ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

COMSEC

An Evaluation of Speculative Retbleed

Bachelor Thesis

Author: Jean-Claude Graf

Tutor: Johannes Wikner

Supervisor: Prof. Dr. Kaveh Razavi

February 2022 to August 2022

Abstract

Speculation and branch prediction are inevitable design principles for modern and fast processors. However, Spectre and an ever-increasing number of Spectre-like attacks have shown the security vulnerabilities induced by these features. RETBLEED has recently been added to the list of such vulnerabilities. As RETBLEED's *Branch Target Injection* (BTI) method triggers *Page Fault*s, RETBLEED was considered detectable and mitigatable in the system's *Page Fault* handler.

In this thesis, we evaluate speculative BTI on various microarchitectures and show that it can be used to poison the *Branch Target Buffer* across privilege boundaries without raising any *Page Fault*s. To the best of our knowledge, this has never been explicitly shown before. Based on these insights, we introduce a new variant of RETBLEED that uses speculative BTI, resulting in a *Page Fault*-free variant of RETBLEED. The successful development of this variant confirms that RETBLEED cannot be mitigated in the *Page Fault* handler of the system and that more in-depth solutions are required.

In-depth mitigations are necessary to secure the CPU, which will likely affect performance significantly. We have benchmarked the patches to evaluate the induced overhead. While the overhead depends on the microarchitectures, it lies in the range of 13 % to 37 %.

Contents

Abbreviations

BCB *Bound Check Bypass*

BHB *Branch History Buffer*

BHI *Branch History Injection*

BPU *Branch Prediction Unit*

BTB *Branch Target Buffer*

BTI *Branch Target Injection*

CP *Cross Privilege*

eIBRS *Enhanced Indirect Branch Restricted Speculation*

FOSS *Free and Open Source Software*

IBPB *Indirect Branch Prediction Barrier*

IBRS *Indirect Branch Restricted Speculation*

ISA *Instruction Set Architecture*

KASLR *Kernel Address Space Layout Randomization*

LIFO *Last In First Out*

LLC *Last-Level Cache*

MMU *Memory Management Unit*

OoO *Out-of-Order*

OS *Operating System*

PA *Physical Address*

PC *Program Counter*

PF *Page Fault*

PoC *Proof of Concept*

RSB *Return Stack Buffer*

SLS *Straight Line Speculation*

SMAP *Supervisor Mode Access Prevention*

SMEP *Supervisor Mode Execution Prevention*

SMT *Simultaneous Multithreading*

STIBP *Single Thread Indirect Branch Prediction*

SUT *System Under Test*

U $\rightarrow$ **K** *User* $\rightarrow$ *Kernel*

VA *Virtual Address*

Chapter 1

Introduction

1.1 Motivation

Retbleed [1] is a new vulnerability belonging to the family of *transient execution attacks*, that has started with *Meltdown* [2] and *Spectre* [3]. Retbleed executes arbitrary kernel memory from systems of various microarchitectures from both Intel and AMD, running the latest kernel and employing all available mitigations.[1]

Similar to Spectre V2 [3], which hijacks the speculative control flow after an indirect branch instruction, Retbleed hijacks the speculative control flow after a return instruction.

As Retbleed's BTI primitive causes a PF, the authors of Retbleed initially considered a mitigation, which detects these PFs and flushes the *Branch Prediction Unit* (BPU) using *Indirect Branch Prediction Barrier* (IBPB) [4]. We refer to this mitigation as IBPB-on-PF. While this mitigation has the advantage that the induced overhead is practically zero, Intel developers suspected that this mitigation is incomplete, and therefore, a more in-depth solution is required. To justify the more in-depth solution, knowing if IBPB-on-PF is indeed incomplete is required. If the PF-based mitigation can indeed be bypassed, we furthermore want to evaluate what the performance cost of the in-depth solution is.

1.2 Research Questions

The Retbleed mitigations from AMD [5] and Intel [6] impose a rather involved set of kernel patches. With such significant changes, it is difficult to estimate the overhead without running full system benchmarks. However, the overhead is likely to be rather significant.

IBPB-on-PF has the clear advantage that the overhead is practically zero as such PFs are never raised during normal execution.[2] It is argued that the mitigation is insufficient as the BTB can be poisoned without causing a PF. This leads us to our first research question:

Research Question 1 (RQ1). Because the IBPB-on-PF mitigation for Retbleed is said to be incomplete, can we verify that this is true by building a variant of Retbleed that does not rely on PFs?

[1] As of August 2022, mitigations have been released in kernel version 5.19 and been back-ported to various stable/LTS versions.

[2] We will talk about what differentiates Retbleed's PFs from regular PFs in subsection 2.1.3 and 2.4.4.

Next, we want to understand better the performance penalty induced by the patches featuring AMD and Intel's proposed mitigations. Most notably, what are the overheads, and how does the underlying microarchitecture impact them? Therefore, our second research question is:

Research Question 2 (RQ2). What is the performance cost for an in-depth Retbleed mitigation?

If we fail to show that IBPB-on-PF is insufficient, we will discuss if the in-depth patches leading to the evaluated overhead are needed.

1.3 Solution

To answer RQ1, we need to evaluate whether both of Retbleed's primitives can be executed without causing PFs. We use the two *Proof of Concept*s (PoCs) provided by Retbleed, namely Ret-BTI and CP-BTI [7]. As a way of suppressing PFs during BTI, we employ speculative BTI.

To answer our second research question, we need to get insights into the performance cost of the in-depth Retbleed mitigations. We benchmark systems of various microarchitectures with Retbleed mitigations enabled and disabled. To calculate the induced overhead, we compare the benchmark's scored.

1.4 Overview

After this short induction (chapter 1), the thesis proceeds as follows. Preliminary information is given in chapter 2. In chapter 3, we try to answer RQ1 by developing Spec CP-BTI. In chapter 4, we evaluate the overhead for the in-depth mitigation to answer RQ2. We present related work in chapter 5 and conclude the thesis in chapter 6.

Chapter 2

Background

In this chapter we provide some necessary background. We start with memory management and *Operating System* (OS)-related topics, namely memory addressing in subsection 2.1.1, the kernel (2.1.2), and the interplay of kernel and user space and how that can lead to PFs (2.1.3). We continue with caches, their vulnerabilities, and attacks on them in section 2.3, followed by speculative execution and a discussion of various transient execution attacks (2.4). Emphasis is put on RETBLEED (2.4.4).

2.1 Kernel, Address Space and Page Faults

2.1.1 Memory Addressing

An address space is an ordered set of contiguous, non-negative integer addresses ranging from 0 to some 2^{N-1}. *Virtual Addresses* (VAs) address the virtual memory, while *Physical Addresses* (PAs) address the physical memory. For VAs, N is equal to the bit width of the CPU.[1] PAs address physical memory, and therefore, the size of the PA space is given by the size of the system's memory. Addresses are mapped back and forth between VAs and PAs at a granularity of pages. A page is of fixed size, which is commonly 4096 B. This translation is done using page tables by the *Memory Management Unit* (MMU). When referring to "address" in this thesis, unless stated explicitly, we are talking about VAs.

2.1.2 Kernel

The kernel is the core part of each OS and facilitates the interactions between hardware and software components. It can be thought of as a special program controlling all hardware. When software and hardware need to communicate, the kernel serves as a middleman. The kernel has full read and write, but not execute, access to all the system's memory. The kernel is given this access via the *physmap*, a large, contiguous region in the kernel's address space that maps to all the system's physical memory [8].

For this thesis, we consider the Linux Kernel [9] as it is *Free and Open Source Software* (FOSS), meaning we can freely download, inspect and modify it. However, RETBLEED exploits features implemented in hardware, and therefore, other kernels are also applicable/vulnerable.

[1] Which is commonly 48 bit

2.1.3 Kernel/User Space and Page Faults

Virtual memory is split into kernel and user space to provide memory protection and enforce privilege boundaries. Kernel space is strictly reserved for running kernel-related processes, while applications are run in user space. The kernel needs full access to all hardware and software, so it runs in unrestricted privilege mode. A user, in contrast, has only full access to everything belonging to the user space.

When a process, whether running in user or kernel space, accesses an address, the MMU must translate it to the corresponding PA. The MMU walks the *page table* so check if a mapping from VA to PA exists. If a valid mapping is found, the MMU tries to access the data at the given PA. In the best case, the data is already in memory. If the data is not in memory or no mapping exists in the first place, a PF is raised. The kernel's PF handler gets invoked to resolve it. If the mapping exists, but the data is not present in memory, the PF handler loads the data, at a granularity of a page, into memory. If no mapping exists, the kernel must find a free page and create a mapping for it in the page table. Then the faulting instruction is re-executed.

Before creating a new mapping, it is always verified whether the subject[2] is allowed to access that particular memory address. If the subject references a page that he is not allowed to access, the type of the PF is *invalid*. The handler informs the subject about the privilege boundary breach by sending the `SIGSEGV` signal. An invalid PF should never be raised in the absence of bugs in the kernel/user process except if the user process has malicious intentions. When talking about PFs in the remainder of this thesis, we refer to invalid PFs.

2.2 Stack

The *stack* is part of the virtual address space of each process. Since the number of registers is limited, the stack is used to store data temporarily. Data is pushed and popped from the stack in a *Last In First Out* (LIFO) manner. The register `%rsp` appoints the top of the stack. It grows from high addresses to lower ones. Therefore, when pushing a value to the stack, the `%rsp` is decreased.

When a function is called, a *frame* is allocated on the stack. In that process, the function return address, and some bookkeeping data is pushed to the stack. Additionally, arguments may be passed to the function via the stack. During the function's execution, temporary data may be pushed to the stack. Once the function is completed and calls `return`, the return address is loaded from the stack, the whole stack frame gets deallocated, and execution continues at the return address.

2.3 Cache Structure

CPUs got faster and faster over time. While memory also got faster, the pace was slower than for CPUs. This gap widened over time. Caches try to compensate for that difference in speed. They are small banks of fast memory used by the CPU to store recently used data.

Caches store data in *lines* that hold a fixed number of bytes each. Multiple lines are combined in a *set*. A cache consists of multiple sets. The number of lines per set is described by the *associativity* of the cache. It ranges from *1-way associative*, in which case the set contains exactly one line, to *fully associative*, in which case there is only one set that contains all lines.

Memory is split into aligned chunks of the same size as the cache line. Each of these chunks maps to precisely one cache set, determined by certain bits of the memory address. To differentiate between multiple lines in the matching set, a *tag* is used.

[2] Which is a specific user or the kernel itself.

When the CPU wants to read data from a given address, it indexes the cache to see if the data is present. If a line with a matching tag is found in the right set, the CPU loads it. Otherwise, it must load the line containing the desired data into the cache and then to the CPU. If the cache is full, an *eviction policy* is used to determine the line to replace. To achieve the best possible performance, these policies are carefully crafted with the goal to evict the line which is accessed furthest in the future.

Modern processors employ multiple layers of caches, where each layer has different properties. Most importantly, there is a trade-off between speed and size. However, caches can also be *private* or *shared*, which describes if a cache is used by a single or multiple cores. The *inclusion policies* describe the interaction between multiple layers. When a cache is *inclusive*, all blocks it stores must also be present in all lower-level caches. An *exclusive* cache is one where a cached block is not allowed to be cached by any lower layer. If the cache is neither inclusive nor exclusive, then it is *non-inclusive*. In this case, the blocks from the upper-level cache may or may not be present in the lower-level cache.

Commodity CPUs consist of three levels, with L1 being the smallest but fastest one, and L3, also called *Last-Level Cache* (LLC), being large but slow. L2 lies somewhere in between. L3 is shared while the other two are private. All three levels are commonly inclusive.

2.3.1 Cache Attacks

The base principle of caches is that retrieving data from cache is much faster than loading it from memory. This property has been exploited using different *side-channel attacks* [10–13].

We will discuss Flush+Reload [10], as it is the one most relevant for Retbleed. It targets the L3, meaning the attacker and victim do not need to share the execution core. However, the attacker and the victim must have shared memory.

The attacker proceeds as follows: Using the `clflush` instruction, the attacker ensures that none of the memory lines are cached. Then, the attacker waits to give the victim time to do some memory operations. By reloading the memory pages and measuring the required time, the attacker knows if a page was cached and can infer if the victim has accessed it. This process is referred to as *convert channel*.

2.4 Speculative Execution

2.4.1 Pipelining

Code is a sequence of instructions. The CPU must do multiple steps to "execute" an instruction. At first, the CPU must load the instruction itself from memory. This process is called *fetching*. Before being able to *execute* it, it needs to figure out what kind of instruction it has just fetched and what its operands are. This process is called *decoding*. Instructions usually affect the system's state, like calculating some value and storing it someplace. We say an instruction *retires* when these effects are made visible to the system. Before retiring, the induced changes are only visible internally to the microarchitecture.

The aforementioned tasks are implemented as hardware units, and instructions "flow" from one stage to the next. Since these stages are independent circuits, the next instruction can already be fetched when the first one gets passed on to the decode stage. Since these stages are independent circuits, when one instruction is in the decode stage, the next instruction can already be fetched. An internal clock orchestrates these stages. This way, each stage knows when it receives new input

from the lower stage and when the results must be ready to be passed to the next stage. One says that such a CPU is pipelined.

Different microarchitectures have different types and numbers of stages. Intel Haswell has, for example, over 14 stages [14, 15].

2.4.2 Speculation

While pipelining increases throughput and hence the overall performance of the CPU, it happens that the CPU executes instructions that it should never have. Multiple cases can lead to that.

When a pipeline stage cannot complete within the given timeframe, which may happen when an operand needs to be fetched from memory, the whole pipeline comes naturally to a halt. One says that the pipeline is *stalling*. Since this is very costly, the CPU tries to prevent that by executing instructions *Out-of-Order* (OoO), which means that when an instruction is not yet ready to be executed, a subsequent instruction may already be executed. Obviously, the CPU needs to respect dependencies. For example, it cannot execute any instructions whose operand depends on the result of a not yet executed instruction. In addition, the system must give the illusion that the "execution" follows the *program order*. This is achieved by retiring the instruction in the program order.

OoO execution leads to speculation, as it is potentially unclear if an OoO executed instruction lies on the architectural path. Speculation happens if the execution of a branching instruction gets delayed while further downstream instructions are already executed. Alternatively, speculation can be caused if an instruction whose execution gets delayed, introduces an unrecoverable fault or an abort, once it is executed. Attacks exploiting the later case are generally referred to as *Meltdown* [2] vulnerabilities.

Speculatively executed instructions that should not have been executed and therefore need to be "undone" are called *transient instructions*.

2.4.3 Branch Prediction

Branching instructions are different from non-branching instruction as they affect the control flow. By potentially jumping to an arbitrary location in code, the CPU is primarily wrong when assuming that subsequent instruction is executed. Therefore, the CPU tries to predict the target for branching instruction. Misprediction of branch targets leads to much work needing to be undone, incurring substantial performance penalties. Therefore, good branch target prediction is essential for modern and performant CPUs.

This prediction is made by the BPU, a little hardware unit located in the instruction pipeline. When the CPU detects a branching instruction, which it does after decoding the instruction, it predicts where the branch leads.[3] The BPU consists of multiple predictors, and the one to use for a particular instruction is chosen based on the type of the instruction.

In the following, we will look at the two predictors relevant for this thesis. Due to the proprietary nature of CPUs, only very little is officially communicated about the BPU's internals, and researchers have to rely on reverse-engineering efforts [16].

Direct/Indirect Branch Prediction

Direct branches are branches where the destination address is given explicitly as an address or relative offset to a value stored in a register. *Indirect* branches provide a pointer to a memory

[3] Actually, there are prediction mechanisms to guess if a not yet decoded instruction could be a branching instruction, but that is not of relevance to us.

location, which gives the jump destination. Furthermore, branches are either *conditional* or *unconditional*. As the name indicates, the destination for conditional branches depends on a condition, while unconditional branches are always taken.

The prediction has two aspects. When a branch is conditional, the BPU needs to predict if the branch is taken or not, and where does the branch lead to, while for indirect branches, only the destination needs to be predicted. *Direct + unconditional* branches are the easiest to deal with as they do not require any prediction. However, predicting *indirect + conditional* requires two predictions; is the branch taken or not, and if it is taken, what is its destination? They have the highest risk of misprediction. Happily, they do not exist in the x86_64 *Instruction Set Architecture* (ISA).

The direct/indirect branch predictors works by the assumption that when a branch is reencountered, it behaves similarly to the previous encounter. A cache-like structure called BTB stores targets of previously encountered branches. When the predictor has to make a prediction for a particular branching instruction, the BTB is indexed to see if the branch has previously been encountered. If there is an entry for this branch, was it taken, and if so, where did it go?

The BTB is similarly structured to caches in the way that they are also composed of sets. Similar to data in a cache, a branch always maps to exactly one set. A set is composed of multiple possible targets, which allows for predicting branches with more complex taken/not-taken patterns and multiple destinations. The indexing of the BTB is microarchitecture dependent but generally done using machine states like the current *Program Counter* (PC). In addition, auxiliary data structures may be used. Some Intel microarchitectures use the branch history, condensed in a hash-like format, stored in the *Branch History Buffer* (BHB), for the indexing. It allows for efficient representation of the N last encountered branches. For Intel Haswell, for example, $N = 29$ [3, 16].

Unlike caches, the ways of the BTB have much smaller tags, which results in *collisions*. A collision is when two different branching instructions map to the same BTB set. It has been shown that for Intel Haswell, two branches collide if bits 6–11 are equivalent [1]. On AMD, it is a bit more complicated. Two addresses collide if the address bits are permuted according to a specific schema. While the collision works for Intel across privilege domains, for AMD, it only works for Zen 1 and Zen 2. For Zen 3, Wikner and Razavi could not detect collisions across privilege boundaries [1].

Function Return Prediction

The *function return predictor*, which may also be referred to as *return target predictor*, is used for return instructions. It works by assuming that a function always returns to the place it was called from. It uses a stack-like cache called *Return Stack Buffer* (RSB) to support multiple nested function calls. When encountering a call, the expected return address is pushed to the RSB.[4] Once we return, the top address of the RSB is popped, and we speculatively execute the path starting at the retrieved address.

The capacity of the RSB is limited, with, for example, 16 entries for Haswell or 22 for Ice Lake [15]. In case of an overflow, the RSB wraps around and overwrites the oldest element. About the behavior of an underflow is not much known. Wikner and Razavi have shown that certain microarchitectures fall back on using the BTB [1]. We will look more closely at this behavior in section 2.4.4.

[4]For the x86_64 ISA, a function residing at PC is expected to return to $PC + 4$.

2.4.4 Spectre Attacks

While speculatively executed instructions do not retire and hence, they do not affect the architectural state, they leave traces in the microarchitectural state. Using *side-channel attacks*, these trances can be used to infer information about the operants of the speculatively executed instructions. Since the publication of *Spectre* [3] in 2018, new versions of Spectre, more generally referred to as *transient execution attacks*, were discovered.

By influencing the *transient code*, attackers can control what data to leak. That is not just limited to data of the current user, but a wide variety of attacks work across privilege boundaries. To achieve the desired speculation, attackers often manipulate the BPU in carefully selected ways [1–3].

For a Spectre attack to be successful, three things are needed:

1. **Speculation Primitive.** Causes the speculation and makes the disclosure gadget to get executed speculatively.
2. **Disclosure Gadget.** Is executed speculatively and causes the leakage.
3. **Convert Channel.** Is used to read out the microarchitectural traces left over by the transient execution.

Mitigating Spectre attacks is non-trivial as speculation is a core principle of modern CPUs. Moreover, since there is a whole range of Spectre attacks, there is also no single patch to fix them all, but rather for each, an individual mitigation has to be found. Spectre attacks can be mitigated in three different ways: [1]

- *Isolation*: Privilege boundaries are enforced such that an attacker cannot influence a victim process acting in a different privilege domain. ⇒ No leakage of sensitive information.
- *Prevention of speculation*: ⇒ Disclosure gadget cannot be executed speculatively.
- *Prevention of convert channel*: ⇒ Stop leakage of microarchitectural traces.
  - This is not feasible for the Linux Kernel.

Next, we will look at a few relevant transient execution attacks. We start by looking at *Spectre V1*, which should serve as a simple introductory example. As *Spectre V2* shares some similarities with Retbleed, we will discuss that after. *SpectreRSB* is relevant because we use it to craft our speculative version of Retbleed. Lastly, we will have an in-depth view on Retbleed, such that we have all the knowledge required to build the speculative version on top of it.

For each of these vulnerabilities, we will quickly discuss what they are, how they work, what makes them feasible, and possibly how they are mitigated.

Spectre V1

Spectre Variant 1 [3] is also referred to as *Spectre Bound Check Bypass (BCB)* and targets conditional branching instruction. When evaluating the condition takes time, the CPU guesses which branch is more likely to be taken and speculatively executes it. The attacker can steer the speculative control flow by mistraining the branch predictor. For example, when placing the conditional inside a for loop where the branch is taken each iteration except for the last one, the branch predictor would "learn" that the branch is always taken, and therefore, mispredict in the last iteration.

To make the speculation work reliable, the *speculation window*, which is the timeframe in which instructions are executed speculatively, has to be large. A large speculation window is achieved by ensuring that evaluating the condition takes a reasonable amount of time. This can be caused, for example, by an uncached memory access.

If a disclosure gadget, as in Listing 2.1, is executed speculatively, data outside the array is accessed. The value can then be inferred using a *side-channel*.

```
1 for (int i = 0; i <= array_size; i++) {
2     // Make sure that array_size is uncached
3     doUncache(array_size);
4
5     // Misprdicts for i = array_size
6     if (i < array_size) {
7         y = array[i];
8     }
9 }
```

Listing 2.1: The for loop trains the BPU to make it belief that the branch induced by the if statement in line 6 is always taken. In the last iteration, this causes the BPU to misspredict. To cause the speculation window, `array_size` is uncached prior to the branch. Using a suitable convert channel, which is not depicted in this snipped, the value of `y` can be leaked.

To mitigate Spectre V1, one can use the `lfence` instruction or create data dependencies, to prevent the speculative out-of-bounds access [17]. Compilers, such as GCC, have been patched to be able to do that automatically [18]. Therefore, by recompilation, Spectre V1 can be mitigated.

This attack is often employed with array out-of-bounds accesses, as in our example. Nevertheless, even if called BCB, it is not limited to them and can be used to exploit any kind of conditional branches.

Spectre V2

Spectre Variant 2 [3] is also referred to as *Spectre BTI* and takes advantage of speculative execution of indirect branches. As seen in section 2.4.3, indirect branches are predicted based on the BTB. An attacker can hijack an indirect branch by injecting a branch target into the BTB, which the predictor uses to predict the destination of the victim branch. This injection causes the victim branch to speculatively executed the injected target, where the disclosure gadget of the attacker resides. As the BTB is not flushed on transitions of privilege domains, this BTI works across privilege boundaries.

To hijack an indirect branch instruction residing at `VICTIM`, the attacker looks for a colliding address `VICTIM'`. By jumping from `VICTIM'` to `GADGET`, a corresponding entry in the BTB is created. When the victim encounters the branch at `VICTIM`, the BTB will use the injected target for the prediction. This steers the speculative control flow to `GADGET`.

Intel proposed *Indirect Branch Restricted Speculation* (IBRS) [19] as a possible mitigation for Spectre V2. IBRS prevents the predictor from using branch prediction resolutions from lower privilege levels on higher ones. This measure prevents the hijack of a victim process across privilege boundaries. In the end, a different mitigation, called *retpoline* [20, 21], was used due to its lower overhead [22]. Retpoline prevents speculative execution from using the indirect branch predictor.

SpectreRSB

SpectreRSB [23] is another Spectre attack. It targets return instructions, and by exploiting the RSB, it can create a speculation window where arbitrary code can be executed. However, it is only

feasible if the attacker can modify the executed code.

When encountering a return instruction, the return instruction predictor thinks that the return leads back to where the function was called from. By modifying the stack, an attacker can change the architectural return address. The discrepancy between the actual and expected return address causes the return instruction predictor to mispredict, which creates a speculation window.

SpectreRSB works quite reliably and the speculations windows created are rather large. In Listing 3.2, we have used this method to cause speculation.

Retbleed

Retbleed [1] is a transient execution attack, sharing many similarities with Spectre V2. It also hijacks branches by poisoning BTB, but in contrast to Spectre V2, it targets return instructions. While SpectreRSB [23], Ret2Spec [24], and Spring [25] all are return-based Spectre attacks, they target the RSB, while Retbleed targets the BTB.

By reverse-engineering aspects of the BPU, Wikner and Razavi gained new insights on the behavior of the BPU under certain microarchitectural-dependant conditions. These findings have led them the creation of the Spectre exploit Retbleed, that leaks arbitrary kernel memory at a rate of 219 B/s for Intel Coffee Lake and 3.9 kB/s for AMD Zen 2.

Retbleed relies on two exploitation primitives, which must be present in a susceptible system; exploitable return instructions and BTI on kernel memory. We will discuss what both of them mean before wrapping up with the end-to-end Retbleed exploit.

Exploitable Return Instructions. Using their reverse-engineering, they showed that in some instances, the RSB-based return instruction predictor falls back to using the BTB. This fallback is referred to as RSB-to-BTB. When the fallback happens, return instruction behave as indirect branches. For Intel microarchitectures, like Haswell and Coffee Lake, this happens for RSB underflows. AMD falls back to using the BTB without the RSB needing to underflow. Instead, they observe the fallback whenever the return instruction's address collides with the address of a previously encountered indirect branch.

These findings mean that for Intel, all return instructions which follow a *sufficiently-deep call stack* (i.e., one such that the RSB underflows) can be hijacked, while for AMD, any return instruction can be hijacked.

Ret-BTI is a PoC, released as part of Retbleed [7], that demonstrates if a system is susceptible to this primitive. The PoC was released for both, AMD and Intel.

BTI on Kernel Returns. After the predictor falls back from using the RSB to the BTB, the attacker can now poison the BTB such that a return instruction gets hijacked, and the attacker's disclosure gadget gets executed speculatively. While the BTI allows for the injection of arbitrary targets, *Supervisor Mode Access Prevention* (SMAP) and *Supervisor Mode Execution Prevention* (SMEP) limit the target to be in kernel space as they prevent the kernel from jumping and executing code in user space. Therefore, the disclosure gadget has to reside in kernel space. But that is not an issue, as the BTB can be poisoned by utilizing colliding branch sources, similarly as with Spectre V2. Wikner and Razavi showed that branches that cross privilege boundaries, and hence, cause a PF, are still taken up by the BTB.

The CP-BTI PoC[5] [7] shows if a system is vulnerable to the second primitive. A system where both PoCs works successfully, is susceptible by Retbleed.[6]

[5] Which is also referred to as *User* → *Kernel* (U → K) PoC.

[6] As the CP-BTI PoC is based on Ret-BTI, it is sufficient if the CP-BTI PoC works.

Retbleed End-to-End Attack. The actual RETBLEED attack can leak arbitrary privileged memory at a rate of 219 B/s for Intel Coffee Lake and 3.9 kB/s for AMD Zen 2. All systems susceptible to both mentioned primitives are vulnerable to RETBLEED. AMD Zen 1, Zen 1+ and Zen 2, as well as Intel Kaby Lake and Coffee Lake, are among the vulnerable microarchitectures [1].

To make this attack work, an attacker must overcome multiple challenges. Firstly, vulnerable and exploitable return instructions must be detected and identified. Next, a suitable disclosure gadget must be found, and the branch history of the victim return must be reconstructed. During the attack, *Kernel Address Space Layout Randomization* (KASLR) needs to be broken, to get the address of the disclosure gadget and relevant parts of the convert channel. Before the BTI can be done, the branch history needs to be setup. Only then the victim return can be executed using a system call.

Mitigation We have already introduced IBPB-on-PF as a possible mitigation in section 1.1 and 1.2. While it enforces isolation by preventing BTI from user to kernel space, it is said to be incomplete.

Enhanced Indirect Branch Restricted Speculation (eIBRS) [19] was shown to mitigate RETBLEED. Therefore, more recent microarchitectures like Coffe Lake Refresh and Alder Lake are secure. On older microarchitectures, where RSB-to-BTB can be observed and which do not support eIBRS, IBRS is enabled [1, 6].

While Intel takes the route of isolation, AMD prevents the speculation from happening altogether. Their mitigation, called *jmp2ret*, replaces all return instructions in the kernel with jumps to a return thunk. An *untrain* procedure secures the last remaining return instruction [1, 5, 26]. Some more details on the mitigation is given in section 3.4, where we discuss mitigation overheads.

Chapter 3

Speculative Retbleed

3.1 Overview

Retbleed causes the function return predictor to stop using the RSB but to fall back to the BTB. By poisoning the BTB, an attacker can steer the speculative control flow after a return instruction executed in kernel space. The speculative execution leads to microarchitectural traces, which are inferred using a side-channel attack.

Why *Page Faults*? The IBPB-on-PF mitigation relies on PF, raised during the attack. To understand why this mitigation may be possibly incomplete, we need to understand why PFs are raised in the first place.

The attacker hijacks the kernel branch by injecting a BTB entry which gets used by the return instruction predictor. It will use the destination of the injected entry as the most likely destination of the branch. The source of the malicious branch is selected to collide with the address of the return instruction that the attacker wants to hijack. The destination is the address of the disclosure gadget. Due to SMAP and SMEP, the kernel cannot execute arbitrary user space code, and therefore, the disclosure gadget itself must reside in kernel space. This results in a poisonous branch, where the source resides in the attacker's user space, and the destination is in the kernel space. I.e., the attacker must do BTI across privilege boundaries. As a user is not allowed to jump to an arbitrary location in kernel space, a PF is raised, and the PF handler takes over to inform the user about the privilege boundary breach.

The Plan. As stated in RQ1, we want to create a speculation primitive which does the described BTI without causing a PF. Our idea is to do speculative BTI, meaning that the poisonous branch of the speculation primitive is itself executed speculatively. To the best of our knowledge, this has never been explicitly shown before. Therefore, we need to verify its operability first. We expect from speculative BTI that it poisons the BTB, however, without raising any PFs, even for branches crossing privilege boundaries. This leads us to the development of a speculative version of the CP-BTI PoC. We will discuss the creation of the PoCs for both Intel and AMD in the next section.

3.2 Implementation

We use the PoCs provided by Retbleed as a basis for our PoCs[1] [7]. Before we develop a speculative version of the CP-BTI PoCs, we want to verify that speculative BTI works in the same privilege domain. For that, we modify the Ret-BTI PoCs to create a version of it where the BTI is done speculatively. We will refer to it as Spec Ret-BTI. If we succeed, we proceed by creating a speculative version of CP-BTI to see if the speculative BTI works across privilege domains, as it does with non-speculative BTI.

Retbleed causes a BTB-to-BTB fallback of the function return predictor. AMD and Intel microarchitectures have shown this behavior, but as seen in section 2.4.4, it is caused by different events. For Intel, this is achieved by underflowing the RSB. Which happens if too many return instructions are encountered in a row. We refer to the mechanism we use to achieve that in our PoC as a *return cycle*. Besides underflowing the RSB, the return cycle has the additional purpose of creating a "normalized" branch history, meaning that the BHB is set into a known and easily reproducible state. Normalizing the history is important since the BTB is indexed using the BHB. The BPU will not use the injected BTB entry, in case the history diverges too much from the one present during the poisoning.

To create the return cycle, a memory address holding a return instruction is repeatedly pushed to the stack. An initial return instruction starts the recursive cycle. After all addresses of the return instruction are popped from the stack, the control flow returns to the address one the stack, which was pushed prior to creating the cycle. Listing 3.1 shows the used method. Alternatively, a recursive function call can also create the return cycle. The length of the return cycle is microarchitecture dependent, but 28 cycles are optimal for Coffee Lake and 29 for Coffee Lake Refresh [1].

```
// Store return instruction to memory location RET_PATH
memcpy((u8*)RET_PATH, "\xc3", 1);
asm(
    // Address to which to return after the cycle
    "pushq %[cycle_dst]\n\t"
    // Create return cycle of length 30
    ".rept 30\n\t"
        "push $RET_PATH\n\t"
    ".endr\n\t"
    // Start the cycle
    "ret\n\t"
    :: [cycle_dst]"r"(BR_SRC1) :
);
```

Listing 3.1: Creation of a return cycle. The address cycle_dst, to which the control flow return after the return cycle, is pushed to the stack first. Next, the return cycle is created by repeatedly pushing the address RET_PATH, where a return instruction is stored, to the stack. An initial return instruction starts the cycle.

AMD CPUs exhibit the RSB-to-BTB fallback for return instruction in case they collide with the address of a previously encountered indirect branch. Also, the cycle is also not required for setting up the branch history, as the BTB does not seem to be indexed using any kind of branch history [1]. Instead, the BTB is indexed using the start and end addresses of the branch, which can be thought of as a "basic block".

Preview. In the subsequent sections, we first discuss the development of Spec Ret-BTI, followed by Spec CP-BTI. For both, we first provide more information on the non-speculative version of

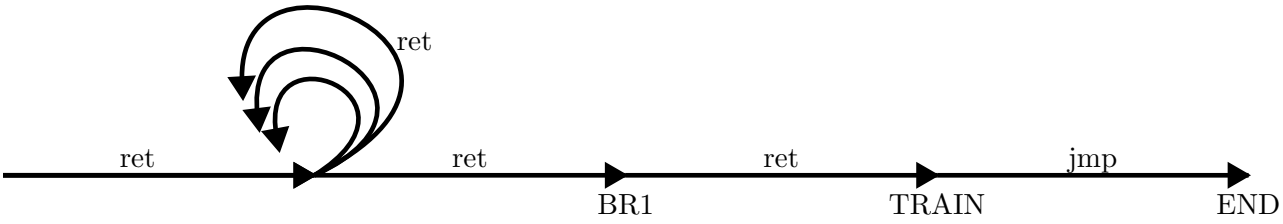
[1] We will consider a slightly modified version of the Retbleed PoCs where we have done some simplifications.

the PoCs. We always first describe the PoCs for Intel, followed by the description for AMD. As common with Spectre PoCs, the PoCs are split into two phases: the *training phase*, where the BTI is done, and the *speculation phase*, where a victim branch gets hijacked. We do not describe the working and implementation of the convert channel, as we use the flush+reload-based [10] one provided by the PoCs without modification. Whenever a return cycle is employed, it serves to underflow the RSB and normalize the branch history, as discussed in section 3.2.

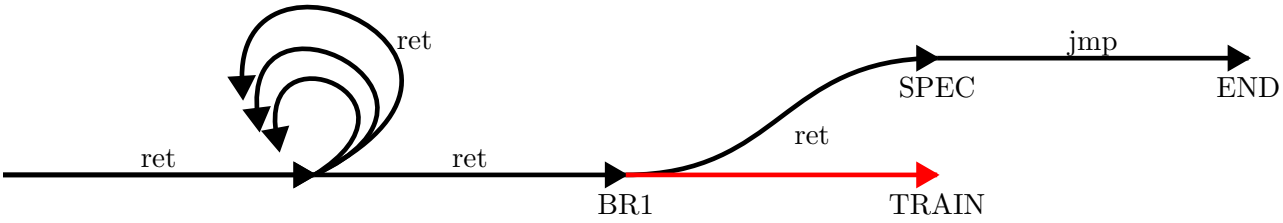
3.2.1 Speculative Ret-BTI

We will discuss the design of the Spec RET-BTI PoC. This PoC aims to verify that speculative BTI works in the same privilege domain. However, before working on Spec RET-BTI, we will discuss how the plain RET-BTI PoC works.

Ret-BTI in detail. After the return cycle, spinning on a particular memory location, we get to `BR1`. Here, the speculation primitive, a return instruction, is located. This return brings us to the disclosure gadget, stored at `TRAIN`, during the training phase. It also does the BTI. In the speculation phase, since the PC is the same (`BR1` in both cases) and thanks to the return cycle, the histories are equivalent, the function return predictor, which has fallen back to using the BTB, predicts erroneously that the destination is `TRAIN` instead of `SPEC`. The misprediction causes the disclosure gadget to get executed. This proceeding is depicted in Figure 3.1.



(a) In the training phase, the branch from `BR1` to `TRAIN` is taken up by the BTB.



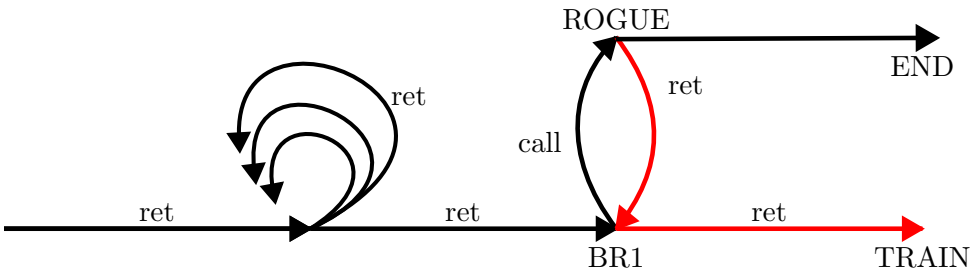
(b) When the control flow reaches `BR1` in the speculation phase, the PC and the branch history are equivalent to the ones of the training phase. Therefore, the return instruction predictor will use the injected BTB entry, steering the speculative control flow to `TRAIN`. The speculation is indicated in red.

Figure 3.1: Control flow of the RET-BTI PoC for Intel. The training phase, poisoning the BPU, is depicted in (a). The BTI causes the victim to mispredict into the disclosure gadget, as visible in (b). A return cycle is used in both phases to underflow the RSB and normalize branch history.

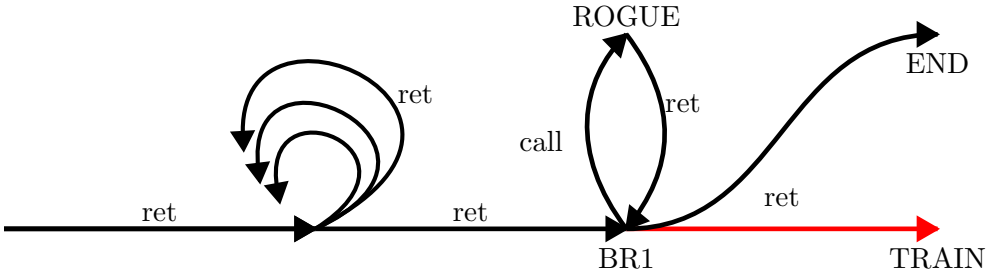
As discussed in section 3.2, no return cycle is needed for AMD as an address collision of the return instruction with a previously encountered indirect branch is sufficient for causing the RSB fallback. Since the BTB is not indexed using the branch history, it is also unnecessary to normalize it. The primitives of Intel are adapted as follows. The return cycles are removed for both phases.

For the training phase, the poisonous return instruction is replaced by an indirect jump. The source and target remain the same. Namely, it goes from `BR1` to `TRAIN`. These changes are already sufficient to poison the BTB and hijack any return instruction located at `BR2`, which collides with `BR1`.

Spec Ret-BTI. For the speculative version, we want that the return from `BR1` to `TRAIN` is not executed architecturally but speculatively. As in the non-speculative version, the return cycle brings us to `BR1`. To cause a speculation window during which the primitive can be executed, we exploit the return target predictor using SpectreRSB [23], as explained in section 2.4.4. From `BR1`, we call the "rogue" function located at `ROGUE`. That function changes the architectural return address such that it returns to `END` instead of `BR1`.



(a) Depicts the training phase. A rogue function, located at `ROGUE`, causes a speculation window. It allows the speculative execution of the poisonous return from `BR1` to `TRAIN`.



(b) The rogue function is also executed during the speculation phase, to ensure that histories are equivalent. However, in contrast to the training phase, no speculation window is created. After returning from the rogue function, the return instruction is mispredicted, using the injected entry pointing to `TRAIN`.

Figure 3.2: Control flow of the Spec RET-BTI PoC for Intel. During the training phase, depicted in (a), a speculatively executed return poisons the BTB. This leads to the hijacking of a return instruction, as visible in (b). Speculatively executed branches are indicated in red, while the architectural branches are black.

To use SpectreRSB, the rogue function increment the `%rsp` by 8 to let it skip the actual return address and point it to a location that we have pushed to the stack before calling `ROGUE`. As discussed in section 2.4.3, the return target predictor predicts that the function returns to `BR1` and, therefore, speculatively executes the return instruction. This return instruction does the BTI. To make the speculation window large enough for the return to execute fully, we use the `clflush` instruction to remove the actual return address, appointed by the `%sp`, from all cache layers. This way, the CPU has to fetch it from memory, incurring a more significant delay.

Three things must be fulfilled to make the indirect jump at `BR1` speculatively take the branch

to `TRAIN` during the speculation phase. First and foremost, the branch to `TRAIN` must have been injected, which we have done in the training phase. Secondly, the PC of the return instruction that we want to hijack must be the same as during training. Lastly, the branch history at the victim and attacker branch must be the same.

To ensure matching histories, we must retain the rogue function during the training phase. However, it must not cause any speculation or introduce new branches. We implemented that part of the speculation primitive using the `cmove` instruction. It allows us to modify the stack depending on whether we are in the training or speculation phase, which is indicated by the state of some register. The rogue function is displayed in Listing 3.2, and the control flow of both, training and speculation phase, are depicted in Figure 3.2.

When returning to `BR1` from the rogue function, the PC and BHB are equal to the ones of the training phase. Therefore, the predictor will use the malicious entry to guide the speculative control flow of the return instruction to the disclosure gadget at `TRAIN`.

```
asm(
    ".align 0x80000\n\t"
    "rogue_spec_dst:\n\t"
        "callq rogue_gadg_dst\n\t"
        // Training: execute following code speculatively
        // Misspredict: execute following code architectually
        "jmp *%r9\n\t"

    "rogue_gadg_dst:\n\t"
        // If %rsi = 1: add 8 to rsp => cause speculation
        // If %rsi = 0: do nothing
        "lfence\n\t"
        "movq %rsp, %rdx\n\t"
        "addq $0x8, %rdx\n\t"
        "cmp $1, %rsi\n\t"
        "cmoveq %rdx, %rsp\n\t"
        "clflush (%rsp)\n\t"
        "ret\n\t"
    "rogue_spec_dst_end:\n\t"
);
```

Listing 3.2: Rogue function, causing a speculation window by employing SpectreRSB. This function must be executed in both phases to maintain a consistent branch history. Depending on the state of %rsi the cmove instruction increments the %rsp by 8, skipping the actual return address. The speculation window is enlarged by using the clflush instruction.

Adapting Ret-BTI to Spec Ret-BTI is more straightforward for AMD than for Intel, as the branch history is irrelevant for indexing into the BTB. While the speculation phase of Spec Ret-BTI is equivalent to the one of Ret-BTI, a rogue function is added to the training phase right before the poisonous indirect branch. This rogue function causes unconditional speculation using SpectreRSB [23].

We comment on the results of these PoCs later in section 3.3 and 3.4.

3.2.2 Speculative CP-BTI

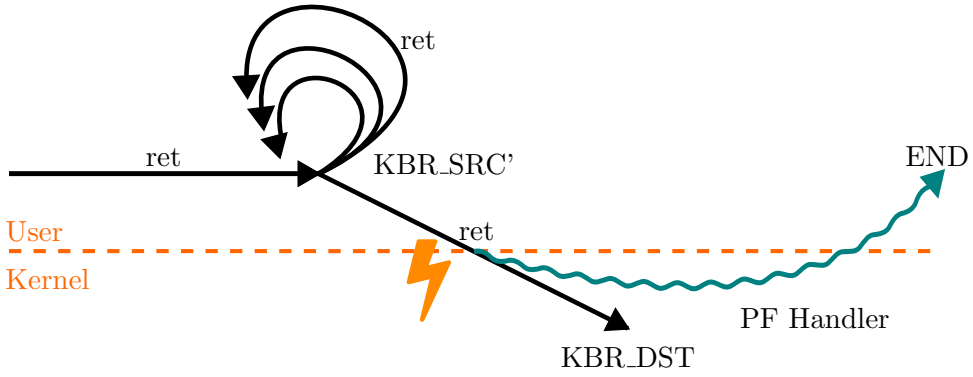
The development of the speculative version of CP-BTI is of main interest to us. It shows if Spec BTI works across privilege boundaries and, therefore, demonstrates if Retbleed's primitives can be implemented without raining any PFs.

These PoCs consist of a user space program and a kernel module. The user space program is the attacker who poisons the BTB across privilege boundaries to hijack a return instruction executed

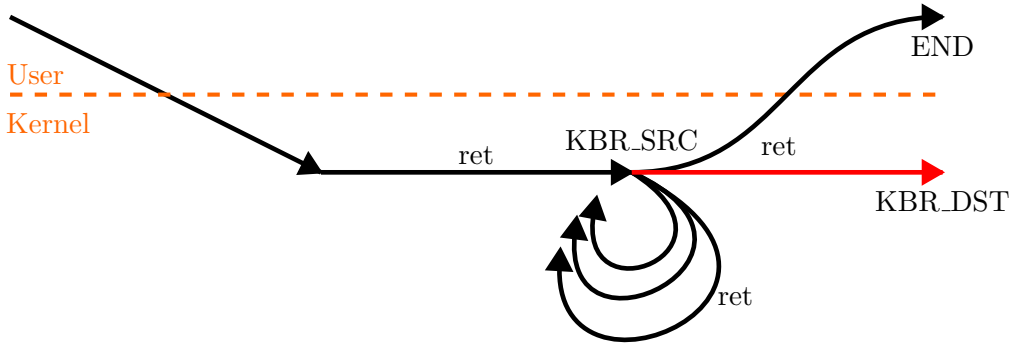
by the kernel. The kernel module provides the speculation primitive and disclosure gadget at predetermined and optimal[2] locations.

Before discussing the speculative variant, we will look at the plain CP-BTI PoC to set a foundation on which we can build.

CP-BTI in detail. A return cycle is set up to spin on `KBR_SRC'`. The final return leads to `KBR_DST`. `KBR_DST` is the location of the disclosure gadget stored in the kernel space. Since a jump to an arbitrary location in kernel space is prohibited, the PF handler takes over, raises a PF, and informs the user. Even if the branch to the disclosure gadget was unsuccessful, it has still influenced the BTB by injecting an entry. Figure 3.3a shows the training phase which we have just described. The speculation phase, which we will discuss next, is depicted in Figure 3.3b.



(a) Depicts the training phase. While a jump from userspace to kernel space is forbidden and caught by the PF handler, indicated in blue, it is still recorded by the BTB. This way, a branch from `KBR_SRC'` to `KBR_DST` is injected.



(b) Depicts the speculation phase. The injected branch, whose source `KBR_SRC'` is selected to collide with `KBR_SRC`, leads to a misprediction. It works even when injected from a different privilege domain. The speculation is indicated in red.

Figure 3.3: Control flow of CP-BTI for Intel. A jump to an arbitrary kernel address from userspace results in a PF, as shown in (a). However, it is still taken up by the BTB and can lead to misprediction across privilege boundaries, as shown in (b). `KBR_SRC'` is selected such that it collides with `KBR_SRC`.

To make use of the poisoned BTB entry, control is handed over to the kernel. It executes

[2] In the way that we can easily find a colliding branch.

the return cycle located at KBR_SRC. Similarly, as with the RET-BTI PoC for AMD, the source of the victim and attacker branch are different, as they lie in different address spaces. As with the mentioned AMD PoC, KBR_SRC' is selected so that it collides with KBR_SRC. Therefore, as the PCs collide and the histories are equivalent[3], the BPU will use the malicious entry for its prediction, guiding the speculative control flow to KBR_DST.

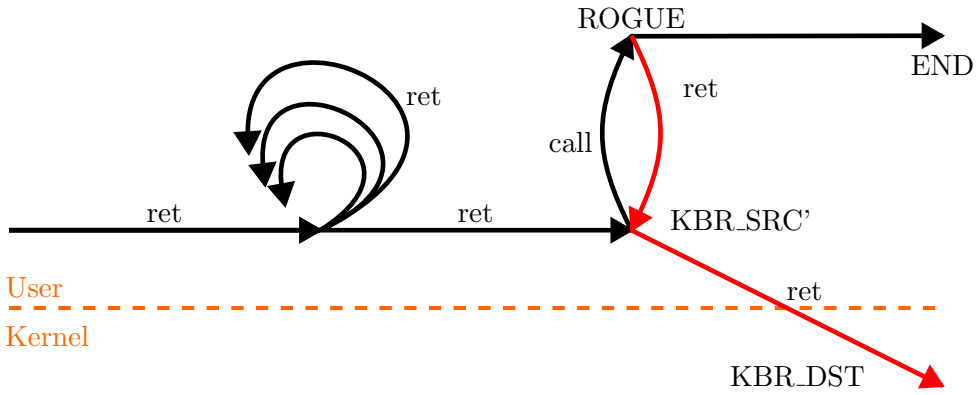
The version for AMD only differs in the way that no return cycles are used.

Spec CP-BTI. Similarly, as for Spec RET-BTI, we want to modify the CP-BTI PoC such that the BTI is done speculatively. In contrast to the non-speculative version, the final return from the return cycle does not do the injection itself but brings us to KBR_SRC'. KBR_SRC' is selected to collide with KBR_SRC, which is the address of the victim branch. As the branch from KBR_SRC' to KBR_DST should be executed speculatively, we create a speculation window using the rogue function located at ROGUE, as we did for Spec RET-BTI. This function causes the speculative control flow to be steered to KBR_SRC'. From here, a return instruction injects the BTB entry with a target of KBR_DST. While this branch is invalid as it crosses privilege boundaries, since executed speculatively, **no PF is raised** while the injection still works. The rogue function is equivalent to the one for the Spec RET-BTI PoC for AMD, where the speculation window is created unconditionally. This training process is visualized in Figure 3.4a.

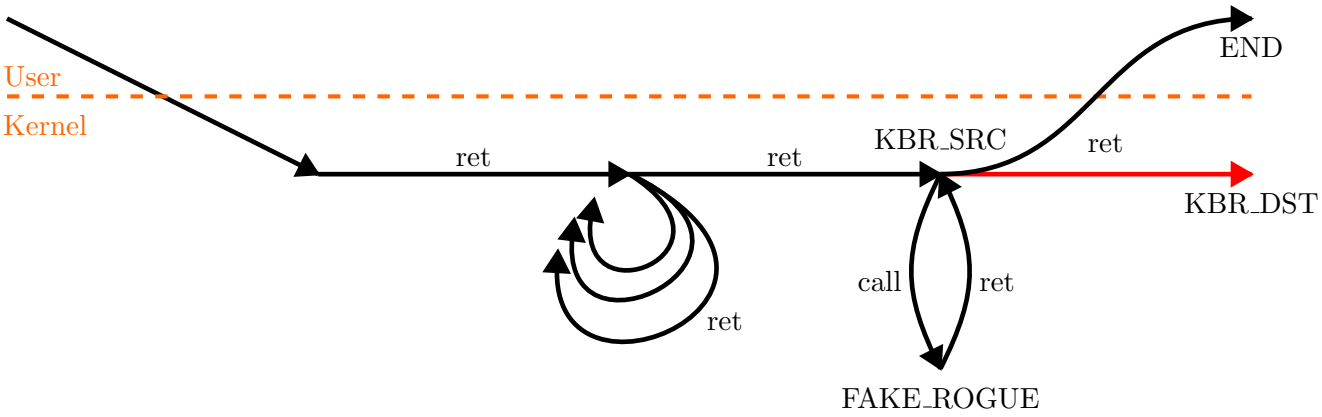
After seeing how the BTB can be poisoned from user space without causing any PFs, we will discuss how the injected BTB entry can impact the control flow of the kernel module. After switching to the kernel module and executing the return cycle, KBR_DST is reached. To make the branch history similar to the one of the training phase, we need to mimic the branches introduced by the rogue function. This is done by the "dummy" function at FAKE_ROGUE, which does nothing but returning back to KBR_SRC. Here, we encounter the victim return. Since the return target predictor has fallen back to the BTB, the PC collides with the source of the injected branch, and the history is similar to the one during training, the injected BTB entry will be used for the branch prediction resolution. Therefore, the disclosure gadget stored at KBR_DST is executed speculatively. The training phase is shown in Figure 3.4b.

We have developed a Spec CP-BTI version for AMD too. It is equivalent to the version of Intel, with the return cycles and the fake rogue function removed.

[3]Even if the return cycle spins on colliding addresses, it hard to say if the histories are actually equivalent or just "similar enough" to make the prediction work.



(a) Depicts the training phase. The rogue function at `ROGUE` causes a speculation window. It allows for the speculative execution of the poisonous return from `KBR_SRC'` to `KBR_DST`. While this branch crosses privilege boundaries, no PF is raised due to being executed speculatively.



(b) To make the branch history of the speculation phase similar to the one of the training phase, a "dummy" function mimics the branches introduced by the rogue function. The injected BTB entry is used to predict the target of the subsequent return, causing a speculative execution of the code residing at `KBR_DST`.

Figure 3.4: Control flow of Spec CP-BTI for Intel. In the training phase, shown in (a), speculative BTI is employed to hijack a return instruction executed in kernel space, shown in (b). As the injection is done speculatively, no PF is raised. Address `KBR_SRC'` is selected such that it collides with `KBR_SRC`. Architectural branches are black, while speculatively executed ones are drawn in red.

3.3 Evaluation

We have developed a speculative version of both Ret-BTI and CP-BTI. The Spec Ret-BTI PoCs for AMD and Intel, and Spec CP-BTI for Intel are working. For all PoCs for Intel, we get reliable signals and have not observed any wrong outputs. While we get some signal for the Spec Ret-BTI PoC for AMD, the signal is not very stable. For all working PoCs, we have verified that the source of speculation comes from the desired location. This process is further described in section A.2. The CP-BTI PoC for AMD is not working.

The implementation details have been discussed in the previous section 3.2. In this section, we want to evaluate their performance by comparing them to their non-speculative counterparts. The PoC can return one of the following outputs:

- The output can be true, meaning that the secret was leaked successfully
- The output can be false, meaning that the "leaked" secret does not conform to the actual value of the secret
- No output is generated as no value could be leaked

Method. Running a PoC once executes the attack N times in series. Since we have observed that the chosen N can have an impact on the performance, we will experiment with various N in the range of 10 to 20000. To check for consistency, we repeat the experiment 10 times for each N. The results are aggregated over the runs as follows; The number of false outputs was summed up. The mean and the standard deviation of true outputs is calculated. In addition, the standard derivation was normalized to N.

Results. Table 3.1 and 3.2 show the results for Ret-BTI and Spec Ret-BTI for Intel. For Ret-BTI, the percentage of correct outputs is mostly constant over N, with a mean of 29.53 %. The mean of the normalized standard deviation is 0.0216.

For Spec Ret-BTI, the mean success rate is 67.82 %, which is ∼38 percentage point higher than for the non-speculative version. However, the speculative version is less stable, which is confirmed by the higher mean of the normalized standard derivation, which is 0.0777.

Table 3.1: Ret-BTI PoC for Intel. Run for various N, accumulated over 10 executions.

N	10	50	100	500	1000	5000	10000	15000	20000
# Wrong	0	0	0	0	0	0	0	0	0
Correct in %	27.00	36.00	34.40	28.06	27.62	28.39	27.51	28.84	28.68
Correct Std.	0.46	1.41	3.01	5.51	34.40	43.17	372.95	81.69	158.71
Normalized Std.	0.04583	0.02828	0.03007	0.01103	0.03440	0.00863	0.03730	0.00545	0.00794

Table 3.2: Spec Ret-BTI PoC for Intel. Run for various N, accumulated over 10 executions.

N	10	50	100	500	1000	5000	10000	15000	20000
# Wrong	0	0	0	0	0	0	0	0	0
Correct in %	56.00	70.80	73.60	66.34	68.68	69.69	62.65	69.43	70.43
Correct Std.	1.80	3.47	8.11	46.15	55.89	169.56	1269.06	849.60	762.67
Normalized Std.	0.18000	0.06940	0.08114	0.09230	0.05589	0.03391	0.12691	0.05664	0.03813

The results of the CP-BTI PoCs for Intel are listed in Table 3.1 and 3.2. For the non-speculative CP-BTI PoC, the percentage of correct answers starts relatively low for a small N, and increases with increasing N. After a peak at $N = 20000$, it decreases again. The mean percentage of correct answers is 12.23 %, and the mean normalized standard deviation is 0.1665. For the speculative version, the success percentage increases with increasing N. The highest reached percentage is 88.77 %, which we got for the highest tested N. The mean success rate is with 60.07 % approximately 5 times as large as for the non-speculative case. The output is much less stable, which is confirmed by the mean normalized standard derivation, which is with 0.2839 twice as large as for the non-speculative PoC.

Table 3.3: CP-BTI PoC for Intel. Run for various N, accumulated over 10 executions.

N	10	100	1000	10000	15000	20000	40000	100000	250000
# Wrong	0	0	0	0	0	0	0	0	0
Correct in %	3.00	11.00	16.36	18.28	13.23	21.48	9.74	13.77	12.01
Correct Std.	0.90	18.45	200.31	1923.31	2980.68	4044.88	7659.87	18380.98	38267.13
Normalized Std.	0.09000	0.18450	0.20031	0.19233	0.19871	0.20224	0.19150	0.18381	0.15307

Table 3.4: Spec CP-BTI PoC for Intel. Run for various N, accumulated over 10 executions.

N	10	100	1000	10000	15000	20000	40000	100000	250000
# Wrong	0	0	0	0	0	0	0	0	0
Correct in %	41.00	36.90	40.32	60.08	41.22	53.00	64.03	85.44	88.77
Correct Std.	4.39	41.53	404.14	3619.00	6006.92	7919.20	11251.06	10842.63	5005.44
Normalized Std.	0.43920	0.41532	0.40414	0.36190	0.40046	0.39596	0.28128	0.10843	0.02002

In both cases, the speculative version performs better than the non-speculative one.

We were not able to collect results for AMD. The Spec Ret-BTI for AMD is functional, but due to its instability and low reliability, it was not possible to do any proper recording. However, the signal strength and reliability were too low to be properly recorded. We were unable to get the Spec CP-BTI PoC to work for AMD.

Discussion. The most notable result is that the speculative versions of the PoCs have a higher success rate than the non-speculative ones. The standard deviation is generally also higher, indicating that the performance fluctuates more.

When looking at the non-speculative PoCs, one might notice that their performance is much worse than the ones shown by Wikner and Razavi [1]. This has multiple reasons; Firstly, we did not use the latest and most performant versions of the PoC to gather these results, but we used the PoCs as of the version on which we based the speculative PoCs. In addition, we disabled compiler optimizations for all PoCs, as some optimizations caused the speculative PoCs to stop working in some cases.

3.4 Discussion

By creating a speculative version of Retbleed's CP-BTI primitive, we have shown that it is possible to create a version of Retbleed that does not rely on PFs. Therefore, in-depth mitigation is required to patch the vulnerability completely. This gives a positive answer to RQ1.

Moreover, the successful development of Spec RET-BTI shows that speculative BTI is possible in the same privilege domain. In addition, Spec CP-BTI shows that it even works across privilege boundaries. The later, we were only able to show for Intel.

3.4.1 Challenges

With the proprietary nature of CPUs and the limited public knowledge on the BPU's internals, developing the speculative primitives was challenging. On and off, the BPU behaved unexpectedly. Sometimes trial and error was required to figure out how to align certain instructions to achieve the best possible performance.

While we failed in creating the Spec CP-BTI primitive for AMD, we believe that AMD Zen1, Zen 1+ and Zen 2 are also susceptible to speculative BTI across privilege boundaries. We presume that our PoC is not working due to some minor error, like alignment issues or other requirements of the BPU, of which we are unaware of.

All in all, these PoCs are very fragile. Many different components influence the success rate of these PoCs. If one is not right, the whole PoC may not work. In section A.1 we have a listing of components that we have considered during the development.

3.4.2 Retbleed in Virtualised Systems

So far, we have only considered bare-metal systems for our discussion on PF-free RETBLEED. We will briefly discuss another scenario that allows for employing RETBLEED without causing any PFs. Hence, even if it is impossible to create a PF-free training primitive for AMD, this gives reason enough to consider a more in-depth mitigation. We assume a virtualized environment with a guest OS where an attacker has obtained root privileges. The attacker aims to hijack a return instruction of a hypervisor process. As the BTB is not flushed on a guest-to-hypervisor switch, the hypervisor is influenced by branch feedback from the guest. This way, the attacker can poison the BTB and hijack a hypervisor branch. Since both the guest's address space and the hypervisor's address space are equally sized, the branch target can be injected without causing any PFs.

Chapter 4

Retbleed Mitigation

4.1 Overview

As seen in chapter 3, we were able to develop a version of Retbleed that does not rely on PFs, and therefore, *IBPB-on-PF* renders to be an insufficient mitigation. Intel and AMD have both released patches for Retbleed for their respective CPUs [5, 6, 27]. As merely every mitigation, they induce an unavoidable overhead, making the CPU operate slower. As these mitigations are rather involved and require extensive changes to the code base, the induced overhead is hard to predict. Therefore, to answer RQ2, we conduct a performance evaluation on the patches. We describe the used methodology in the next section, followed be an evaluation on the results.

4.2 Methodology

Benachmarking Suite. Byte-UnixBench [28] is the benchmarking suite we use for this evaluation. This suite is composed of 12 end-to-end benchmarks which test different aspects of the system. While certain tests will not interact with the mitigations, as they do not leave the user space, other tests, like the *System Call Overhead* test, are designed to benchmark the cost of entering and leaving the kernel space. Table B.1 shows the index score of all tests, for mitigation disable and enabled. It is clearly visible which tests are affected by the mitigation and which not.

Most tests are composed of a loop, where a specific task is performed in each iteration. Depending on the *System Under Test* (SUT)'s performance, this task takes a different amount of time to complete. The loop is run for a fixed amount of time, and the number of iterations is recorded. The *result* is directly linked to the achieved number of repetitions. As different tests evaluate various parts of the system, the raw results of different tests are hardly comparable. To allow for better comparability, the benchmarking suite brings the SUT's results into relation to a baseline system's results. The normalized result is referred to as *index score* and expresses how many times the SUT is more performant than the baseline system.

In addition, each test is run twice. Firstly, only a single instance of the test is run, while in the second run, N copies, with N being the number of hyper-cores, are run simultaneously.

System Under Test. As discussed in section 2.4.4, different mitigations have been released for different microarchitectures. To get a good overview of the performance impacts, we benchmark multiple vulnerable microarchitectures; Intel Coffee Lake, AMD Zen 1, and AMD Zen 2. All SUT run unmodified Linux 5.19.0-rc6, where Retbleed mitigations have been merged into [27]. For each SUT, we make two measurements; Once with Retbleed mitigation enabled and once with all

mitigations disabled. The mitigation is controlled using the kernel parameters `retbleed=auto` and `retbleed=off`, respectively. For AMD Zen 1, the default mitigation setting does not fully mitigate the issue, as a thread remains vulnerable to attacks from its sibling thread. To fully mitigate RETBLEED on Zen 1, it is required to disable *Simultaneous Multithreading* (SMT). SMT can be disabled using the parameter `retbleed=auto,nosmt`. For each SUT, we repeat the benchmarking 10 times[1] and record the results for each run.

4.3 Evaluation

The index scores of the 10 runs are accumulated by taking the geometric mean of the median of the test cases. We also compute the standard deviation of the index scores and normalize it to the accumulated index score. The standard deviation allows us to reason about the consistency and possible fluctuations of the benchmark over the 10 runs. Table 4.1 shows these results for the single-threaded and multi-threaded case. The index score of the unpatched systems is always larger than that of the patched ones, and the index score of multicore runs is higher than when only a single instance is run. The standard deviations are generally very low. However, for the single-threaded case of Zen 1, it is a few folds larger than the average for the other measurements.

Table 4.1: Accumulated index scores of the Byte-UnixBench benchmarking suit for different microarchitectures. The index score is accumulated for 10 runs by taking the geometric mean of the median of the scores of the test cases. The standard deviation is calculated for the same runs and is normalized to the accumulated index score.

		Index Score		Normalized Std.	
Microarch.	Patched	Single	Multiple	Single	Multiple
Coffee Lake	N	1945.29	9098.28	0.00200	0.00219
Coffee Lake	Y	1534.24	7452.39	0.00155	0.00140
Zen1	N	1982.78	8100.45	0.00702	0.00204
Zen1	Y	1746.12	7723.06	0.00614	0.00158
Zen1 (NoSmt)	Y	1772.93	5873.07	0.01427	0.00193
Zen2	N	2548.24	10840.09	0.00278	0.00198
Zen2	Y	2206.41	9581.83	0.00301	0.00163

To be able to better reason about the results, we normalize the index scores by taking the index score of the unpatched version as a baseline. For Zen 1 without SMT, we consider the unpatched version with SMT enabled as the baseline.[2] We calculate the overhead of the mitigation as:

$$\left(\frac{\text{unpatched}}{\text{patched}} - 1\right) \cdot 100 \tag{4.1}$$

The normalized index score and the overhead are listed in Table 4.2. For the single instance case, the overhead lies between 11.8 % and 15.5 % for the AMD CPUs, and for Intel Coffee Lake, it is over 26.8 %. When looking at the results of the multiple instances case, Zen 1 without SMT is the highest, with a massive 37.93 % overhead. On the other hand, Zen 1 with SMT is the lowest,

[1] One run takes approximately 1 h to complete.

[2] This is a reasonable assumption as there is generally no reason to disable hyper-threading.

with an overhead of 4.89 %. The overhead for Coffee Lake is a bit smaller, compared to the single instance case, with 22.09 %. The overhead for Zen 2 is 13.13 %.

Table 4.2: The index scores were taken from Table 4.1 and normalized to the score of the unpatched kernel. The overheads are calculated as in Equation 4.1.

	Single		Multiple	
Microarch.	Norm. Index Score	Overhead in %	Norm. Index Score	Overhead in %
Coffee Lake	0.78869	26.79	0.81910	22.09
Zen 1 (Smt)	0.88064	13.55	0.95341	4.89
Zen 1 (NoSmt)	0.89416	11.84	0.72503	37.93
Zen 2	0.86586	15.49	0.88393	13.13

4.4 Discussion

Based on the low standard deviation, there are no significant fluctuations between consecutive runs of the benchmarking suite. This is desired as this way the results can be considered more convincing. Interesting is that Zen 1 seems less stable than the other microarchitectures when only one instance is run. However, with less than 1.5 % fluctuations in the index score, the results are still acceptable. As these instabilities also occur with mitigation disable, it is most likely not related to the patches.

Coffee Lake exhibits a rather significant overhead. This is due to the IBRS [19]-based mitigation [6]. IBRS is an indirect branch control mechanism that restricts the speculation of indirect branches, preventing Retbleed from hijacking branches across privilege boundaries. Kernel developers started considering lower-cost mitigation for Intel, which works by detecting and preventing the RSB from underflowing [29].

AMD Zen 1 and Zen 2 use the same jmp2ret "untrain return thunk" as the basis for their mitigation. The 4.89 % overhead for Zen 1, with SMT enabled, in the multi-threaded case, probably directly represents the overhead of jmp2ret. However, jmp2ret does not fully mitigate the issue, as it leaves a thread vulnerable to attacks from its sibling thread. Therefore, we consider 5.0 ∼ as the base overhead for AMD Zen 1 and Zen 2.

To overcome this vulnerability issue, Zen 2 uses *Single Thread Indirect Branch Prediction* (STIBP) [30]. This mechanism prevents indirect branch prediction resolution from influencing sibling threads. STIBP adds ∼8 % overhead on top of the base overhead, resulting in an overhead of 13.13 %. Without STIBP on Zen 1, the only option to protect a thread from its sibling is to disable SMT. Disabling SMT has a substantial performance impact; it adds over 32 % on top of the base overhead. For AMD microarchitectures, the mitigation overhead in the single-threaded cases is roughly the same, with ∼13 % on average. That is because neither disabling SMT nor using STIBP really affects the single-threaded case. However, it is not clear why the overheads of the single-threaded case are still roughly ∼8 percentage point larger than the jmp2ret base overhead. We have assumed that the overhead is roughly equal.

The benchmarking suite contains multiple test cases, evaluating different parts of the system. While certain test cases are not affected by the mitigation, we assume that the combination of multiple cases is a reasonable estimate of the overhead of real-world workflows.

The benchmark's documentation indicates that the suite's results depend on hardware, op-

erating systems, libraries, and used compiler. While SUTs are based on different hardware, the normalization of the patched system to the unpatched ones should make SUTs comparable among each other. All systems run the same version of Linux, which was compiled from source using the same compiler version. On top of that, the benchmarking suite was compiled with the same compiler for the different SUTs.

All these steps help to make the results as meaningful as possible. However, a benchmarking suite still serves only as an abstraction of a real-world workload, and there are many possible sources of errors. Therefore, the results must be considered carefully.

Chapter 5

Related Work

This thesis builds on the results and findings of various papers. They are all cited in the appropriate places. First and foremost, this paper builds on the discoveries of Spectre [3]. However, Retbleed [1] has inspired us to evaluate speculative BTI.

We want to use this section to comment on a few papers we have not had the chance to introduce yet and which we find interesting or relevant.

Phantom JMPs. During the research on Retbleed, Wikner and Razavi discovered a new class of speculative executing attacks, which they refer to as Phantom JMPs [31]. Phantom JMPs are speculative branches triggered by a non-branching instruction, such as an arithmetic instruction. They exploit a prediction mechanism that prediction, before the instruction has been decoded, if the instruction is a branching instruction and where it possibly jumps to. By tricking that predictor into assuming that at a particular location, there is a branch, the speculative control flow can be hijacked. They showed that AMD Zen 1, Zen 1+ and Zen 2 are affected. Phantom JMPs were mitigated with the same patches that mitigate Retbleed [26, 31].

Straight Line Speculation. Another interesting Spectre vulnerability, targeting Arm and AMD CPUs, is *Straight Line Speculation* (SLS) [32]. SLS differs from Retbleed and most Spectre vulnerabilities we have looked at in that it does not rely on misprediction of the branch prediction unit but exploits a strange behavior of the BPU itself. When encountering an unconditional branching instruction, the BPU predicts that the control flow continues as there was no branching instruction. This behavior also appears for return instructions. When an attacker can control the executed code, it can place a disclosure gadget after such an instruction. If successful, the disclosure gadget gets executed speculatively, causing the possible leakage of data. While hard to exploit, it has been mitigated. First compilers like GCC [33] and LLVM [34] were patched, and later on, the Linux kernel [35]. The idea of the mitigation is simple. A speculation barrier, for example, the debug instruction `int3`, is added after an affected branching instruction.

Branch History Injection. *Branch History Injection* (BHI) [36], an extension of Spectre V2, has recently been published. Barberis *et al.* have shown that hardware mitigation employed specifically for Spectre attacks, like eIBRS [19] and CSV2 [37], can be circumvented, enabling an attacker to conduct Spectre V2 attacks. These hardware mitigations work by keeping track of the privilege level where a branch is executed, using global history. However, an attacker is can tamper with the history from user space to cause speculative execution of a disclosure gadget.

Chapter 6

Conclusion

We have developed a version of Retbleed's CP-BTI primitive for Intel, which does not rely on PF. This PoC demonstrates that the IBPB-on-PF mitigation is insufficient in fixing the vulnerability and that in-depth patches are required. In the course of developing the PF-free primitive, we have shown that speculative BTI works. We have developed PoCs for both Intel and AMD, which demonstrate that Spec BTI works in the same privilege domain, and for Intel, we were also able to show it is working across privilege boundaries. The Spec CP-BTI PoC for AMD did not perform as expected. Further investigation is needed to know whether our findings based on the Intel microarchitecture transfer to AMD microarchitectures.

To the best of our knowledge, the idea of using speculation to cause another speculation is new. We believe this novel approach might prove useful in the discovery of unknown vulnerabilities of the BPU.

By benchmarking the mitigation, we were able to show that common microarchitectures suffer from 13 %–37 % performance penalty when ultimately mitigating Retbleed. As this is substantial overhead, we support the development of alternative mitigations like the ones from Gleixner [29]. Alternatively, one might consider different mitigation techniques like preventing convert channels altogether [38].

Acknowledgements

I want to thank all the people who have supported me while writing my bachelor thesis.

My tutor, Johannes Wikner, who introduced me to the topic and guided me along the way. For his support and the sharing of insight and knowledge on the matter.

My supervisor, Prof. Dr. Kaveh Razavi, to enable to me to work on this exciting thesis at his group [39]. For his suggestions, hints and explanations at the regular meeting.

Dr. Hendrik Dietrich for proofreading and feedback on the thesis.

List of Figures

List of Tables

Listings

Bibliography

[1] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *31th USENIX Security Symposium (USENIX Security 22)*, Jul. 12, 2022.

[2] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[4] Intel, *Indirect branch predictor barrier*, Mar. 1, 2018. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html` (visited on 08/20/2022).

[5] AMD, *Technical guidance for mitigating branch type confusion*, Jul. 12, 2022. [Online]. Available: `https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf` (visited on 07/25/2022).

[6] Intel, *Return stack buffer underflow / CVE-2022-29901, CVE-2022-28693 / INTEL-SA-00702*, Intel, Jul. 5, 2022. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/return-stack-buffer-underflow.html` (visited on 07/25/2022).

[7] J. Wikner and K. Razavi. "RETBleed artifact." (Jul. 12, 2022), [Online]. Available: `https://github.com/comsec-group/retbleed` (visited on 07/22/2022).

[8] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.

[9] "The linux kernel archives." (2022), [Online]. Available: `https://www.kernel.org/` (visited on 07/23/2022).

[10] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Securityj 14)*, USENIX Association, 2014, pp. 719–732.

[11] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "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.

[12] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *2013 IEEE Symposium on Security and Privacy*, IEEE, 2013, pp. 191–205.

[13] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Cryptographers' track at the RSA conference*, Springer, 2006, pp. 1–20.

[14] Chip Wiki. "Haswell - microarchitectures - intel." (2022), [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client) (visited on 07/14/2022).

[15] A. Fog, *3. the microarchitecture of intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, Jun. 11, 2022. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf (visited on 08/20/2022).

[16] Project Zero Team. "Reading privileged memory with a side-channel." (Jan. 3, 2018), [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html (visited on 07/26/2022).

[17] AMD, *Software techniques for managing speculation on AMD processors*, Jul. 14, 2022. [Online]. Available: https://developer.amd.com/wp-content/resources/90343-B_SoftwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf.

[18] H. J. Lu. "X86: CVE-2017-5715, aka spectre." (Jan. 7, 2018), [Online]. Available: https://gcc.gnu.org/legacy-ml/gcc-patches/2018-01/msg00422.html (visited on 07/24/2022).

[19] Intel, *Indirect branch restricted speculation*, Jan. 3, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html (visited on 08/20/2022).

[20] Intel Corporation, *Retpoline: A branch target injection mitigation*, Jun. 2018. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf (visited on 07/24/2022).

[21] P. Turner, "Retpoline: A software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886 (visited on 07/24/2022).

[22] L. Torvalds. "X86/enter: Create macros to restrict/unrestrict indirect branch speculation." (Jan. 21, 2018), [Online]. Available: https://lwn.net/Articles/745112/ (visited on 07/24/2022).

[23] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, USENIX Association, 2018.

[24] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[25] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, "Spring: Spectre returning in the browser with speculative load queuing and deep stacks," in *WOOT*, May 2022. [Online]. Available: https://comsec.ethz.ch/wp-content/files/spring_woot22.pdf%20URL=https://comsec.ethz.ch/research/microarch/spring (visited on 08/20/2022).

[26] AMD. "AMD CPU branch type confusion." (Jul. 12, 2022), [Online]. Available: https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion_v7_20220712.pdf (visited on 07/25/2022).

[27] L. Torvalds. "Merge tag 'x86_bugs_retbleed'." (2022), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ce114c866860 (visited on 07/23/2022).

[28] kdlucas. "Byte-UnixBench." (2022), [Online]. Available: `https://github.com/kdlucas/byte-unixbench` (visited on 07/23/2022).

[29] T. Gleixner. "X86/retbleed: Call depth tracking mitigation." (Jul. 17, 2022), [Online]. Available: `https://lwn.net/ml/linux-kernel/20220716230344.239749011@linutronix.de/` (visited on 07/23/2022).

[30] AMD, *Indiredct branch control extension*. [Online]. Available: `https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf` (visited on 08/20/2022).

[31] J. Wikner, D. Trujillo, and K. Razavi, "Addendum to Retbleed: Arbitrary speculative code execution with return instructions," Jul. 2022.

[32] ARM, "Straight-line speculation whitepaper," Jun. 2020. [Online]. Available: `https://developer.arm.com/documentation/102825/0100/?lang=en` (visited on 07/24/2022).

[33] M. Malcomson. "Straight line speculation (SLS) mitigation." (Jun. 8, 2020), [Online]. Available: `https://gcc.gnu.org/pipermail/gcc-patches/2020-June/547520.html` (visited on 07/26/2022).

[34] K. Beyls. "[llvm-dev] mitigating straight-line speculation vulnerability CVE-2020-13844." (Jan. 8, 2020), [Online]. Available: `https://lists.llvm.org/pipermail/llvm-dev/2020-June/142109.html` (visited on 07/26/2022).

[35] P. Zijlstra. "[PATCH v2 0/6] x86: Add stright-line-speculation mitigations." (Dec. 4, 2021), [Online]. Available: `https://lwn.net/ml/linux-kernel/20211204134338.760603010@infradead.org/` (visited on 07/24/2022).

[36] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *USENIX Security*, Aug. 2022. [Online]. Available: `Paper=http://download.vusec.net/papers/bhi-spectre-bhb_sec22.pdf%20Web=https://www.vusec.net/projects/bhi-spectre-bhb%20Code=https://github.com/vusec/bhi-spectre-bhb`.

[37] Arm, *Feature names in a-profile architecture*, 2021. [Online]. Available: `https://developer.arm.com/downloads/-/exploration-tools/feature-names-for-a-profile` (visited on 08/20/2022).

[38] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, "Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core," 2020.

[39] "ComSec: Computer security group." (2022), [Online]. Available: `https://comsec.ethz.ch/` (visited on 07/23/2022).

Appendix A

PoC Additions

A.1 Improve Signal Strength and Reliability

In this subsection, we want to briefly mention a few techniques we have employed to improve the signal strength and reliability of the PoCs. We will also emphasize findings we have made during the implementation related to these techniques.

Branch History. As the BTB for Intel is indexed using the branch history condensed in the BHB, the probability that the injected branch is used for the prediction is the highest, if the history at the time of the hijack is equivalent to the one in training. While the return cycle is vital for creating a normalized history, subsequent branches can easily disrupt the history again. That is why we call the rogue or fake-rogue function, also in the speculation phase. While having equivalent branch histories is ideal, this is not achievable for CP attacks.

Besides the overall branching schema, which should match, branch addresses should also be aligned. This is achieved by the appropriate use of `nop`s.

However, we have also seen cases where perfectly aligned branches performed worse than cases when the history is slightly off. Contrary to our expectations, the performance also degenerates quickly when making the return cycle for the speculative Intel PoCs longer than 29 entries. We had also seen cases where the performance degraded when the PoC was executed many consecutive times.

Align Basic Blocks. Since the BTB for AMD is indexed only using the start and end address of a fetch unit, it is required that they are perfectly aligned. As with the branch history, this is achieved by perfectly aligning instructions using `nop`s.

Speculation Window Size. We use SpectreRSB to create the speculation window used for the speculative BTI. It is a reliable way to create speculation, and the window size is generally relatively large. It can be improved by ensuring that obtaining the architectural return address takes as long as possible. For that, we flush the address from all cache layers, leading to a memory access.

Prevent Speculative Escape. With "speculative escape", we refer to cases when the (speculative) control flow of the PoCs does not follow the indented control flow. When branches in the primitive are subject to speculation themselves, signal strength may be degraded, the PoC works unreliable, wrong secrets may be leaked, or the PoCs appear to work while it is not.

To prevent that from happening, we apply speculation barriers before branching instructions. For x86_64 this is done using `lfence` instruction.

A.2 Verify the Source of Speculation

The output of the PoC that the secret has been leaked successfully does not necessarily mean that the PoC works as expected. An earlier branch in the primitive may cause speculative execution of the disclosure gadget. This way, we do not know if the desired branch is hijacked as intended. To verify that, we can use a trick. Right before the victim branch, we modify the secret by, for example, incrementing it. If the leaked secret conforms with our changes, the speculation comes from the right place. Otherwise, we must reconsider our design and prevent speculation from other parts of the primitive.

Appendix B

Mitigation

Table B.1: Byte-UnixBench Benchmark for Coffee Lake with RETBLEED mitigations disabled and enabled. ST represents the single-threaded case, where MT represents the multi-threaded case.

	Mitigation Off		Mitigation Auto	
Testcase	ST	MT	ST	MT
Dhrystone 2 using register variables	5152.1	39353.2	5144.6	39369.9
Double-Precision Whetstone	1840.7	18228.2	1839.3	18248.0
Execl Throughput	1306.0	8836.7	1011.2	7410.7
File Copy 1024 bufsize 2000 maxblocks	2452.1	5777.0	1705.5	4672.2
File Copy 256 bufsize 500 maxblocks	1511.0	3670.2	1039.3	2849.1
File Copy 4096 bufsize 8000 maxblocks	5199.1	11045.8	3931.9	10453.6
Pipe Throughput	960.8	6627.8	643.2	4431.5
Pipe-based Context Switching	641.4	3964.4	567.4	3058.2
Process Creation	1057.4	8014.8	842.8	6693.8
Shell Scripts (1 concurrent)	3536.1	19618.4	3039.2	16238.6
Shell Scripts (8 concurrent)	12226.1	17954.7	9617.3	14930.5
System Call Overhead	445.1	2959.5	290.3	1810.2
System Benchmarks Index Score	1948.3	9108.0	1537.2	7455.6

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work:

An Evaluation of Speculative Retbleed

Thesis type and date:

Bachelor Thesis, August 21, 2022

Supervision:

Prof. Dr. Kaveh Razavi
Johannes Wikner

Student:

Name:	Jean-Claude Graf
E-mail:	jegraf@student.ethz.ch
Stud.-Nr.:	19-940-171
Study Semester:	6

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the citation etiquette[1] information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

Zürich, 21.08.2022

Jean-Claude Graf

Place, Date

Jean-Claude Graf

[1] https://ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/plagiarism-citationetiquette.pdf