitrary memory leakage using PHANTOM. This mitigation would only stop the arbitrary kernel leak, evaluated in Section 3.5.3, but would not prevent attacks that leak address space information, such as the one presented in Section 3.5.1. In addition, finding all vulnerable conditional branches is nontrivial [23, 38, 28, 53, 64]. Simply patching all conditional branches would impose an intolerable performance penalty [53]. Using *lfence* is therefore not a complete solution against PHANTOM attacks.

Address Space Isolation (ASI). Attempting to prevent transient execution attacks from leaking secrets, Address Space Isolation (ISA) unmaps kernel address space when possible, reducing secrets [47]. While the arbitrary kernel leak using an MDS gadget would be affected by this mitigation, it does not impact our attacks that detect mapped memory. Furthermore, ASI is only partially effective, as the entire address space still needs to be mapped in for certain interactions.

IBPB. Indirect Branch Prediction Barrier (IBPB) is a command that prevents predictions for indirect branches to be used that were inserted prior to issuing the command. When issuing IBPB upon a privilege switch, BTB targets inserted by user space processes are invalidated. In

addition, IBPB is supported on all AMD Zen CPUs, and it would thus be a suitable candidate to mitigate Phantom attacks. However, IBPB on every privilege switch comes with an substantial performance penalty, which we will quantify later in this thesis, in Section 4.7.2.

Chapter 4

Inception: Exposing New Attack Surfaces with Training in Transient Execution

Publication details

The results of this chapter are part of a paper that has been submitted to a conference for publication:

Trujillo D., Wikner J. and Razavi, K., 2023. **Inception: Exposing New Attack Surfaces with Training in Transient Execution**. 32nd USENIX Security Symposium (USENIX Security '23). [*Accepted*]

Contributions

Johannes Wikner and I share first authorship of this paper. My contributions to this paper is the discovery of PhantomCall, the design of Inception (e.g. recursive PhantomCall in a PhantomJmp, dueling PhantomCalls) and the implementation of Inception. I have also designed and implemented the TTE experiments concerning transient training of the RSB, i.e. TTE_{RSB}.

4.1 Threat Model

We assume a realistic scenario in which an attacker aims to leak secret data from a victim machine. The victim machine runs Linux 5.19.0-28-generic with all recent mitigations deployed, such as retpoline [2, 26], call-depth tracking [63], jmp2ret and SuppressBPOnNonBr [7], user pointer sanitization [50], KPTI [22], and disabling of unprivileged eBPF [37]. To evaluate TTE, we consider both Intel and AMD microarchitectures. For our end-to-end exploit Inception, we assume the machine to be equipped with an AMD Zen microprocessor.

4.2 Overview

In this chapter, we wish to investigate whether there exists attack surface of transient execution attacks that does not rely on gadgets which are actively patched, unlike Phantom as discussed in Chapter 3. We focus on a class of transient execution attacks that do their **T**raining in **T**ransient **E**xecution (TTE).

When switching privilege modes, the RSB and BTB are sanitized to prevent known transient execution attacks [7, 26, 12]. Our first challenge is to understand whether it is possible for an attacker to retrain these microarchitectural buffers during transient execution of the victim, and if so, under which circumstances. That is, determine the attack surface of TTE:

Challenge (C1). Understanding the attack surface of TTE and its requirements for an attack.

We address this challenge in Section 4.3 by discussing scenarios in which TTE of the BTB and RSB could occur. These scenarios require the presence of certain gadgets in the victim code, to which we refer as *TTE gadgets*. Our experiments reveal TTE of the RSB to be effective on AMD microarchitectures, and in Section 4.6 we present a more in-depth analysis of TTE, showing that TTE of the BTB works on both Intel and AMD microarchitectures. However, TTE gadgets are not necessarily trivial to find. We therefore question whether we can somehow relax the constraints of such gadgets.

As investigated extensively in Chapter 3, AMD CPUs are susceptible to speculation arising from arbitrary instructions, a class of transient execution attacks to which we refer as PHANTOM. Since PHANTOM allows an attacker to trigger transient execution windows with few, if any, requirements on the victim code, we wish to understand the interplay between PHANTOM and TTE:

Challenge (C2). Understanding the possibilities of combining TTE with PHANTOM speculation.

Our reverse engineering efforts, described in Section 4.4, reveal a TTE primitive that we refer to as PHANTOMCALL. A PHANTOMCALL is a predicted call instruction for an arbitrary instruction, and we discover that such prediction transiently trains the RSB, *before the decoder realizes there is no architectural call instruction*. Interestingly, we discover PHANTOMCALL to be effective on all AMD Zen CPUs, *even on Zen 3 and Zen 4*. Counterintuitively, PHANTOMCALL is not stopped by AMD's mitigation against PHANTOMJMPS on Zen 2.

Having obtained a TTE primitive using PHANTOM that allows manipulation of the RSB with barely any requirements, our last challenge is:

Challenge (C3). Enabling practical exploitation with PHANTOMCALL.

In Section 4.5 we introduce our end-to-end exploit INCEPTION, which achieves TTE of the RSB using PHANTOMCALL. INCEPTION leaks arbitrary kernel memory on fully-patched AMD systems. Exploitation using a single PHANTOMCALL, however, introduces challenges that are hard to overcome. Instead, we realize that we can trigger an infinite transient hardware loop that trains the RSB multiple iterations by establishing a recursive PHANTOMCALL.

Section 4.5 discusses how INCEPTION breaks KASLR and consequently leaks data from the kernel. We also measure its bandwidth and error rate on Zen 2 and Zen 4. Furthermore, we reveal that AMD's brand-new feature AutoIBRS, which aims to prevent cross-privilege transient execution attacks, *fails* to stop INCEPTION.

4.3 Training in Transient Execution

In order to trigger TTE, an attacker requires a *TTE gadget* to be present in the victim code. The attacker needs to be capable of setting the BPU state such that the TTE gadget is transiently executed during victim execution. Triggering this transient window does not directly lead to (arbitrary) data leakage, for example because the window is not long enough or because there is no (unmitigated) *disclosure gadget* along the transient execution path of the TTE gadget.

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

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

Instead, the attacker relies on a branch in the TTE gadget to transiently insert a prediction to a disclosure gadget during the transient window. This prediction triggers transient execution of the disclosure gadget afterwards, while (architecturally) executing a branch for which this prediction is served, ultimately leading to information disclosure. TTE thus escalates a transient window previously considered harmless to a powerful one that is capable of leaking secrets. The TTE gadget may, in addition to the training branch, contain instructions that causes faults, given that we execute it transiently.

Compared to training the BPU architecturally during victim execution, TTE significantly increases the attack surface. With architectural training, the victim code would need to execute a branch to a disclosure gadget during regular program execution such that this branch is mispredicted upon its future execution while the victim has sufficient control over registers or memory locations. Although theoretically possible, such scenario is unlikely to occur.

We now discuss the general method of accomplishing TTE. In Section 4.6 we provide a more in-depth analysis of the possible TTE variants. We use $\text{TTE}_{\text{A-B}}$ to refer to using a transient execution triggered by method A to train the microarchitectural buffer B. More generally, we use TTE_{B} to refer to transient training of microarchitectural buffer B.

4.3.1 Training BTB in transient execution

Listing 1 presents an example of a TTE gadget, potentially vulnerable to $\text{TTE}_{\text{PHT-BTB}}$. If the attacker can trigger the conditional branch to be predicted as taken, the indirect branch to b is executed transiently. Our hypothesis is that transient execution of this branch trains the BPU, even though it never retires. If b is under control of an attacker during this transient window, they would thus be able to insert a branch prediction to an arbitrary address in the BTB, such as one containing a disclosure gadget. When (architecturally) executing this TTE gadget afterwards, the branch triggers transient execution of the disclosure gadget, enabling information leakage. The register previously containing the address of our disclosure gadget (b) is now available for other purposes, such as providing values used by disclosure gadget.

However, controlling b would imply that the attacker is able to execute the disclosure gadget while transiently training the BTB. Thus, we already reach the disclosure gadget while executing the TTE gadget, effectively eliminating the need for TTE. In addition, some of the current mitigations affect the presence of these TTE gadgets. For example, retpolines replace indirect branches with return instructions to mitigate Spectre-BTB [51]. In spite of this, TTE_{BTB} increases the attack surface of transient execution attacks in some cases of interest, and Section 4.6 discusses these in detail.

```
1  void TTE_pht_rsb (state_t *a) {
2      if (*a) {          /* mispredicted as true */
3          f();            /* pushes the return target to RSB */
4          DISCLOSURE_GADGET; /* top of the RSB would point here */
5      }
6      return; /* the return speculates to LEAK_GADGET */
7  }
```

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

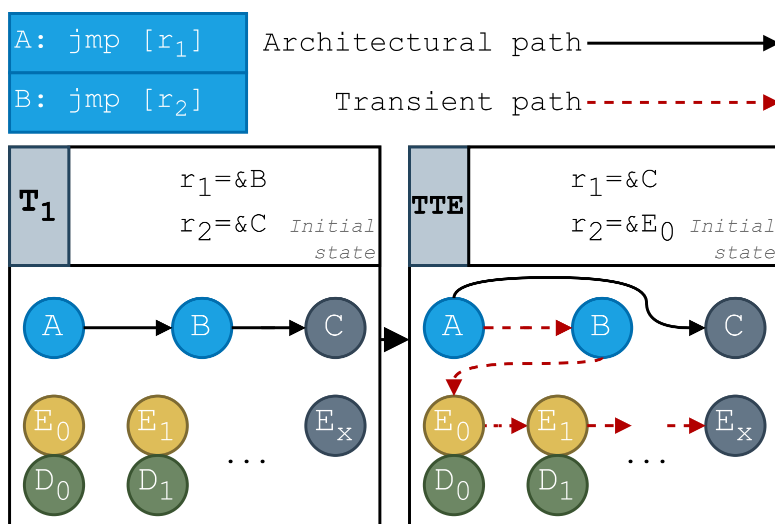


Figure 4.1: Injecting RSB entries in the transient execution window of an indirect branch ($\text{TTE}_{\text{BTB-RSB}}$). In T_1, A is trained to execute B. Next, in *TTE*, A transiently executes B, which in turn executes a series of call instructions E_i, each followed by a disclosure gadget D_i.

4.3.2 Training RSB in transient execution

Listing 2 demonstrates a TTE gadget that may update the RSB with a transiently executed call instruction, triggered by a mispredicted conditional branch, i.e. $\text{TTE}_{\text{PHT-RSB}}$. We call such TTE gadget a *call-and-disclose gadget*, since it consists of a call instruction which is immediately followed by a disclosure gadget. If this transiently executed call would update the RSB, the address of the disclosure gadget would be pushed onto the RSB.

Executing the example in Listing 2 yields some interesting results: the transiently executed call updates the RSB on certain microarchitectures but not reliably, which we can check by determining whether any of a number of return instructions executed afterwards trigger transient execution of the disclosure gadget. Hence, we construct a more thorough experiment, this time using $\text{TTE}_{\text{BTB-RSB}}$. In Section 4.6 we adapt this experiment for $\text{TTE}_{\text{PHT-RSB}}$ and $\text{TTE}_{\text{RSB-RSB}}$.

Experiment setup. Figure 4.1 illustrates how we verify RSB training using a mispredicted indirect branch with two procedures, T_1 and TTE. The goal of the experiment is to determine whether transient execution of a call instruction manipulates the state of the RSB.

For any i, E_i and D_i together form the call-and-disclose gadget. Green nodes (D_i) represent the disclosure gadgets, whose addresses we anticipate to inject into the RSB using TTE. Each D_i issues a memory load to a reload buffer *RB*, leaving a distinct observable trace in the data cache to indicate its execution. Yellow nodes (E_i, $0 \leq \text{i} \leq \text{X}$) represent the call instructions. Each call E_i is followed immediately by its respective disclosure gadget D_i.

E_i calls the next 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 barriers that stop speculation using a memory barrier instruction, specifically using *mfence*. We run our experiment for $0 \leq \text{X} \leq 50$, to be able to compare the results of executing different numbers of calls transiently.

Before triggering the TTE procedure, we need to establish a known state of the RSB in order to determine whether it has been transiently manipulated. Therefore, each experiment starts with priming the RSB by issuing a number of calls, as shown in Listing 3, where N is the size of the RSB on the target microarchitecture. Each call is again followed by a distinguishable memory load to our reload buffer *RB*. Upon executing these call instructions architecturally, the addresses of these memory loads will be pushed onto the RSB. We further flush the buffer *RB* from the cache hierarchy before starting the experiment. This allows us to later infer the state of the RSB, revealing whether a transiently executing call E_i has changed the RSB state.

Microarchitecture	$\mathrm{TTE}_{\text{BTB-RSB}}$
Zen 1	✓
Zen 1+	✓
Zen 2	✓
Zen 3	✓
Zen 4	✓
Coffee Lake	–
Coffee Lake R	–
Ice Lake	–
Comet Lake	–
Rocket Lake	–
Golden Cove	–
Gracemont	–
Raptor Cove	–

Table 4.1: CPUs that are vulnerable speculative training of the RSB with $\mathrm{TTE}_{\text{BTB-RSB}}$. A checkmark indicates that we do not always return back to our primed return sites.

Furthermore, to avoid cross-interference of experiments, we flush the branch predictor state using the IBPB command before each round of the experiment.

We first execute T_1 which is a preparatory procedure that primes the BTB with a state that ensures transient execution of B in the TTE step. Next, we execute TTE which triggers a series of calls to be transiently executed, potentially manipulating the RSB. After performing the experiment, we want to examine the state of the RSB. To do this, we issue a number of returns, as shown in Listing 4. In the case that the RSB is not manipulated under transient execution, we expect to transiently execute the return sites primed in Listing 3, which we can identify using the distinguishable memory accesses they perform. However, if the RSB indeed was manipulated, we expect one or more return instructions to not trigger the memory load issued by the primed return site. Even stronger would be if we can observe memory accesses caused by transient execution of our disclosure gadgets (D_i).

Results. The results can be seen in Table 4.1 and reveal that manipulating the RSB transiently is feasible on all considered AMD microarchitectures. This is in line with their Software Optimization Guide, which states that incorrect pushes and/or pops to the return address stack may occur during speculative execution [4, 5]. Generally, we observe that a transiently executed call evicts an entry at the *bottom of the RSB*, i.e. the oldest RSB entry. That means that, in the case of a single transiently executed call, the last return executed (the 31st return since AMD's RSB effectively holds 31 entries) will not predict back to our primed return site. Likewise, executing two transient calls generally evicts two entries at the bottom of the RSB, causing the last two returns executed (return #30 and #31) to not predict back to our primed return sites.

```
1    .id=0
2    .rept N
3        call 1f
4        *(RB + (id * 4096))      ; leaving a distinguishable trace in cache
5        1: pop %r8               ; restore stack state
6    .id=.id+1
7    .endr
```

Listing 3: Priming the RSB with distinct return sites, whose execution can be measured using a cache side-channel. N is set to size of the RSB.

```
.rept N
    push 1f
    clflush %rsp        ; ensuring a large transient window by flushing the stack pointer
    ret
    1:
.endr
```

Listing 4: Dumping the RSB contents by issuing N returns.

We hypothesize that we manipulate the bottom entries of the RSB because AMD microarchitectures implement two RSB pointers for a circular buffer: a committed- and a speculative one. When misprediction is detected, the speculative pointer is restored to the committed one, which effectively puts the transiently injected entries at the bottom of the buffer. Figure 4.2 depicts this behavior using an RSB of size 8.

On Zen 1(+) and Zen 2, our results show that we generally do not execute the disclosure gadgets D_i when transiently executing less than 31 calls (the size of the RSB), i.e. when $X < 31$. Instead, the last RSB entry that is still intact after transiently executing our calls will be recycled for return prediction. For example, in step ④ of Figure 4.2, RSB entry 1 would be used for prediction upon execution of a return, instead of the transiently overwritten entry 8. Interestingly, we observe that overwriting all RSB entries *does* consistently cause our return instructions to use the transiently injected disclosure gadgets. In other words, we observe that the disclosure gadgets D_i are actually executed, instead of the return sites primed in Listing 3.

Conversely, on Zen 3 and Zen 4, we find that for many values X, most of the overwritten RSB entries are used as soon as we execute the corresponding return instruction (i.e., return #31 for the first transiently injected call), even if $X < 31$.

We hypothesize this behavior to be caused by the specific implementation of the RSB, which may be more complex than the model depicted in Figure 4.2. Furthermore, this implementation is likely different between the tested microarchitectures.

Observation (O1). We can hijack return instructions on AMD microarchitectures by transiently overwriting RSB entries using $TTE_{BTB\text{-}RSB}$. On Zen 1(+) and Zen 2 we need to overwrite all RSB entries to reliably control the predicted target of a return.

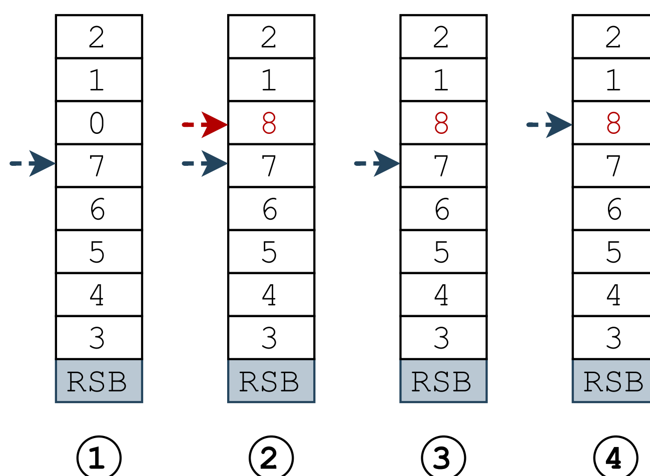


Figure 4.2: An implementation of a circular RSB with an committed top-of-the-stack pointer (shown in blue) and a speculative counterpart (shown in red). RSB entry numbers indicate when they were inserted (0 first, 8 last), and those depicted in red were inserted transiently. First, ① shows the state of the RSB 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 return instructions.

On Intel microarchitectures, we were not able to poison any RSB entries, on any of the tested CPUs. Although we cannot claim to know the cause of this, we note that patents assigned to Intel describe a speculative RSB [29, 18], whose implementation would result in the behavior that we observe.

In Section 4.6 we show that other transient execution windows also allow for TTE_{RSB} on AMD microarchitectures. However, a call-and-disclose gadget reachable in a transient execution window as shown in Listing 2 may be difficult to find in victim code. We therefore question whether we can relax this constraint by abusing other properties of AMD's microarchitectures.

4.4 TTE and PHANTOM

In Chapter 3 we presented a in-depth analysis of speculation arising from arbitrary instructions, or PHANTOM speculation. We realize that performing TTE with PHANTOM would reduce the constraints of a TTE gadget, given that it does not need to be reachable by a mispredicted architectural branch. To trigger TTE, however, the training branch in the TTE gadget needs to be transiently executed. According to our results in Chapter 3, transient execution due to PHANTOM is only possible on AMD Zen 1(+) and Zen 2 CPUs. Furthermore, mitigations against PHANTOM are available on Zen 2.

4.4.1 Chicken out from PHANTOM

To mitigate PHANTOM, AMD published an advisory in which they recommend to enable SuppressBPOnNonBr, an MSR configuration that mitigates PHANTOM speculation on non-branch instructions. This so-called *chicken bit* is currently enabled on Linux by default. The mitigation reduces the attack surface of PHANTOM, but is only available on Zen 2 microarchitectures. In addition, arbitrary branches remain susceptible to PHANTOM, despite the mitigation.

4.4.2 Exploring the limits of PHANTOM

In Section 4.3.2, we discussed $\text{TTE}_{\text{BTB-RSB}}$, and we provided an example of a code snippet that accomplishes this in Listing 2, using a call-and-disclose gadget. The example skews the direction of the conditional branch to transiently execute the call instruction, performing TTE_{RSB}.

Our goal is to reduce the constraints of such gadget by triggering transient execution of the call-and-disclose gadget using PHANTOM instead. If it is possible to perform TTE_{RSB} using PHANTOM, any arbitrary branch can be mispredicted to trigger transient execution of the call-and-disclose gadget on Zen 1(+) and Zen 2. This eliminates the need for the call-and-disclose gadget to be reachable from a mispredicted conditional branch. On Zen 1(+), we could even target non-branch instructions.

TTE_{RSB} using PHANTOM. To determine whether TTE_{RSB} using PHANTOM is feasible, we set up an experiment. Since we are interested in relaxing the constraints of performing TTE_{RSB} as much as we can, we focus on PHANTOM speculation on arbitrary non-branch instructions. If TTE_{RSB} if feasible using PHANTOM, we expect this to work on Zen 1(+) only, since Zen 2 is protected by the *chicken bit*, and Zen 3 and Zen 4 do not exhibit transient execution due to PHANTOM, as shown in Chapter 3.

Using BTB indexing functions found by previous work [56], we allocate two addresses A and B that map to the same BTB entry. On A, we execute an indirect branch to an address containing a call instruction. We then prime the RSB as done before, and as shown in Listing 3. On B, we execute non-branch instructions, specifically *nops*. Lastly, we dump the RSB contents as done previously by executing the code shown in Listing 4.

If TTE_{RSB} using PHANTOM succeeds, we expect to see that the last return executed during our dump step does not trigger transient execution of the call return site as primed.

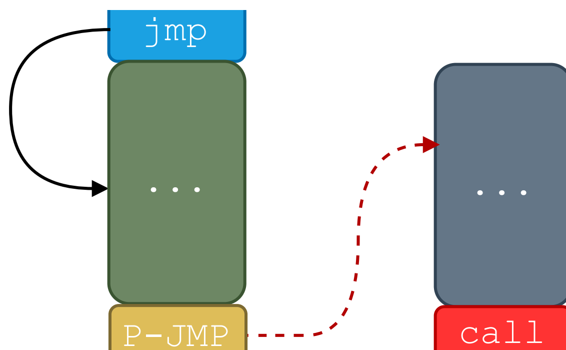


Figure 4.3: Triggering TTE_{RSB} inside a PHANTOM speculation window on Zen 3 and Zen 4. The colored boxes identify different cache lines. Architectural jumps are indicated by solid black arrows, while speculative jumps are indicated by dashed red arrows.

Results. Our experiments reveal unexpected results. First, as hypothesized, we are able to perform TTE_{RSB} on Zen 1(+), proven by the fact that the last return in our dump step does not transiently execute the primed return site. However, our results show that *we even corrupt an RSB entry on Zen 2*, despite the mitigation deployed against PHANTOM on non-branch instructions. We verify that transient execution of a data load is blocked, whereas it executes as soon as we disable the mitigation.

Even more surprising, the results reveal that TTE_{RSB} using PHANTOM is *effective on Zen 3 and Zen 4 as well.* Chapter 3 showed that transient execution due to PHANTOM is not possible under any circumstances on Zen 3 and Zen 4. This is in line with AMD's advisory, which states that BTC is limited to Zen 1(+) and Zen 2 only [7]. Despite all of this, we show that the call instruction, transiently executed using PHANTOM, has an observable effect on the RSB state.

Observation (O2). We can perform TTE_{RSB} using PHANTOM on all AMD Zen CPUs. This allows us to reach the call-and-disclose gadget from any arbitrary instruction.

Unlike on Zen 1(+) and Zen 2, however, we notice that TTE_{RSB} using PHANTOM on Zen 3 and Zen 4 is only effective under certain circumstances. Further experimentation shows that TTE_{RSB} using PHANTOM on Zen 3 and Zen 4 requires both the PHANTOMJMP and the call to be at specific locations relative to the address using which the BTB is consulted. Specifically, the PHANTOMJMP must appear in the cache line following the one in which an address falls that is used to consult the BTB. As an example, if the PHANTOMJMP is preceded by an architectural branch to byte 50 of a 64-byte cache line, we use that address to consult the BTB, and the PHANTOMJMP must not appear earlier than 14 bytes further. This is also the case for the call instruction: it must appear in the cache line following the one containing the address using which the BTB is consulted due to the PHANTOMJMP. That is, if the PHANTOMJMP branches to byte 40 of a cache line, the call must be preceded by at least 24 bytes of other instructions. Figure 4.3 shows an example, where colored boxes indicate cache line mappings. In Section 4.4.3 we discuss a possible explanation for this behavior.

Observation (O3). On Zen 3 and Zen 4, TTE_{RSB} using PHANTOM speculation only succeeds when the PHANTOMJMP and the call are located in the cache lines following the ones in which the addresses fall that are used to consult the BTB.

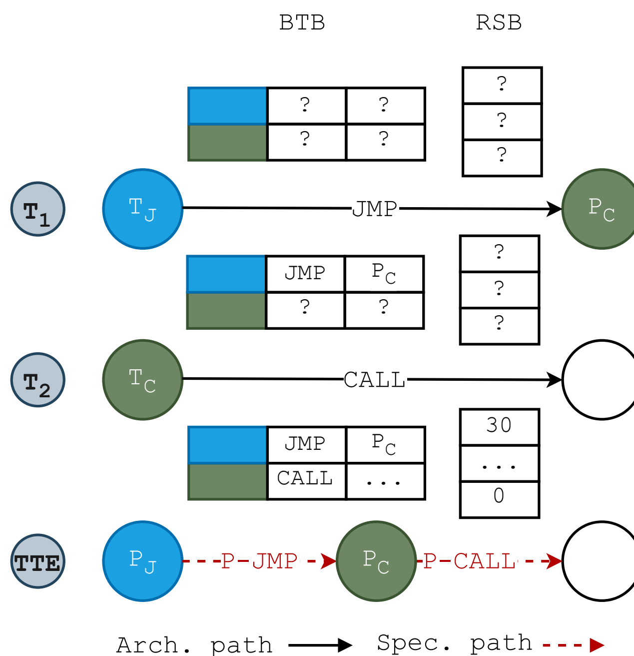


Figure 4.4: The experiment setup to test the feasibility of executing a PhantomCall in a PhantomJmp. Colors indicate BTB entry mapping.

4.4.3 PhantomCall

One hypothesis that we consider is that TTE of the RSB using Phantom works because of a *call prediction* on the target location of the PhantomJmp, and not because of the architectural call at the target location. We design an experiment to test our hypothesis, as shown in Figure Figure 4.4, with the state of the BTB and RSB shown after each step.

TTE_{RSB} using a call prediction only. In step T_1, we first execute a branch from T_J to P_C. T and P stand for Training and Phantom respectively, whereas J and C indicate the type of branch, i.e. a jump or a call. This step creates a BTB entry for a branch, with its target set to P_C, at which *nop* instructions reside. In step T_2, we execute a call at T_C, inserting a BTB entry for a call, of which the target is irrelevant. After performing steps T_1 and T_2, we 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 P_J, which maps to the same BTB entry as T_J. Thanks to step T_1, a branch prediction exists with P_C as the predicted target. We thus expect the CPU to select P_C as our new (transient) instruction pointer. As there also exists a BTB call-prediction for P_C thanks to step T_2, we expect the CPU to transiently push an entry onto the RSB. Lastly, we flush our reload buffer and execute return instructions as shown in Listing 4. We reload our memory pointers to determine which of the RSB entries are still intact. On Zen 3 and Zen 4, we take the cache line placement of the branches into account, as discussed previously.

Results. The outcome of this experiment confirms our hypothesis: the last return does not transiently execute the primed return site, and thus we have overwritten an RSB entry using a PhantomCall inside a Phantom-induced speculation window, i.e., using a nested Phantom speculation. We thus conclude that AMD microarchitectures do not require decoding of an instruction for them to believe there will be a call. If there exists a call-prediction for the target of our PhantomJmp, *the call prediction itself prematurely pushes to the RSB*, before decoding finishes. This also explains why TTE_{RSB} on Zen 3 and Zen 4 succeeds: transient execution, or even transient decode, is not required for the RSB to be manipulated.

In summary, we have a new primitive that allows us to manipulate the RSB from any instruction, without any architectural call instruction on the transient path. We refer to this new primitive as PhantomCall.

> **Observation (O4).** On all AMD Zen CPUs, we can corrupt an RSB entry using a PhantomCall.

As discussed previously, on Zen 3 and Zen 4 the PhantomJmp and the call, which we now know is a PhantomCall, need to be on cache lines following the ones using which the BTB is consulted. We now 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 realize the predictions are incorrect. If the PhantomJmp and the PhantomCall fall in the cache line using which the BTB is consulted, decoding may happen before an entry is pushed onto the RSB.

PhantomCalls hugely simplify the requirements for exploitation with TTE_{RSB}. If the PhantomCall corrupts an entry that will be used for return target predict, we can use an arbitrary disclosure gadget by injecting a PhantomCall right before it. Furthermore, we can target any return instruction in the kernel, including the return protected with *jmp2ret* on Zen 1(+) and Zen 2, by injecting a PhantomJmp on an instruction before the return. We leverage these capabilities in our end-to-end exploit, Inception, which we discuss next.

4.5 Inception

Our results in Section 4.3.2 showed that the RSB can be updated using transient instructions, enabling TTE_{RSB}. In Section 4.4.2, we demonstrated that we can manipulate the RSB in a transient window triggered by a PhantomJmp. We then discovered in Section 4.4.3 that the call instruction does not even need to be backed by an architectural instruction for it to manipulate the RSB: a predicted call, or PhantomCall, acts *before* the instruction is decoded.

However, to turn PhantomCall into an end-to-end exploit, we need to overcome several challenges. Most importantly, on Zen 1(+) and Zen 2, we need to overwrite all RSB entries to reliably trigger speculative execution at transiently inserted return sites, as discussed in Section 4.3.2. Although less entries can be overwritten on Zen 3 and Zen 4, deep return stacks would be needed to reach the injected RSB entry, complicating exploitation. We therefore investigate how we can overwrite as many RSB entries as possible by executing multiple PhantomCalls in a single speculation window.

Addressing this challenge requires new insights that we discuss in Section 4.5.1 and Section 4.5.2. We then proceed to the design of our end-to-end exploit Inception in Section 4.5.3 through Section 4.5.7. Lastly, we evaluate Inception in Section 4.5.8 through Section 4.5.10.

4.5.1 Recursive PhantomCall

To turn PhantomCalls into a practical exploit, we somehow need to execute a large number of PhantomCalls in a single speculation window. We therefore construct a chain of PhantomCalls to determine how many we can execute in a PhantomJmp speculation window. While trying to maximize the number of PhantomCalls that fit in a single speculation window, we realize that nothing prevents us from establishing a single PhantomCall that branches into itself, i.e. a *recursive* PhantomCall. We hypothesize that this may result in overwriting more RSB entries compared to transiently executing a non-recursive chain of PhantomCalls, since the instruction pointer does not change.

To measure the number of recursions possible, we monitor the amount of RSB entries that gets corrupted by our recursive PhantomCall. We repeat the experiment described in Sec-

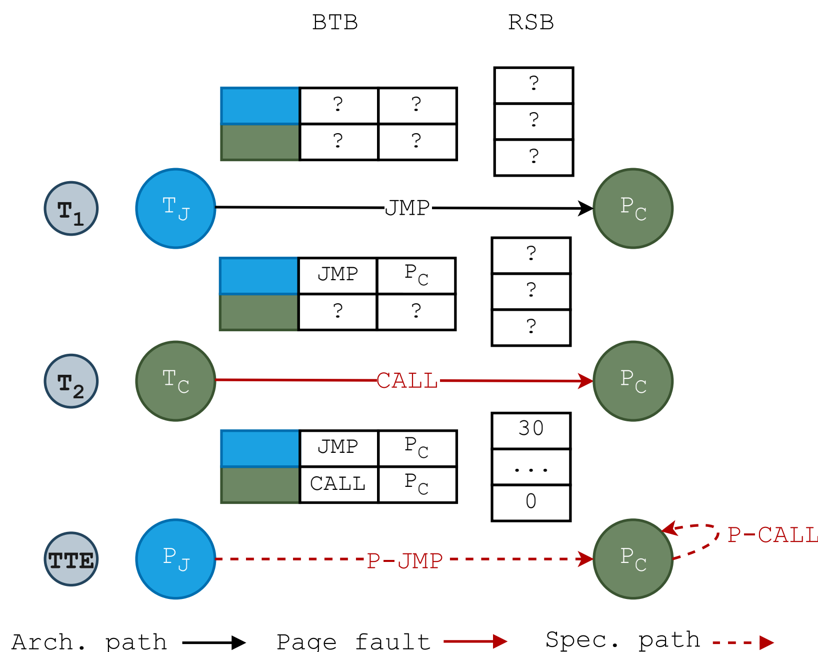


Figure 4.5: The experiment setup to test the number of entries we can pollute with a recursive PhantomCall in a PhantomJmp. Colors indicate BTB entry mapping.

tion 4.4.3, but this time T_C is a call to P_C, establishing a recursive prediction. An overview of the experiment is shown in Figure 4.5.

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 immediately after it has been inserted by T_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 [56]. Interestingly, we find this to be unnecessary on Zen 1(+) and Zen 2. We hypothesize that this could be due to a race condition that happens to be in our favor. The prediction associated with T_C may not yet have updated the BTB as soon as we are executing P_C. We verify that executing P_C again separately (i.e., not directly after execution of T_C) indeed invalidates the prediction.

Results. Our experiment shows that we can corrupt many RSB entries using our recursive PhantomCall on all Zen microarchitectures. Figure 4.6 presents the results in detail for Zen 2, showing that we are able to corrupt 18 entries of the RSB with high probability. For reference, *baseline* shows the result of negative testing, which is done by determining the number of corrupted RSB entries without provoking any transient execution.

Although we can corrupt a large number of RSB entries with our recursive PhantomCall, we find that the associated returns do not always predict to the transiently injected return sites, as also observed during our earlier experiments in Section 4.3.2. Figure 4.7 gives an overview of the entries corrupted on each Zen microarchitecture, and shows which of those corrupted entries are actually used for return target prediction. On Zen 1(+) and Zen 2, none of the returns cause transient execution of the injected return site due to the recursive PhantomCall. On Zen 3, we find that only one RSB entry is consumed for return target prediction. On Zen 4, many RSB entries are corrupted, and most of them predict back to our injected return site. The few last returns, however, are predicted using uncorrupted RSB entries.

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. In Section 4.5.13 we discuss the impact of these effects in more detail.

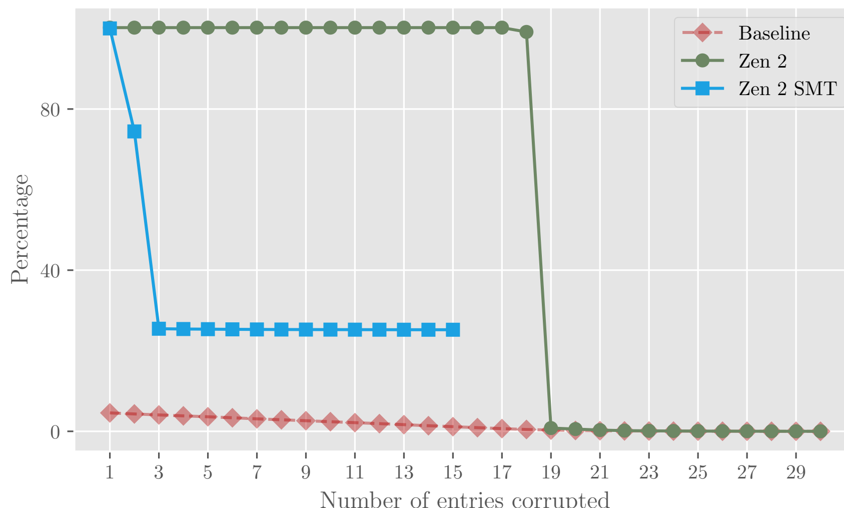


Figure 4.6: Number of RSB entries corrupted on Zen 2 over 100,000 runs, with- and without SMT. We can corrupt 18 RSB entries using our recursive PHANTOMCALL, and all 15 entries can be corrupted with SMT..

An interesting observation we make is that the call prediction at P_C is not invalidated after the TTE step for most of the iterations, unlike the prediction for the PHANTOMJMP. This may be because a resteer happens before the instructions at P_C are decoded.

Our results show that we would be able to hijack returns after a deep return stack on Zen 3 and Zen 4. Specifically, for Zen 3 we require no more or less than 17 returns between the recursive PHANTOMCALL and the victim return. On Zen 4, the recursive PHANTOMCALL needs to be followed by at least 8 returns, after which we control the return target predictions for the next 16 returns executed. We again note that these results may differ depending on various parameters, such as where the PHANTOMJMP and the recursive PHANTOMCALL are triggered.

On Zen 1(+) and Zen 2 microarchitectures, however, we do not overwrite enough RSB entries to enable arbitrary transient code execution. Our results in Section 4.3.2 showed that transiently overwriting all RSB entries leads to the corrupted entries actually being used for prediction. We therefore expect that overwriting all RSB entries using a recursive PHANTOMCALL would also trigger execution of the return site of the PHANTOMCALL. To construct an exploit on Zen 1(+) and Zen 2, we thus somehow need to overwrite all RSB entries.

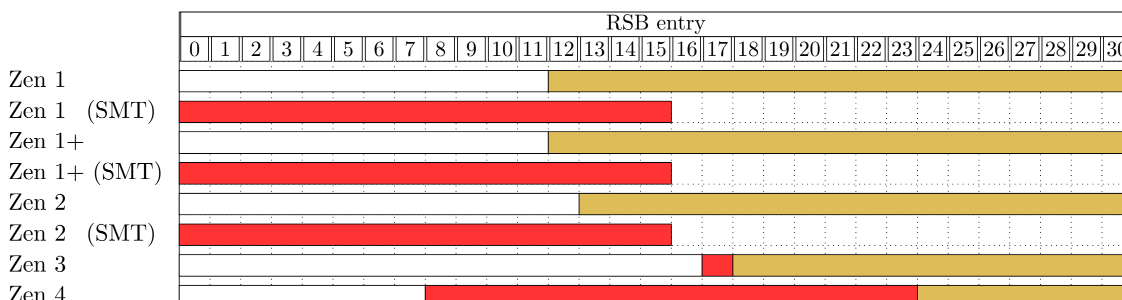


Figure 4.7: Entries affected by the recursive PHANTOMCALL. Yellow shows corrupted entries that remain unconsumed, while red indicates that an entry is used for prediction, enabling arbitrary transient code execution.

4.5.2 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 switch to *dual-threaded mode* [4], reducing the RSB to 15 entries for each thread instead of the 31 entries we have worked with so far. As shown earlier, we can poison 18 entries in a nested Phantom speculation, and we can thus potentially overwrite the entire RSB associated with a 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 4.5 reveals that we can indeed overwrite all 15 RSB entries on Zen 1(+) and Zen 2 microarchitectures. Figure 4.6 shows the success rate of overwriting all entries for Zen 2. Having overwritten all entries, our transiently injected return site is actually used by all returns issued, as shown in Figure 4.7. This means that we do not rely on deep return stacks on Zen 1(+) and Zen 2: any return can be hijacked in dual-threaded mode by issuing the recursive PhantomCall right before it is executed.

4.5.3 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(+), Zen 2 and Zen 4. Inception is not fully successful on Zen 3, as discussed later this section.

Figure 4.8 shows a visualization of Inception together with the resulting state of the BTB and RSB after each training step. In the first step T_1, the attacker executes a branch from a virtual address T_J that collides with the BTB entry of virtual address of P_J. Residing in the

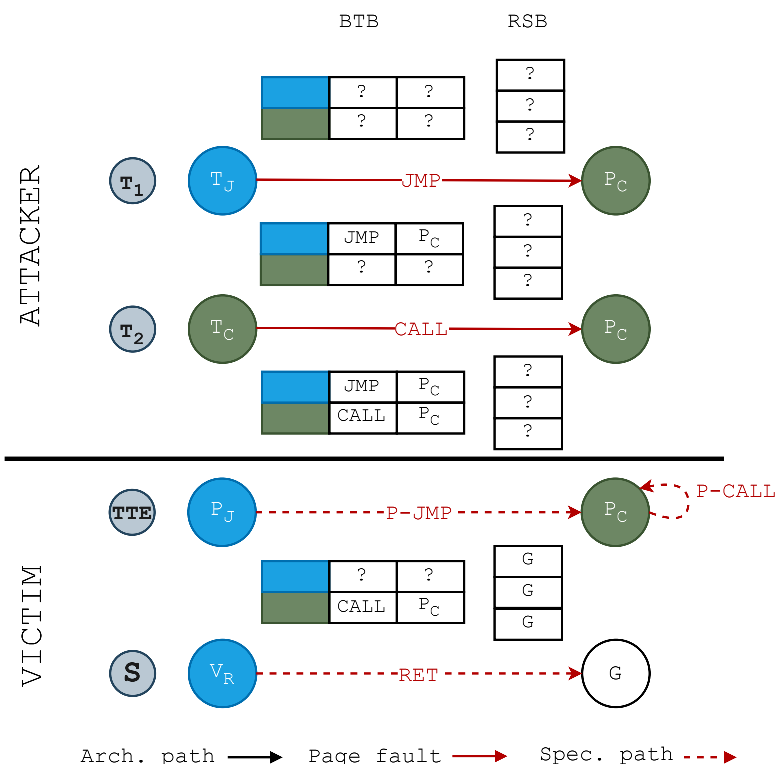


Figure 4.8: Phantom visualized. The BTB and RSB state is shown following steps T_1, T_2, and TTE. Green and blue colors indicate two different BTB mappings.

kernel address space, P_J is the address where we want to trigger the PhantomJmp that initiates the recursive PhantomCall. The victim return V_R is allocated after P_J in the control flow. P_C is the target of the PhantomJmp, at which we trigger the recursive PhantomCall to insert RSB predictions to disclosure gadget G, which immediately follows the PhantomCall in P_C . In step T_2, the attacker executes a call on virtual address T_C that collides with P_C in the BTB, which is where we want to trigger the recursive PhantomCall. The call target of T_C is set to P_C to establish a recursive BTB prediction when executing P_C. As P_C resides in kernel space, the training branches T_J and T_C will page fault when branching to it.

On Zen 3 and Zen 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 decoded, and the CPU eventually corrects the misprediction by resetting the instruction pointer back to P_J. Finally, in step S the victim return at V_R will take the prediction from the RSB. Since we have overwritten RSB entries with return site G during the TTE step, we start executing the disclosure gadget G, accomplishing a long speculation window in which we control the instructions executed.

4.5.4 Dueling recursive PhantomCalls

It may happen that the desired disclosure gadget does not exist in the kernel code. In this case, Inception can rely on executing two separate disclosure gadgets within the same transient window, that together achieve the desired operation. Inception achieves this by introducing two recursive PhantomCalls, or dueling recursive PhantomCalls, establishing a transient 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 entries need to be used for return target prediction without needing to overwrite the entire RSB, which is the case only on Zen 3 and Zen 4. Furthermore, our results in Figure 4.7 showed that on Zen 3 we can not always corrupt enough entries to overwrite multiple entries used for return target prediction.

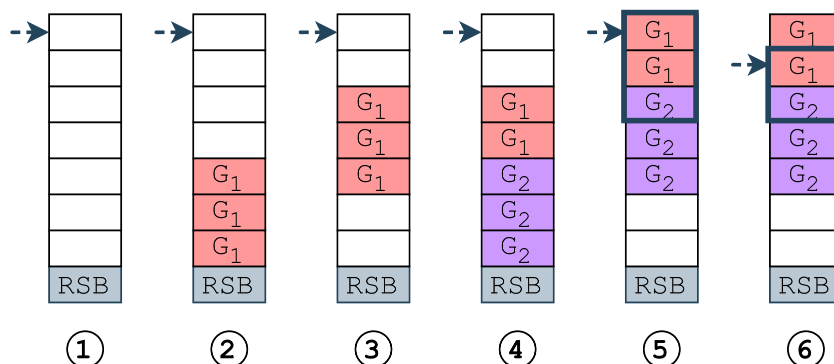


Figure 4.9: Triggering dueling recursive PhantomCalls to chain two disclosure gadgets G_1 and G_2 together.

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. We therefore give the first recursive PhantomCall a head start by triggering it a few returns prior to the second recursive PhantomCall. Figure 4.9 shows the approximate progression of the RSB state over time. Step ① shows the state before the first recursive PhantomCall, and thus the RSB is yet unaffected. Step ② shows the RSB state after triggering the first recursive PhantomCall, which precedes G_1. Step ③ shows the state after two returns. Step ④ shows the state after issuing the second recursive PhantomCall, which precedes G_2. Lastly, step ⑤ shows the state after again 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. If this is not the case, an attacker should target the next return instruction, which executes G_1 once before reaching G_2, as shown in step ⑥.

4.5.5 Victim return instruction

Having designed Inception, we proceed with searching for an appropriate 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. This is needed to achieve arbitrary information leakage through the kernel's physmap area, similarly to previous work [56, 19].

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

Previous work built an open-source framework to trace register contents in the Linux kernel at the time of executing a return instruction [56]. We use this framework to find vulnerable returns that meet our requirements.

4.5.6 Derandomizing KASLR

As in previous work [56], we derandomize KASLR in three steps. In all, we prime the BTB with the PhantomJmp and the recursive PhantomCall before we issue the system call.

① **Finding the kernel text.** According to previous work, the kernel text can be loaded at 488 possible locations [32]. We use a disclosure gadget that simply performs a data load on the attacker-controller value V_1, i.e. $*V_1$. Upon execution of the system call, the transient load will succeed only if we guess the kernel text location right. If we measure contention in the data cache using Prime+Probe, we thus deduce that we have found the correct kernel text location.

② **Finding physical address mapping.** To find the physical address of our reload buffer, we trigger a transient load to an offset from the physmap base address. We achieve this by using a disclosure gadget that adds V_1 to the physmap base address, and then dereferences the resulting address, i.e. *(*page_offset_base + V_1). Using Flush+Reload we can detect whether we have guessed the physical address correctly.

③ **Finding physmap.** According to previous work, the physmap area can start at 25600 possible locations, depending on configurations [32]. To derandomize physmap, we use the same disclosure gadget as in ①. We trigger a transient load on V_1, which is the physical address of our reload buffer added to our guess of the physmap location. Using Flush+Reload, we deduce whether we have guessed correctly.

```
1 leave
2 xor    edx,edx
3 mov    esi,edx
4 mov    edi,edx
5 jmp    0xae4021c0    ; jumps to return (jmp2ret mitigation)
```

Listing 5: The location where we trigger Phantom speculation in *__fdget_pos* at offset `0x41db94` from the start of the kernel text on Zen 1(+) and Zen 2. The jump branches into the return thunk.

4.5.7 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 loads address V_1 from memory, and uses its result to load an offset to the address in V_2, which is the address of the reload buffer in the kernel's physmap area. That is, the disclosure gadget executes $*(V_2 + *V_1)$. If V_1 or V_2 are memory locations, they first need to be loaded from memory in the desired register. Using Flush+Reload on our reload buffer, we can deduce which offset was transiently loaded, allowing us to infer the secret residing at address V_1.

4.5.8 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 can trigger the PhantomJmp on the **xor** instruction, poisoning the RSB right before we jump to the return.

Disclosure gadgets. We could find the desired gadgets with simple string matching. Listing 6 shows the disclosure gadgets found. The second line of Listing 6-top is used for steps ① and ③ of breaking KASLR, which is finding the kernel text and the physmap base location respectively. Both lines of Listing 6-top are used for step ② of breaking KASLR, i.e. finding the physical address of our reload buffer. Lastly, Listing 6-bottom presents the disclosure gadget used to leak arbitrary data.

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%.

```
1 add    r12,QWORD PTR [rip+0x9f9dfd]
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).

4.5.9 Inception on Zen 3

Our results show that building Inception for Zen 3 is rather challenging. During the *sendto()* system call we control memory locations pointed to by the **R13** register upon execution of the

```
1  call   0x8cfbe640
2  test   eax,eax       ; the location using which the BTB is indexed upon return from call
3  jg     0x8cf9040d    ; the location where we trigger the PhantomJMP (in next cacheline)
```

```
1  call   0x8cf1bbf0
2  test   eax,eax       ; the location using which the BTB is indexed 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       ; the location where we trigger the PhantomJMP (in next cacheline)
```

Listing 7: The locations where we trigger Phantom speculation in *udpv6_rcv* (top, at offset `0xd905be` from the start of the kernel) and *udpv6_queue_rcv_one_skb* (bottom, at offset `0xdbbbbe` from the start of the kernel text) on Zen 3 (bottom only) and Zen 4.

return instructions of *ip6_local_out()* and *ip6_send_skb()*. We control these memory locations using the message buffer whose address we pass in **RSI** (i.e. second argument). We trigger the PhantomJmp to our recursive PhantomCall in the *udpv6_queue_rcv_one_skb()* function, specifically on the **xor** instruction as shown in Listing 7-bottom.

Although we successfully hijack at least one of these return instruction, we are unable to leak data due to the lack of a disclosure gadget that uses **R13**, which we attempt to look for using a tool built by previous work [56] and manual string matching. Although dueling recursive PhantomCalls could overcome this issue as discussed in Section 4.5.4, we do not find two locations at which we can reliably trigger recursive PhantomCalls such that we achieve the desired RSB state. Furthermore, we find that executing a workload on the sibling hyperthread causes our recursive PhantomCall to not corrupt the desired RSB entry, proving by the fact that we do not hijack a return instruction anymore while **R13** is set to our message buffer. This may complicate an attack further, since a workload on the sibling hyperthread is helpful to increase the transient execution window, as discussed later in Section 4.5.13.

Despite the fact that we are unable to leak arbitrary data on Zen 3 using our basic setup, we are convinced that additional engineering efforts would result in Inception being effective on Zen 3 as well. For example, less conventional disclosure gadgets may be used to leak arbitrary data. In addition, the memory locations we control may be pointed to by a different register on other Linux kernel versions. We leave this challenge for future work.

4.5.10 Inception on Zen 4

Vulnerable return. On Zen 4, we also 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()*, our message buffer is reachable using the address in **RBX**. 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.

Disclosure gadgets. Listing 8 shows disclosure gadgets found for Zen 4. We find that Prime+Probe on the data cache is very noisy. Therefore, for step ① of breaking KASLR, we use a disclosure gadget that issues 3 distinct loads to our guessed kernel text address, 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 presents the arbitrary disclosure gadget, found using tools of previous work [56]. On Line 7, the secret is XORed with a value,

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

```
mov    rax,QWORD PTR [rbx+0x48]         ; Loads physmap address guess from memory
mov    rdi,QWORD PTR [rax+0xc0]
```

```
mov    rsi,QWORD PTR [rbx+0x48]         ; Loads the address of the secret from memory
lea    eax,[rdx+0x2]                    ; rdx == 0
shl    r14d,cl
mov    rcx,QWORD PTR [rbx+0x60]         ; Loads the address of the reload buffer from memory
and    edx,DWORD PTR [rbx+0x40]
movzx  eax,BYTE PTR [rsi+rax*1]         ; Loads the secret from memory (rax contains 2)
xor    eax,r14d                         ; XORs the secret with some value
and    eax,DWORD PTR [rbx+0x74]         ; ANDs the secret with an attacker-controlled value
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).

before it is used to access the reload buffer. We find that after a reboot, this value potentially changes. Therefore, INCEPTION leaks one byte of its own memory, set to 0, to determine which value the secret is XORed with.

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 using a PHANTOMJMP on the **xor** instruction, shown on Line 10 of Listing 7-bottom.

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%.

During our evaluation, we notice that INCEPTION is vulnerable to noise on Zen 4, causing unexpected hits in our reload buffer. This results in leaking incorrect bytes, decreasing its accuracy. We expect this to be caused by a memory prefetcher. We find that the (preceding)

```
mov    rax,QWORD PTR [rbx+rax*8+0x20]   ; Load address guess of physical address from memory (rax == 0)
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 [rip+0x185c43a]    ; Adds the physmap base to rax
mov    QWORD PTR [rax],rdx              ; Loads physical address from memory
```

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).

workload on the sibling hyperthread influences this behavior significantly. Therefore, before leaking the randomized data from the kernel, INCEPTION leaks 100 bytes of its own memory, to judge its performance. If the accuracy is low, the attack is restarted on a different core.

4.5.11 Leaking root password hash

We furthermore show that INCEPTION is capable of locating secrets in the physical memory. Specifically, we let INCEPTION search for `/etc/shadow` on Zen 4, 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, to speed up the searching process. We try 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.

4.5.12 AutoIBRS

A relatively low-cost mitigation for INCEPTION on Zen 4 would be AutoIBRS, a feature preventing predictions inserted in user mode from being used by the kernel. Our kernel version does not yet support the feature due to its recent introduction, but nevertheless we evaluate AutoIBRS against INCEPTION by manually enabling the feature. Our results show that AutoIBRS *does not have any effect* on the success of INCEPTION. We hypothesize that AutoIBRS is implemented too deep in the pipeline for it to prevent RSB manipulation before instructions are decoded, thus failing to be effective against INCEPTION. This hypothesis is in line with our results in Chapter 3, where we showed that AutoIBRS does not prevent transient fetch of the predicted target due to PHANTOM speculation.

4.5.13 Optimizations

Increasing the transient window. As mentioned before, the transient window of the PHANTOMJMP is dependent on various factors. First, evicting a cache line preceding the address where the PHANTOMJMP is triggered typically improves our chances of polluting a large number of RSB entries. Likewise, it is beneficial on Zen 1(+) and Zen 2 to trigger the PHANTOMJMP at the target location of an architectural branch for which a BTB prediction exists. While establishing the exact cause of this behavior is difficult, we believe the context in which we execute instructions influences the time it takes to decode them. Given our observations, we do not exclude the possibility that, in certain circumstances, one would be able to poison all 31 RSB entries using a PHANTOMJMP-induced recursive PHANTOMCALL. Hence, we do not think disabling hyperthreading would be a sufficient mitigation against INCEPTION on Zen 1(+) and Zen 2.

Evicting the stack. After polluting the RSB successfully, the attacker can increase the transient window of the mispredicted return by evicting the stack address containing the return address from the cache hierarchy, as also pointed out in earlier work [56]. This optimization, however, is no longer fully possible since our target systems have kernel stack randomization enabled by default [45]. Instead, INCEPTION evict an arbitrary cache set on the sibling hyperthread, causing some iterations of the attack to enjoy a long transient window.

Masking out bits. The disclosure gadget found for arbitrary information disclosure on Zen 4, shown in Listing 8-bottom, masks the secret value with an attacker-controlled value on Line 8. This mask can be used by an attacker to leak bit by bit, which we do to improve performance. Furthermore, this may be helpful if the attacker is searching for specific kind of data: if the first bit(s) leaked do not match your pattern, the byte can be skipped, speeding up the search.

4.6 Alternative TTE variants

We have demonstrated how one variant of TTE can be leveraged on AMD machines to leak arbitrary data. In this section, we will discuss the security impact of other potential variants of TTE. We then systematically explore which 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. Furthermore, we hope to stimulate the development of effective mitigations against these new attack surfaces.

4.6.1 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.

First, conditional branches in the kernel may be followed by call instructions, similar to the example with the call-and-disclose gadget presented in Section 4.3.2. We previously showed that on AMD, transiently executed call instructions triggered due to BTB-misprediction or PHANTOM speculation can manipulate the state of the RSB. Therefore, by 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., $\text{TTE}_{\text{PHT-RSB}}$). On AMD Zen 3 and Zen 4, we found that we do not need to control all RSB entries to reliably trigger misprediction to our transiently injected return site, after deep call stacks. If the return site of the call contains a disclosure gadgets, this would allow an attacker to leak arbitrary data.

Second, on newer Intel microarchitectures that support eIBRS, BHI [9] has shown that although kernel branch predictions are isolated from user mode, an attacker can still influence the choice made between previously seen branch destinations in the target privilege level. Exploiting this to leak arbitrary data, however, requires a disclosure gadget at a previously chosen destination of the target indirect branch. With TTE, we can loosen this requirement. Instead of requiring a disclosure gadget, we use an indirect branch at a previously executed destination, which we then leverage to transiently train the BTB inside the kernel (i.e., $\text{TTE}_{\text{BTB-BTB}}$).

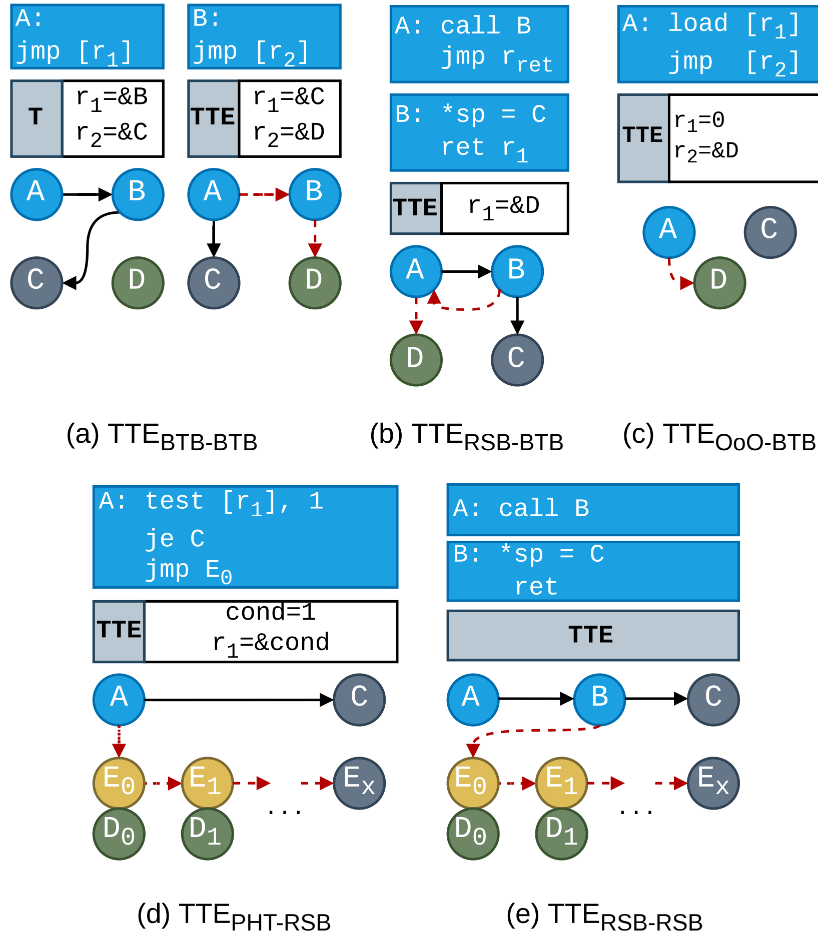
Lastly, target locations of conditional branches in the kernel may contain indirect branches when retpolines are disabled (e.g., on Intel CPUs that support eIBRS). Transient out-of-bound memory accesses are prevented by index masking [59], but this mitigation is not necessarily applied to target locations of indirect branches. 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 code execution whenever the indirect branch is executed architecturally as also discussed in Section Section 4.3.1 (i.e., $\text{TTE}_{\text{PHT-BTB}}$).

Having established the scenarios in which TTE would bypass existing mitigations, we now evaluate different variants of TTE.

4.6.2 Testing for TTE variants

We discussed $\text{TTE}_{\text{PHT-BTB}}$ and $\text{TTE}_{\text{BTB-RSB}}$ in Section 4.3. We now explore other possible variants of TTE. Figure 4.10 describes the experiments that we designed. Similar to experiments in Section 4.3, A and B code locations are used to manipulate the branch predictors, C is a barrier and D is a disclosure gadget. To avoid interference across experiments, we issue the IBPB command to flush branch predictor state in each round.

TTE_{BTB} experiments. To test for $\text{TTE}_{\text{BTB-BTB}}$, we first train the branch predictor to branch from A to B in a preparatory step called T. In 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 its branch target. To test for $\text{TTE}_{\text{RSB-BTB}}$, we train the RSB to return to the instruction immediately following the call in A. We prevent this



(a) $\text{TTE}_{\text{BTB-BTB}}$ (b) $\text{TTE}_{\text{RSB-BTB}}$ (c) $\text{TTE}_{\text{OoO-BTB}}$

(d) $\text{TTE}_{\text{PHT-RSB}}$ (e) $\text{TTE}_{\text{RSB-RSB}}$

Figure 4.10: Training in Transient Execution (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.

from architecturally executing by overwriting the architectural 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 $\text{TTE}_{\text{OOO-BTB}}$, the training branch source follows a load instruction that faults in step TTE, 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. Interestingly, previous work [30, 56] has made use of $\text{TTE}_{\text{OOO-BTB}}$ for poisoning a BTB entry used in a different security context.

TTE_{RSB} experiments. To test for $\text{TTE}_{\text{PHT-RSB}}$, we again 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 $\text{TTE}_{\text{RSB-RSB}}$, 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 sites (D_i) to the RSB. Testing $\text{TTE}_{\text{OoO-RSB}}$ is challenging, since an invalid memory access as done for $\text{TTE}_{\text{OoO-BTB}}$ requires kernel-level page-fault handling, which trashes the RSB state. We therefore excluded this experiment for the RSB.

Results. Table 4.2 shows the results of running all TTE experiments 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 is can be overcome by fine-tuning them for the given microarchitecture. The results show that transient training of the BTB 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

Model	Microarch.	Year	$TTE_{BTB\text{-}BTB}$	$TTE_{PHT\text{-}BTB}$	$TTE_{RSB\text{-}BTB}$	$TTE_{OOO\text{-}BTB}$	$TTE_{BTB\text{-}RSB}$	$TTE_{PHT\text{-}RSB}$	$TTE_{RSB\text{-}RSB}$
Ryzen 5 1600X	Zen	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 4.2: CPUs that are vulnerable speculative training of the BTB, i.e., $TTE_{*\text{-}BTB}$, and of the RSB, i.e., $TTE_{*\text{-}RSB}$.

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 transient training of the RSB, although the injected entry is not always used, as discussed in Section 4.3.2. 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 ($TTE_{PHT\text{-}RSB}$ on AMD, $TTE_{BTB\text{-}BTB}$ on Intel, and $TTE_{PHT\text{-}BTB}$ on both) are realistic on the microarchitectures that we considered, and future mitigations should consider their attack surfaces.

4.7 Mitigation

In this section we consider various mitigations against INCEPTION and other TTE attacks. To properly prevent INCEPTION, hardware modifications are necessary. For the time being, however, we conclude that a full flush of the branch predictor state is necessary to prevent INCEPTION. Unfortunately, this comes with a substantial performance penalty, which we quantify in this at the end of this section.

4.7.1 Analysis of possible mitigations

Synchronization. AMD and Intel recommend the usage of `lfence` against Spectre-PHT attacks [6, 13]. While such serializing instructions are effective against $TTE_{PHT\text{-}*}$ variants, they are ineffective against other variants of TTE, most notably INCEPTION. Furthermore, finding all gadgets in a victim code vulnerable to $TTE_{PHT\text{-}*}$ is nontrivial.

Address Space Isolation (ASI). A possible direction for mitigating TTE could be to reduce the number of secrets present in the kernel. For example, INCEPTION achieves arbitrary memory leakage through the physmap area of the kernel. Address Space Isolation (ISA) attempts to thwart transient execution attacks by restricting the mapped kernel address space when possible [47]. However, for certain operations, the entire address space must be available, and thus ASI is not a full mitigation against TTE, or in particular against INCEPTION.

Micro-architecture	Model	Performance overhead	IBPB cost (median)
Zen	Ryzen 5 1600X	239.2 % (single) 198.4 % (multi) *no SMT mode* 234.3 % (single) 216.9 % (multi)	8,803 cycles
Zen +	Ryzen 5 2600X	226.6 % (single) 183.1 % (multi) *no SMT mode* 205.0 % (single) 204.0 % (multi)	8,196 cycles
Zen 2	Ryzen 5 3600X	130.1 % (single) 95.2 % (multi)	1,306 cycles
Zen 2	EPYC 7252	128.6 % (single) 93.1 % (multi)	1,306 cycles
Zen 3	Ryzen 5 5600G	35.05 % (single) 29.35 % (multi)	738 cycles
Zen 4	Ryzen 7 7700X	59.90 % (single) 87.33 % (multi)	962 cycles

Table 4.3: Performance overhead of single and multicore benchmarks with the IBPB-on-entry mitigation, including the cost of issuing one IBPB. We benchmark with and without Simultaneous Multi-Threading (SMT) enabled when relevant.

Stopping transient training. TTE relies on the fact that BPU structures are trained during transient execution, before instructions are retired. By postponing BPU updates to post-retirement, Inception and all other TTE attacks can be prevented. However, this may reduce accuracy of speculative execution considerably. As an example, a call and a return may be in the same fetch window. If the call can only update the RSB upon retirement, the return would never benefit from a correct RSB prediction. Instead, the return would desynchronize the RSB from the architectural stack. One can imagine similar scenarios for other types of branches. It is thus unlikely that CPU vendors will fully eliminate transient training.

Speculative BPU structures. By implementing dedicated speculative variants of BPU 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 creating speculative variants of all BPU structures, TTE attacks can be prevented. However, while the RSB typically contains only 16 or 32 entries, the BTB is much larger. For example, AMD's BTBs contain thousands of entries. Creating a speculative counterpart for every BPU structure is thus a costly operation, and unlikely to be implemented.

Isolating the branch predictor state. Hardware mitigations that fully isolate BPU states between privilege levels would mostly mitigate TTE, and in particular Inception. However, our results show that AMD's AutoIBRS does not prevent Inception, suggesting it operates too deep in the pipeline for it to be effective. Likewise, in Section 4.6.1 we describe TTE variants that would be possible in spite of Intel eIBRS, due to the fact that the BHB is not properly isolated. We conclude that isolating the branch predictor state is only effective when *all* BPU structures are isolated, and checks must be performed at the very start of the pipeline.

4.7.2 IBPB-on-entry

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 mitigation enabled. We compute

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

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 needed for issuing IBPB (using the precise APREF clock cycle counter [1]) over 1 M samples.

Table 4.3 shows the results of our benchmarks. Because Zen 1(+) and Zen 2 do not support STIBP, for a complete mitigation, we also take benchmark with SMT disabled. Clearly, IBPB is an expensive operation, particularly for older Zen microarchitectures. However, we conclude that for the time being, it is unfortunately the only available mitigation against INCEPTION.

Chapter 5

Related Work

In this section we discuss work related to PHANTOM, TTE and INCEPTION. Specifically, we discuss microarchitectural side channels, transient execution attacks and their mitigations.

Microarchitectural side channels. Many variants of microarchitectural side channels have been studied in the literature. The first timing attacks focused on CPU caches [39, 60, 21, 15, 20], but later work researched side channels on other microarchitectural components such as µop caches [44], execution units [10], branch predictors [16, 17, 33], interconnects [40, 14] and prefetchers [62, 34].

Cache attacks. Caches are a widely studied component in microarchitectural sidechannels, and many attacks rely on cache side channels, including those presented in this thesis. Osvik et al. introduced Prime+Probe and Evict+Time [39]. Yarom et al. introduced Flush+Reload, a more robust cache side channel which relies on shared memory between the attacker and victim [60]. More recent works have introduced other types of cache attacks [15, 42, 21].

Transient execution attacks. In 2018, Kocher et al. presented *Spectre*, the first transient execution attack that leaks memory by manipulating BPU structures. Spectre-PHT (Spectre v1), also known as Bounds Check Bypass (BCB), triggers the misprediction of a conditional branch to force out-of-bounds array reads. Spectre-BTB (Spectre v2) injects a branch target, yielding arbitrary transient code execution when used by a victim indirect branch.

One class of transient execution attacks are those that manipulate the RSB to trigger mispredictions. Maisuradze et al. [36], Koruyeh et al [31], and Wikner et al. [58] revealed such attacks. These works adversarially manipulate the RSB, like INCEPTION as discussed in Chapter 4. However, unlike these previous works, INCEPTION does so in a transient window.

Another class of transient execution attacks relies on misprediction that can be detected by the decoder. Wikner et al. revealed that predictions can be served for arbitrary instruction on AMD CPUs [57], giving rise to PHANTOMJMPS. Likewise, Zhang et al. found that Intel microarchitectures are also susceptible to some limited form of PHANTOM speculation, where the CPU confuses the exact location of a branch inside a tracking window of the µop-cache [61]. Lastly, Wieczorkiewicz [54, 55] showed that AMD microarchitectures suffer from Straight-Line Speculation (SLS) when no prediction is available in the BTB. In Chapter 3, we continued investigation of decoder-detectable mispredictions on AMD CPUs, presenting novel insights.

Bypassing mitigations of transient execution attacks. To mitigate Spectre-BTB, Turner et al. proposed retpolines [51], and AMD adopted their own version, relying on an lfence barrier. Both of these mitigations have been bypassed. Milburn et al. showed that AMD's retpoline is racy, and Wikner and Razavi [56] presented Retbleed, showing that BTB predictions can be served for return instructions under certain circumstances. To mitigate Retbleed on AMD, *jmp2ret* was deployed to only allow execution of a single return instruction, sanitized upon kernel entry. In Chapter 4, we present INCEPTION which bypasses this mitigation by transiently manipulating the RSB instead.

In addition, vendors such as Intel and AMD introduced mitigations that restrict speculation at the hardware level, such as IBRS. However, Branch History Injection (BHI) showed that Intel's eIBRS does not fully isolate branch prediction states, allowing an attacker to trigger mispredictions by manipulating the BHB only [9]. Similarly, in Chapter 4 we reveal that INCEPTION is effective despite AMD's AutoIBRS. However, unlike BHI, we hypothesize that this is because of performing the check too late in the pipeline.

Microarchitectural Data Sampling. Microarchitectural Data Sampling (MDS) attacks leak data from various microarchitectural buffers [46, 11, 52, 43]. To bring desired secret data in such buffers, MDS attacks may make use of MDS gadgets which perform a transient load to the desired secret in the victim security context. Since only Intel CPUs are known to be vulnerable to MDS, these gadgets were previously believed to be relatively harmless on AMD CPUs. Wikner et al. previously suggested that PHANTOM could be used to re-purpose MDS gadgets on AMD machines to leak arbitrary data, which was later independently reported by AMD [7]. In Chapter 3 we show such scenario to be practical, by presenting a PoC that leaks arbitrary kernel memory using a realistic dummy MDS gadget.

Chapter 6

Conclusion

In this thesis, we investigated whether remaining attack surface of transient execution attacks exists. In Chapter 3 we addressed our first research question by systematically exploring PHANTOM speculation, a class of transient execution attacks that relies on decoder-detectable mispredictions. We show that PHANTOM allow us to derandomize physmap KASLR on Zen 1 and Zen 2, and to consequently find virtual-to-physical address mappings of our user space program. We also prove that leaking arbitrary data with PHANTOM is practical, by presenting a PoC that leaks kernel memory on Zen 2 using a realistic dummy MDS gadget in the kernel. Furthermore, we reveal that mitigations are only partially effective against PHANTOM attacks.

As a second research question, we wished to understand whether there exists attack surface that does not rely on gadgets that are actively being patched. In Chapter 4 we tackled this research challenge by introducing **T**raining in **T**ransient **E**xecution (TTE). We show that TTE expands the attack surface of transient execution attacks by training BPU structures such as the BTB and RSB during transient execution, bypassing sanitization techniques.

As a last research question, we wanted to understand whether PHANTOM enables TTE with less requirements on the victim code. We show in Chapter 4 that PHANTOM allows us to perform TTE by abusing the CPU as a confused deputy. We discover that a PHANTOMCALL trains the RSB on all AMD Zen microarchitectures, allowing us to construct an infinite transient hardware loop that poisons the RSB with an attacker-controlled address. We construct our end-to-end attack INCEPTION that arbitrary memory at a rate of 39 bytes/s with an accuracy of 93.5%, despite all existing mitigations against transient execution attacks deployed in the Linux kernel. Furthermore, we show that INCEPTION leaks the root password hash in 40 minutes on Zen 4, in 6 of our 10 attempts. Our analysis shows that a full flush of the BPU is necessary to prevent INCEPTION, and we demonstrates that this comes with a substantial performance penalty.

We hope to inspire future research on PHANTOM and TTE, and to stimulate future work on developing efficient mitigations against these classes of attacks.

Bibliography

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

[2] AMD. "AMD64 Technology Indirect Branch Control Extension". In: (2018). URL: https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf.

[3] AMD. *Security Analysis of AMD Predictive Store Forwarding.* Accessed on 13.5.2023. 2021. URL: https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf.

[4] AMD. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors.* https://www.amd.com/en/support/tech-docs/software-optimization-guide-for-amd-family-17h-models-30h-and-greater-processors. accessed on 1.2.2023. Mar. 2020.

[5] AMD. *Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors.* https://www.amd.com/en/support/tech-docs/56665-software-optimization-guide-for-amd-family-19h-processors-pub. accessed on 7.2.2023. Mar. 2020.

[6] AMD. *SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS.* Accessed on 21.5.2023. 2023. URL: https://www.amd.com/system/files/documents/software-techniques-for-managing-speculation.pdf.

[7] AMD. *TECHNICAL GUIDANCE FOR MITIGATING BRANCHTYPE CONFUSION.* Accessed on 1.8.2022. 2022. URL: https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf.

[8] ARM. *Straight-line Speculation.* Accessed on 8.2.2023. 2020. URL: https://developer.arm.com/-/media/Arm%5C%20Developer%5C%20Community/PDF/Security%5C%20Update%5C%2008%5C%20June%5C%202020/Straight-line_Speculation-v1.0.pdf.

[9] Enrico Barberis et al. "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks". In: *USENIX Security.* 2022.

[10] Atri Bhattacharyya et al. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* CCS. https://doi.org/10.1145/3319535.3363194. Association for Computing Machinery, 2019.

[11] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).* ACM. 2019.

[12] 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. Accessed on 8.2.2023. 2022.

[13] Intel Corporation. *Analyzing Potential Bounds Check Bypass Vulnerabilities*. Accessed on 21.5.2023. 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analyzing-bounds-check-bypass-vulnerabilities.html.

[14] Miles Dai et al. "Don't Mesh Around:{Side-Channel} Attacks and Mitigations on Mesh Interconnects". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2857–2874.

[15] Craig Disselkoen et al. "Prime+abort: A timer-free high-precision l3 cache attack using intel {TSX}". In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 51–67.

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

[17] Dmitry Evtyushkin et al. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2018, pp. 693–707.

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

[19] Enes Göktas et al. "Speculative Probing: Hacking Blind in the Spectre Era". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1871–1885.

[20] Ben Gras et al. "ASLR on the Line: Practical Cache Attacks on the MMU." In: *NDSS*. Vol. 17. 2017, p. 26.

[21] Daniel Gruss et al. "Flush+Flush: a fast and stealthy cache attack". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 279–299.

[22] Daniel Gruss et al. "KASLR is Dead: Long Live KASLR". In: *Engineering Secure Software and Systems*. 2017.

[23] Marco Guarnieri et al. "Spectector: Principled detection of speculative information flows". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1–19.

[24] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. accessed on 1.2.2023. Apr. 2022.

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

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

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

[28] Brian Johannesmeyer et al. “Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel”. In: *NDSS*. Feb. 2022. URL: `https://download.vusec.net/papers/kasper_ndss22.pdf`.

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

[30] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. 2019.

[31] Esmaeil Mohammadian Koruyeh et al. “Spectre Returns! Speculation Attacks using the Return Stack Buffer”. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. `https://www.usenix.org/conference/woot18/presentation/koruyeh`. USENIX Association, 2018.

[32] Jakob Koschel et al. “TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs”. In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2020, pp. 309–321.

[33] Sangho Lee et al. “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing.” In: *USENIX Security Symposium*. Vol. 19. 2017, pp. 16–18.

[34] Moritz Lipp, Daniel Gruss, and Michael Schwarz. “{AMD} Prefetch Attacks through Power and Time”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 643–660.

[35] Moritz Lipp et al. “Take a way: Exploring the security implications of AMD’s cache way predictors”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020, pp. 813–825.

[36] Giorgi Maisuradze and Christian Rossow. “Ret2Spec: Speculative Execution Using Return Stack Buffers”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS. 2018.

[37] 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.

[38] Oleksii Oleksenko et al. “SpecFuzz: Bringing Spectre-type vulnerabilities to the surface”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1481–1498.

[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Cryptographers’ track at the RSA conference*. 2006, pp. 1–20.

[40] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. “Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical.” In: *USENIX Security Symposium*. 2021, pp. 645–662.

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

[42] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. “Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2906–2920.

[43] Hany Ragab et al. “CrossTalk: Speculative Data Leaks Across Cores Are Real”. In: *S&P*. 2021.

[44] Xida Ren et al. “I see dead μops: Leaking secrets via Intel/AMD micro-op caches”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 361–374.

[45] Elena Reshetova. *[RFC PATCH] x86/entry/64: randomize kernel stack offset upon syscall.* https://lkml.org/lkml/2019/3/18/246. Accessed on 8.2.2023. 2019.

[46] Stephan van Schaik et al. "RIDL: Rogue In-flight Data Load". In: *S&P*. May 2019.

[47] Junaid Shahid and Ofir Weisse. *https://lwn.net/Articles/909469/.* accessed on 02.02.2023. 2022. URL: https://lwn.net/ml/linux-kernel/20220223052223.1202152-1-junaids@google.com/.

[48] Teja Singh et al. "2.1 Zen 2: The AMD 7nm Energy-Efficient High-Performance x86-64 Microprocessor Core". In: *2020 IEEE International Solid-State Circuits Conference-(ISSCC).* IEEE. 2020, p. 5.

[49] Daniel Sneddon. *[PATCH 5.4 14/15] x86/speculation: Add RSB VM Exit protections.* 2022. URL: https://lkml.org/lkml/2022/8/9/728.

[50] *The Linux kernel user's and administrator's guide: Spectre Side Channels.* Accessed on 29.1.2022. URL: https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst.

[51] Paul Turner. "Retpoline: a software construct for preventing branch-target-injection". In: (2018). URL: https://support.google.com/faqs/answer/7625886.

[52] Jo Van Bulck et al. "LVI: Hijacking transient execution through microarchitectural load value injection". In: *41th IEEE Symposium on Security and Privacy (S&P'20).* 2020, pp. 1399–1417.

[53] Guanhua Wang et al. "oo7: Low-overhead defense against spectre attacks via program analysis". In: *IEEE Transactions on Software Engineering* (2019).

[54] Pawel Wieczorkiewicz. *The AMD Branch (Mis)predictor: Just Set it and Forget it!* Accessed on 8.2.2023. 2022. URL: https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it.

[55] Pawel Wieczorkiewicz. *The AMD Branch (Mis)predictor Part 2: Where No CPU has Gone Before (CVE-2021-26341).* Accessed on 8.2.2023. 2022. URL: https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before.

[56] Johannes Wikner and Kaveh Razavi. "Retbleed: Arbitrary Speculative Code Execution with Return Instructions". In: *USENIX Security.* 2022. URL: https://comsec.ethz.ch/wp-content/files/retbleed_sec22.pdf.

[57] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. "Addendum to Retbleed: Arbitrary Speculative Code Execution with Return Instructions". In: Aug. 2022.

[58] Johannes Wikner et al. "Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks". In: *16th IEEE Workshop on Offensive Technologies (WOOT'22).* https://comsec.ethz.ch/wp-content/files/spring_woot22.pdf. IEEE, May 2022.

[59] Dan Williams. *LKML: [PATCH v6 02/13] array_index_nospec: sanitize speculative array de-references.* https://lore.kernel.org/lkml/151727414808.33451.1873237130672785331.stgit@dwillia2-desk3.amr.corp.intel.com/. Accessed on 8.2.2023. 2018.

[60] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium.* 2014, pp. 719–732.

[61] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. "Exploring branch predictors for constructing transient execution trojans". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020, pp. 667–682.

[62] Zhiyuan Zhang et al. "BunnyHop: Exploiting the Instruction Prefetcher". In: (2023).

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

[64] Jordy Zomer and Alexandra Sandulescu. *Linux Kernel: Spectre-v1 gadgets*. 2023. URL: `https://github.com/google/security-research/security/advisories/GHSA-m7j5-797w-vmrh`.

Appendix A

PHANTOM: Collision with kernel addresses

Prior to the start of this thesis, we reverse engineered cross-privilege collision functions of the BTB on Zen 3 and Zen 4. For completeness, this Appendix describes our efforts and its results.

Wikner and Razavi showed that triggering a misprediction on a kernel address can be achieved from user space by branching to a kernel address and catching the resulting page fault [56]. In order to collide with the desired kernel address, they reverse engineer BTB indexing functions. However, they did not discover cross-privilege functions on AMD Zen 3. Furthermore, AMD Zen 4 was not yet released. To evaluate our primitives and build exploits using them, we need to reverse engineer the cross-privilege BTB indexing functions on these newer microarchitectures.

We start on Zen 3 by allocating a kernel address K using a custom kernel module, which contains *nops* and finally a return instruction. By changing the Page Table Entry (PTE) attributes of address K, we make it accessible to user space.

Brute forcing. We first attempt to create collisions with K by brute forcing a pattern such that, when applied to the kernel address K, it yields a user-space address that collides with K, as done in [56]. Using performance counters and timing results we determine whether a collision was successful. In line with previous work, however, this approach does not yield any results between user- and kernel addresses when flipping up to 6 bits. A possible reason of failing to find collisions could be that bit 47 is involved in multiple functions, requiring us to to flip many bits. Since brute forcing all combinations with more than 6 bits takes an unreasonable amount of time, we consider an alternative approach.

SMT solver. Instead, we will try to randomly find collisions between user- and kernel addresses, and then observe patterns in the addresses that collide. For this, we use a Z3 SMT solver, as done in previous work [35]. For each kernel address K, we collect lists L_K of user space addresses that collide with the kernel address. To shrink the search space, we do not randomize the lower twelve bits of our user space addresses. Instead, we set them equal to K_{0-11}. We wish to find functions on address bits, such that they all yield the same value for K and all addresses in L_K. For this, we attempt to find coefficients for the equation system $(x_0 \times A_0) \oplus (x_1 \times A_1) \oplus ... \oplus (x_{46} \times A_{46}) \oplus (1 \times A_{47}) = y$ such that it yields the same value y for all addresses that collide. At the same time, we impose $x_0 + x_1 + ... + x_{46} + x_{47} \leq n$, where n is the maximum number of coefficients set to 1, which we gradually increase. This is to prevent solutions that themselves consists of also valid solutions with less bits set.

Results. Our results are shown in Figure A.1, and were found when $n = 4$. Some functions found are omitted, since they are non-unique. Specifically, we find that whenever b_{13} is toggled in the randomly generated user-space address with respect to K, b_{17} is toggled as well. Likewise, whenever b_{12} is toggled, b_{16} is flipped as well, and vice versa. In essence, that means that these

$$
\begin{aligned}
f_0 &= b_{47} \oplus b_{35} \oplus b_{23} & f_1 &= b_{47} \oplus b_{36} \oplus b_{24} \oplus b_{12} \\
f_2 &= b_{47} \oplus b_{37} \oplus b_{25} \oplus b_{13} & f_3 &= b_{47} \oplus b_{38} \oplus b_{26} \oplus b_{14} \\
f_4 &= b_{47} \oplus b_{39} \oplus b_{26} \oplus b_{13} & f_5 &= b_{47} \oplus b_{39} \oplus b_{27} \oplus b_{15} \\
f_6 &= b_{47} \oplus b_{40} \oplus b_{28} \oplus b_{16} & f_7 &= b_{47} \oplus b_{41} \oplus b_{29} \oplus b_{17} \\
f_8 &= b_{47} \oplus b_{42} \oplus b_{30} \oplus b_{18} & f_9 &= b_{47} \oplus b_{43} \oplus b_{31} \oplus b_{19} \\
f_{10} &= b_{47} \oplus b_{44} \oplus b_{32} \oplus b_{20} & f_{11} &= b_{47} \oplus b_{45} \oplus b_{33} \oplus b_{21}
\end{aligned}
$$

Figure A.1: Functions for creating cross-privilege collisions in the BTB on Zen 3. Least significant 12 bits not considered.

bits are used in multiple, partially overlapping functions. Therefore, we erroneously obtained functions almost-identical to the ones presented.

Comparing our results with those in [56], we see that we mostly add bit 47 to functions previously found. However, we did not find some of the functions previously discovered, potentially because they do not involve bit 47. We also observe some functions that were previously not found.

Overlapping functions. While trying to create collisions with kernel addresses by flipping multiple bits according to the functions found, we discovered that using lower bits shown in Figure A.1 does not yield a colliding addresses. We suspect that this is due to overlapping functions, just as b_{12}, b_{13}, b_{16} and b_{17} are used in multiple functions. These functions may not involve bit 47, or use address bits we did not consider. Overlapping functions may be because some functions are used for tag generation, while others are used for set selection. Therefore, to create collisions, one should use the higher bits (i.e. the first three bits of each function). As an example, for a kernel address K, one can obtain a user-colliding address by computing $K \oplus$ `0xffffbff800000000` or $K \oplus$ `0xffff8003ff800000`. We confirm both of these patterns to work on AMD Zen 4 as well.

Appendix B

PHANTOM: Breaking code KASLR

μarch	Model	Accuracy	Median time
Zen 2	AMD EPYC 7252	97%	4.09 s
Zen 3	Ryzen 5 5600G	100%	1.38 s
Zen 4	Ryzen 7 7700X	95%	1.23 s

Table B.1: Accuracy and median time needed to derandomize kernel image location on AMD Zen microarchitectures using **P1**, over 100 runs.

Prior to the start of this thesis, we developed an exploit that derandomizes code KASLR on AMD microarchitectures using PHANTOM. For completeness, this Appendix describes the exploit and its performance results. We run Linux kernel 5.19.0-28-generic with the latest mitigations enabled.

KASLR places the kernel image in one of 488 possible locations [32]. We can detect at which location the kernel resides by using **P1**. The basic idea is to prime the BTB with each location and then test whether an instruction fetch happened through a Prime+Probe attack on the instruction cache. We decide to target the `getpid()` system call as the victim function that we try to bruteforce. Listing B.1 shows the exact victim instruction we target. We train the BTB for this *nop* instruction to speculate as a *jmp** instruction to a target L1 instruction cache set. In summary, for each guess, we ① prime a chosen cache set, ② prime the BTB with a branch to an address falling in the chosen cache set, ③ issue `getpid()` and ④ probe the cache set.

```
nop     DWORD PTR [rax+rax*1+0x0]
push    rbp
mov     rbp,rsp
```

Listing B.1: We trigger speculation at the nop instruction in `__task_pid_nr_ns()`. Found at kernel offset 0xf6520.

Results. We run our KASLR exploit 100 times on our AMD Zen machines, each time rebooting the machine to refresh KASLR. Table B.1 presents the success rate and median time needed to derandomize the kernel text location.