RETBLEED: Arbitrary Speculative Code Execution with Return Instructions

Johannes Wikner
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

Modern operating systems rely on software defenses against hardware attacks. These defenses are, however, as good as the assumptions they make on the underlying hardware. In this paper, we invalidate some of the key assumptions behind *retpoline*, a widely deployed mitigation against Spectre Branch Target Injection (BTI) that converts vulnerable indirect branches to protected returns. We present RETBLEED, a new Spectre-BTI attack that leaks arbitrary kernel memory on fully patched Intel and AMD systems. Two insights make RETBLEED possible: first, we show that return instructions behave like indirect branches under certain microarchitecture-dependent conditions, which we reverse engineer. Our dynamic analysis framework discovers many exploitable return instructions inside the Linux kernel, reachable through unprivileged system calls. Second, we show how an unprivileged attacker can arbitrarily control the predicted target of such return instructions by branching into kernel memory. RETBLEED leaks privileged memory at the rate of 219 bytes/s on Intel Coffee Lake and 3.9 kB/s on AMD Zen 2.

1 Introduction

> *The patches do things like add the garbage MSR writes to the kernel entry/exit points. That's insane. That says "we're trying to protect the kernel". We already have retpoline there, with less overhead. So somebody isn't telling the truth here.*
> – Linus Torvalds on IBRS patches [59].

In the absence of hardware mitigations, widespread and particularly dangerous transient execution vulnerabilities [9, 11, 36, 39, 40, 49, 50, 54, 61, 62] are mitigated in software [46, 60, 63, 68]. Given proprietary hardware, software developers must sometimes rely on the information provided by hardware vendors for the most efficient implementation of their defense. In this paper, we show this lack of transparency can result in vulnerable systems despite deployed software mitigations. In particular, we present a new transient execution attack that can leak kernel memory from an unprivileged user on a variety of microarchitectures despite the deployed *retpoline* mitigations against Spectre Branch Target Injection (BTI).

Spectre-BTI. Also referred to as Spectre Variant 2 [36], Spectre-BTI forces the speculative execution of an arbitrary target for a victim indirect branch. For exploitation, the attacker must inject the desired branch target inside the Branch Target Buffer (BTB) for speculative execution (i.e., branch poisoning). To poison the BTB, the attacker must find collisions with the victim branch inside the correct BTB set. On a successful collision, upon observing the victim indirect branch, the branch predictor serves the poisoned target for speculative execution. The hijacked speculative execution can then leak sensitive information by encoding it inside the cache, which can then be leaked via a timing attack.

Spectre-BTI mitigations. Indirect Branch Restricted Speculation (IBRS) was originally proposed by Intel to mitigate Spectre-BTI against privileged software. IBRS enables the processor to ignore (potentially poisoned) BTB entries created from lower-privileged software, but requires expensive Model-Specific Register (MSR) writes on user–kernel transitions. A competing proposal, retpoline [60] outperformed IBRS by replacing indirect branch instructions with return instructions. Return instructions use a different prediction scheme that is cheaper to flush on user–kernel transitions. Despite concerns about the behavior of return prediction in deep call stacks on Intel Skylake [15, 20, 69], the risk was considered low [15, 58] and ultimately retpoline became the de facto mitigation against Spectre-BTI in privileged software. Retpoline is supported by modern compilers [8, 13], and even recommended by AMD [5].

RETBLEED. Previous work has reverse engineered the behavior of the Branch Prediction Unit (BPU) for indirect branches [18, 32, 36]. In this paper, we try to understand the BPU's behavior on return instructions. Our reverse engineering results show that all return instructions that *follow sufficiently-deep call stacks* can be hijacked using a precise branch history on Intel CPUs. On AMD CPUs, we

find that *any return instruction can be hijacked, regardless of the previous call stack*, as long as the previous *branch destination* is correctly chosen during branch poisoning. Furthermore, in many microarchitectures, it is possible to create collisions on kernel return instructions from user mode.

Armed with these insights, we build a dynamic analysis framework on top of the standard testing and tracing facilities in the Linux kernel to look for microarchitecture-dependent *exploitable* return instructions that provide the attacker sufficient control over registers or memory. On AMD, without the need for deep call stacks, return instructions right after the transition to the kernel via unprivileged system calls are readily exploitable. Even on Intel, we could find many instances of exploitable return instructions that come after deep call stacks. These exploitable returns form the first part of our end-to-end exploit called RETBLEED. Unlike previous return-based Spectre attacks [37, 40, 67], RETBLEED exploits return instructions to gain arbitrary kernel-level speculative code execution by targeting the BTB instead of the RSB. It is, however, not possible for an attacker to direct speculative execution from a hijacked kernel return to user space due to Supervisor Mode Execution Prevention (SMEP). To circumvent SMEP, the attacker can instead use a destination in kernel space. We make a key observation that branch resolution feedback is recorded in the BTB *across privilege domains*, independently of the correct execution of the branch target. This allows an attacker to inject a poisoned BTB entry with a kernel target from user space. Our evaluation shows that with suitable disclosure gadgets, RETBLEED can leak arbitrary kernel data from an unprivileged process with all the mitigations up on various Intel and AMD systems.

Contributions. We make the following contributions.

1. We provide a detailed reverse engineering of the behavior of return instructions on Intel and AMD processors.
2. Using the insights from our reverse engineering, we build a framework using standard Linux testing and tracing facilities that allows us to identify vulnerable return instructions for different microarchitectures.
3. We show that the destination of BTB entries can be poisoned to a kernel address by an unprivileged process.
4. Building on vulnerable returns and poisoned BTB entries, RETBLEED achieves arbitrary kernel-level speculative execution, leaking arbitrary kernel data at the rate of 219 bytes/s (98 % accuracy) on Intel Coffee Lake and 3.9 kB/s (> 99 % accuracy) on AMD Zen 2.

Responsible disclosure. We disclosed RETBLEED to the affected parties in February 2022. RETBLEED was under embargo until July 12, 2022 to provide adequate time for the development and testing of new mitigations, which we discuss in Section 9. RETBLEED is tracked under CVE-2022-29900 (AMD) and CVE-2022-29901 (Intel). Further information can be found at: https://comsec.ethz.ch/retbleed

2 Background

We provide some necessary background on caches and cache attacks, branch prediction and speculative execution attacks, and the deployed mitigations.

2.1 CPU caches

Commodity CPU caches are organized in multiple levels, where lower levels are faster and smaller than their higher level counterparts. Assuming a common three-level cache hierarchy, lower level caches, L1 and L2, reside in each physical core, and the Last Level Cache (LLC or L3) is shared across all cores. Caches are managed in the granularity of a *cache line* which is a contiguous block of physical memory. Depending on its physical address, each cache line is mapped to a specific *set* inside a cache. Each set can store N entries, known as the *wayness* of the cache. With this organization, any given memory location can be translated into a cache line for a specific cache set, and a tag that identifies it within the set.

Because of their limited size, caches must also implement a set-level replacement policy, such that rarely used cache lines can be evicted and replaced. Moreover, each level has a cache inclusion policy that determines whether cache lines present in one level must also exist in higher levels, thus called *inclusive*, or *non-inclusive* if it is not required. A cache can furthermore be *exclusive* from some other cache level if they may not share cache lines. Whereas commodity CPUs typically implement inclusive caches, non-inclusive and exclusive inclusion policies are common for higher levels caches. By the nature of caches, cache misses are significantly slower than hits, which is exploited in cache attacks.

2.2 Cache attacks

There are many methods to leak information from CPU caches [17, 24, 44, 47, 70]. We discuss the two most popular ones that are also used in this paper, namely PRIME+PROBE [44] and FLUSH+RELOAD [70]. With PRIME+PROBE the spy first fills up an entire cache set with their own memory. They then trigger, or wait for, some victim activity to occur that might be significant for the given cache set. In the last step, the spy then reloads their own memory while measuring the access time. If the access time is *above* a computed threshold, the spy infers that the cache set was populated with the victim's memory, which consequentially evicted some of the spy's memory from the cache. A PRIME+PROBE attack only requires shared caches between spy and victim, but it requires deeper knowledge of the cache implementation (i.e., indexing, replacement and inclusion policy), and it has higher error rate and lower bandwidth than FLUSH+RELOAD.

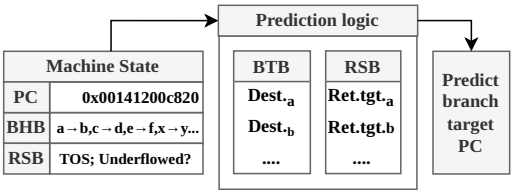

Figure 1: Simplified BPU overview. BHB is a fingerprint of the last control flow edges (i.e., taken branches). Using Machine State, such as BHB and PC at a branch instruction, BPU predicts the branch target via the BTB. For returns, unless underflow condition, it uses the current RSB Top Of Stack (TOS) pointer to predict return target.

FLUSH+RELOAD assumes not only that the target cache level is shared between spy and victim, but that they also share some memory. Here, the spy first flushes some shared memory from the cache and then waits for, or triggers, the victim's activity. The spy then reloads the shared memory and measures the access time. If the access time is *below* a computed threshold, the spy infers that the victim accessed the shared memory since it was flushed. As we will soon discuss, Spectre attacks [36] often provide an attacker with the possibility of sharing memory with speculatively-executed code, enabling the use of FLUSH+RELOAD. In this paper, we refer to this shared memory as the *reload buffer*.

2.3 Branch prediction

Modern CPUs rely on speculative execution of code to improve the overall performance by reducing stalling. The BPU, located early in the execution pipeline, uses a set of branch predictors to predict the next instruction address when it encounters *direct* or *indirect* branches. In the x86_64 ISA, direct branches are either *conditional* or *unconditional*, whereas indirect branches are always *unconditional*.

Figure 1 shows an overview of a simplified BPU. Direct and indirect branch targets are predicted using the Branch Target Buffer (BTB), which stores possible targets for a given branch. Indexing of the BTB is microarchitecture-dependent, and it uses machine state, such as the current Program Counter (PC) and previous branches recorded in Branch History Buffer (BHB) to accurately predict the next branch target. Jann Horn reverse engineered the BTB indexing for the Haswell microarchitecture as part of his work on Spectre [32]. This information, however, is currently missing for newer Intel microarchitectures and recent AMD microarchitectures that use different prediction schemes [57].

On top of predicting direct and indirect branches, modern microarchitectures use a dedicated scheme for return target prediction. A Return Stack Buffer (RSB) records return targets inside a microarchitectural stack to predict target addresses of return instructions after multiple function calls. However, given the limited capacity of the RSB, different microarchitectures may resort to other branch prediction schemes as needed. As an example, while this has never been explicitly shown, the Skylake microarchitecture is known to revert to BTB prediction on RSB underflows [15, 20, 69].

2.4 Spectre attacks

Speculative execution attacks exploit branch predictors by influencing (i.e., "training") them to execute incorrect code paths. Given that branch prediction is imprecise, speculative execution sometimes takes code paths that are architecturally incorrect. In these cases, processors squash incorrect computation and restart the execution from the correct path. Although incorrect speculative execution is not committed to the architectural state, speculative memory reads affect the state of CPU caches and can thus be leaked by measuring memory access times through cache attacks.

Attacks that are collectively referred to as Spectre attacks [36] have different variants, with the first two variants being the most prominent. Spectre Variant 1, known as Spectre-BCB (Bounds Check Bypass), forces incorrect paths of conditional branches to be taken. In many exploitable cases, these conditions check for a valid bound before a memory access. In essence, Spectre-BCB forces a speculative out of bound memory access. Spectre Variant 2, or Spectre-BTI (Branch Target Injection), exploits unconditional indirect branches by forcing speculative execution of an incorrect indirect branch target. Spectre-BTI achieves this by poisoning the BTB with attacker-controlled targets. This poisoning was shown to be possible across different address spaces because the BPU records a branch target even if the target is not accessible in the same address space [36]. We show in this paper that this is also possible across privilege boundaries to enable user to kernel exploitation.

Another variant of interest is Spectre-RSB [37, 40, 67] that triggers speculation through RSB underflows, or by overwriting the return address on the stack. Because RSB predictions are limited to previously-valid return targets, Spectre-RSB is difficult to exploit in scenarios where the attacker cannot generate code (e.g., inside the kernel), and when mitigations are in place, which we will discuss in Section 2.5.

Exploitation. Spectre attacks require three components:

1. **Speculation Primitive.** This is a code path that forces a desired branch prediction. The predicted branch target or direction (e.g., *taken* or *not taken*) is attacker-controlled.

2. **Disclosure Gadget.** The target code gadget that is speculatively executed due to misprediction at the Speculation Primitive. Due to the nature of Spectre, the Disclosure Gadget and Speculation Primitive reside in the same address space [36].

3. **Covert Channel.** The disclosure gadget leaves a microarchitectural trace that can be detected using a covert channel, typically a cache attack.

To use the Speculation Primitive, the attacker executes code that trains the BPU to make a misprediction. The most trivial example is Spectre-BCB, where an attacker executes a conditional branch a number of times with the same input. They then execute the same conditional branch with a different input to invert the conditional outcome. This causes a misprediction, since the BPU assumes that the conditional branch has the same outcome as before. While similar in-place training is possible for Spectre-BTI, a less constrained attack creates collision without the need to execute the victim branch at all, referred to as *out-of-place* (OOP) training [10].

2.5 Spectre mitigations

The three directions of Spectre attack defenses are isolation, prevention of speculation, and prevention the of covert channel [41]. Whereas the latter category is popular among web browsers [46, 63], it is unfeasible in native contexts.

To mitigate Spectre-BCB, one possibility is to make sure that untrusted pointers are adequately sanitized right before they are accessed. This is a mitigation that is commonly adopted by the browser [46] and the kernel [2, 68]. Another possibility is to remove secrets from a potentially-malicious address space. This is done in the context of site isolation for browsers by making sure that each origin runs in its own address space [1]. It is, however, not trivial to isolate different security domain inside a monolithic kernel.

To mitigate Spectre-RSB, it is common to fill the RSB with harmless return targets on context switch to ensure to stop RSB poisoning across user processes.

To mitigate Spectre-BTI, indirect branches should either not make use of untrusted entries in the BTB or they should be removed altogether. The former was proposed by Intel through a microcode update that provide *Indirect Branch Restricted Speculation* (IBRS) [14] enabling the temporary restriction of branch resolution feedback gathered in a lower privilege mode to be used in a higher privilege. IBRS requires writing MSR values on every privilege mode transition and causes significant performance overhead on many microarchitectures. Recent Intel processors have simplified the interface and improved the performance with *enhanced IBRS* (eIBRS). *Retpoline* [60] is a mitigation based on removing indirect branch prediction that was adopted instead of IBRS due to its lower performance overhead. Retpoline operates by converting all indirect branches into return instructions. However, AMD recommends an alternative retpoline implementation for their systems, which does not convert indirect branches. Instead, it adds a synchronizing instruction (i.e., `lfence`) between loading of the branch target and the indirect branch itself, which prevents speculation by making the speculative window too small to exploit [5].

In this paper, we try to understand the attack surface of return instructions under Spectre-BTI on various microarchitectures.

3 Threat Model

We consider a realistic threat model where an unprivileged attacker process aims to leak privileged information from the victim kernel. We assume the target processor to support speculative execution and the target kernel to be free of software vulnerabilities. We further assume that the target kernel is protected against known speculative execution attacks. More specifically in this paper, we assume the latest Linux kernel that enables all available mitigations against transient execution attacks, such as KPTI [22], retpoline [5, 15], user pointer sanitization [2] and disables unprivileged eBPF [42]. We show how an attacker can leverage Spectre-BTI to hijack return instructions on various Intel and AMD microarchitectures to leak arbitrary kernel data despite these mitigations.

4 Overview

RETBLEED aims to hijack a return instruction in the kernel to gain arbitrary speculative code execution in the kernel context. With sufficient control over registers and/or memory at the victim return instruction, the attacker can leak arbitrary kernel data. To achieve this, however, the attacker must overcome a number of challenges discussed next.

4.1 Challenges

First, it is unclear under which conditions the attacker can hijack the return instruction. The BPUs in different microarchitectures exhibit different behavior when observing a return instruction. We need to understand this behavior before we can hijack a return instruction, which is our first challenge.

> **Challenge (C1).** Understanding the behavior of the BPU when observing a return instruction in different microarchitectures.

Section 5 addresses this challenge by reverse engineering the behavior of BPU when observing return instructions on multiple microarchitectures and how they can be speculatively hijacked by creating collisions in the BTB. Our reverse engineering provides us with the necessary microarchitecture-dependent conditions for hijacking return instructions. But how can we find kernel return instructions that satisfy these conditions and provide an attacker with sufficient control over registers and/or memory? This is our second challenge:

> **Challenge (C2).** Finding return instructions in the kernel that an attacker can hijack while retaining sufficient control over registers and/or memory.

Section 6 addresses this challenge by using kernel function graph tracing for discovering call stacks of interest. By analyzing these call graphs, we discover system calls that

lead to *vulnerable* return instructions. We then match system call inputs with registers and their referenced memory when executing the vulnerable return instruction. System call inputs that match with registers and memory at the vulnerable return instruction are likely to be controllable by the attacker and are as such considered *exploitable* return instructions. But even with these return instructions at the attacker's disposal, how can they control the speculation target?

Challenge (C3). Controlling the speculation target of a hijacked return instruction.

We also address this challenge in Section 6 by showing that an unprivileged attacker can perform BTI on indirect branches and return instructions across privilege domains, without requiring permission to allocate executable memory at high addresses. With this capability, we now have full control over the speculative execution of a kernel return. Our next challenge is finding an exploitable disclosure gadget in the kernel.

Challenge (C4). Finding an exploitable disclosure gadget in the kernel.

We address this challenge in Section 7, where we show how we can find disclosure gadgets and how they can be used to leak arbitrary data. Furthermore, we address a number of practical challenges for reliable exploitation. These include increasing the speculation window, poisoning of the right BTB entry given the history of victim return instruction, and derandomizing KASLR.

4.2 Summary of the full-chain attack

Offline phase. The following steps are executed on a different machine than the victim, where the attacker has full access.

① *Detecting vulnerable returns.* We create a list of locations in the victim kernel where we can arbitrarily control the speculative instruction pointer.

② *Identifying exploitable returns.* We reduce the list of locations to only include those where we additionally control registers or memory references.

③ *Finding compatible disclosure gadgets.* We scan the kernel image for gadgets that enable FLUSH+RELOAD with any of the exploitable returns. We select the victim exploitable return accordingly.

④ *Detecting branch history at the victim return.* We generate a list of branch sources and target addresses preceding the victim return.

Online phase. The following steps are executed on the victim machine, where the attacker has no special privileges.

⑤ *Derandomizing kernel addresses.* We detect the absolute addresses of the disclosure gadget and the reload buffer in kernel memory. On AMD CPUs, we develop a new technique for this purpose.

⑥ *Setting up branch history for BTI.* We set the BHB to the same state as it will be at the victim return. We then perform BTI to the disclosure gadget which poisons the right BTB entry.

⑦ *Executeing the victim return via a system call.* Finally, executing the victim return triggers a BTB prediction to our disclosure gadget, allowing us to leak arbitrary kernel memory.

5 Branch Target Injection on Returns

Given the limited number of entries in the RSB, in this section we want to understand the behavior of the BPU in different microarchitectures when observing return instructions under different conditions. We first reverse engineer the BPU behavior of the indirect branches in different microarchitectures and then show how it interacts with returns.

5.1 Finding BTB collisions

As discussed in Section 2.3, the BPU makes use of BTB to predict the target of an indirect branch. The BTB is usually indexed using the PC of an indirect branch to find the branch target. However, an indirect branch can have multiple targets. The BPU tries to distinguish between these using branch history preceding the indirect branch. The combination of branch history and the branch's PC for the selection of the target makes the reverse engineering of BTB indexing particularly challenging.

BTB collisions on Intel. Previous BTB reverse engineering work on Intel revealed two BTB predictors [32, 36]. The history-backed predictor indexes the BTB using the PC of the last byte of the indirect branch source instruction and its preceding branch history of taken branches. The generic predictor is used if no prediction backed by branch history is available, and selects branch target solely using the PC of the indirect branch source instruction. Because we will target return instructions that follow a deep call stack, we expect a consistent branch history, resulting in using the history-backed predictor. To gain further insight into BTB indexing on Intel platforms, we reproduce the experiment described in [32], shown in Figure 2. This experiment executes a number of taken branches *branch_path*, to set the branch history to a consistent state before executing an indirect branch instruction. When training, the indirect branch target is the "Poisoned target", which signals its execution through a memory access that can later be inferred using FLUSH+RELOAD. For the victim, the branch target *should* lead to no signal, but

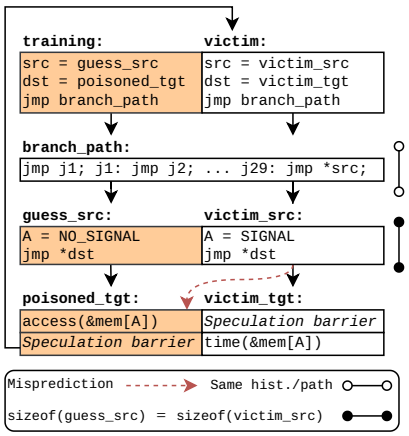

Figure 2: Experiment for reverse engineering BTB indexing. Triggering BTB collisions on AMD Zen 1 and 2 does not require executing *branch_path* to match; only the instruction addresses of the last branch target and branch source are significant for the BTB index. However, *branch_patch* is necessary for Intel BTB predictor.

if the indirect branch source addresses collide in the BTB, the CPU incorrectly speculates using the poisoned target.

On Intel Haswell, branch history is derived from the 29 last taken branches and stored as a fingerprint in the BHB. We verified that Intel Skylake, Kaby Lake and Coffee Lake use a BHB that is similar to Haswell's. The experiment primes the BHB using 29 direct branch (`jmp`) instructions. On Intel, the BHB is updated using the lower 19 bits of the previous branch source and target addresses. The attacker therefore needs to know these bits of the preceding branches, in addition to the victim indirect branch, to inject the poisoned target at the correct BTB entry. Our evaluation of an Alder Lake system shows that up to 99 branches are accounted for by the BHB.

With the same *branch_path* for training and victim, we can focus on finding indirect branch source addresses that successfully collide with our victim indirect branch source. We guess a colliding branch source address *guess_src* by brute-forcing addresses with 1–3 bits mutated across the entire address *victim_src*. The results show that an indirect branch, with bits 6–11 matching the victim's indirect branch, causes the victim to mispredict (unlike lower 12 bits as reported for Haswell [32]). This collision is also possible cross-privilege for Intel systems prior to Coffee Lake Refresh, which do not support eIBRS.

BTB collisions on AMD. There is no known study on creating BTB collisions on recent AMD platforms. We adopt the same approach as Figure 2 for creating BTB collisions on AMD. Interestingly, the same approach allows us to successfully create collisions with the victim indirect branch on AMD's Zen 1 and Zen 2 microarchitectures. Again, bruteforcing variations of the victim indirect branch, shows that *guess_src* poisons *victim_src* when mutating two or more bits. On Zen 2, we derive the combinational logic shown in Figure 3 by observing the bits mutated once a misprediction

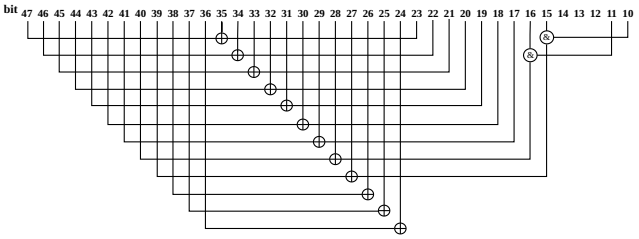

Figure 3: Combinational logic for creating BTB collisions with a victim indirect branch on AMD Zen 2 microarchitecture.

to the poisoned target occurs. By moving the victim branch source into a kernel module while keeping *guess_src* in user mode (always mutating bit 47 to stay in valid user mode address-space), we verified that this collision is possible *even across privilege boundaries*. This works by disabling SMAP and SMEP (Supervisor Mode Execution/Access Prevention) protections. However, as we show Section 6.2, this is not a requirement for exploitation.

We observe collisions on Zen 3 as well when mutating similar bits as in Zen 2. However, Zen 3 has a less reliable signal from the poisoned target, and we have to rerun the experiment several times to observe all the possible bits. We refer to Appendix A for details regarding the combinations of mutated bits that successfully collide with the victim indirect branch. Furthermore, we were unable to observe collisions across privilege boundaries on Zen 3.

Our reverse engineering further shows that the preceding 29 branches are *redundant* for crafting a valid history to poison the BTB on Zen 1 and Zen 2. In fact, a single jump instruction to an address that matches the victim's last branch target address, before the guessed indirect branch, is enough for setting the correct history to create collisions with the victim indirect branch. This essentially resembles Figure 2, but *branch_path* can be omitted. Therefore, it helps to think of the BTB index generation as a function of a "basic block", where start and end addresses together form the BTB index.

5.2 Creating mispredictions with returns

After gaining understanding of how the BTB works with indirect branches, we proceed to reverse engineer the behavior of return instructions on different microarchitectures.

Hijacking returns on Intel. To confirm that RSB underflows result in using BTB predictions for returns, we make an experiment in which we execute N nested function calls. Each function ends with a return and is allocated at a random location for every round. This way, only RSB prediction is possible, because the randomly located returns have not executed in the past, which is necessary for BTB predictions. We calculate the number of mispredicted branches and subtract the number of mispredicted calls by sampling the performance counters *br_misp_retired.near_taken* and

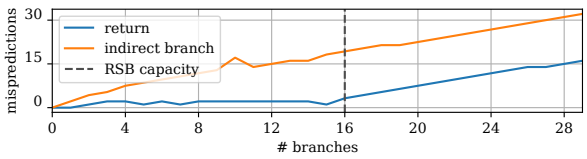

Figure 4: Return instructions behave like indirect branches instructions when reaching the threshold of more than 16 calls.

br_misp_retired.near_call before the first call and after the N^{th} return. We compare the results with the same experiment, only this time we replace each return with an indirect branch to the return target (i.e., `pop %rax; jmp *%rax`).

Figure 4 shows that for $N \geq 16$, which is the capacity of the RSB, the number of mispredicted branches increases linearly with N. Hence, once we exhaust the RSB, returns start to mispredict similar to indirect branch instructions. This experiment indicates that the BPU tries to resolve the returns through some other, apparently *near branch*, prediction scheme. The question is whether this prediction scheme uses the BTB, and whether we can hijack one of these returns by poisoning its BTB entry.

To answer this question on Intel, we need to set up a specific history, and a PC that collides with the victim return instruction. The problem is that if returns are to be treated as indirect branches, they need to be predicted by the BTB instead of the RSB. To verify whether returns can be predicted this way, we replace all 30 `jmp` instructions in the experiment described in Figure 2 with returns and check whether we can still observe the misprediction to the poisoned target. Doing so requires an additional step, where we first push all the return targets (previously jump targets) to the stack. Replacing all `jmp` instructions with returns serves two purposes: not only will returns preceding the victim branch source prime the BHB (like the replaced `jmp` instructions did), but since the number of returns exceeds the RSB capacity, we expect them to be treated as indirect branches. The results show that *returns are indeed treated as taken indirect branches*. Moreover, our results show that on our Intel platforms, we can successfully hijack all returns that come after an RSB underflow. This means that *all returns following a sufficiently deep call stack can be hijacked*.

Hijacking returns on AMD. Similar experiments on AMD Zen 1 and 2 microarchitectures show vastly different results: we observe BPU mispredictions on returns *without RSB underflow*, whenever there is a colliding indirect branch. This means that unlike Intel, *we can hijack any return, simplifying exploitation significantly*.

We want to know why we observe collisions without exhausting the RSB on AMD. We confirm that the RSB is being used (32 entries on Zen 2) by reusing the previous experiment but instead sampling the performance counter *ex_ret_near_ret_mispred*. To check whether RSB predictions are used at all when there is a BTB entry, we create a new experiment, illustrated in Figure 5, where we try to create an

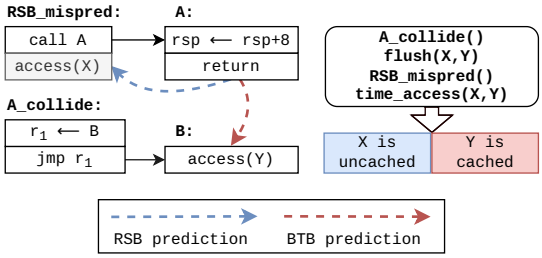

Figure 5: This experiment shows that BTB prediction takes precedence over RSB on AMD CPUs. Note that `access(X)` can never be architecturally executed due to the stack adjustment in A.

RSB and BTB misprediction on the same return at the same time. Following Koruyeh et al.'s method to create an RSB misprediction [37], we move the stack pointer ahead so that the latest return target is skipped when executing the return from A. Following our own insights from Section 5.1, we inject a BTB entry for the return in A by executing the function $A_{collide}$, which is allocated such that it matches its start and end addresses with A, except its address bits are mutated according to Figure 3. The question is whether a BTB or RSB prediction will be used when returning from A. To signal an RSB misprediction, we access memory address X at the skipped return target. To signal a BTB prediction, our injected branch target accesses memory location Y. When we run the code, we find that X is uncached and Y cached. Conversely, if we run the experiment without executing $A_{collide}$, we see the opposite: X is cached and Y is uncached. Our conclusion is that *BTB prediction takes precedence over RSB prediction.*

Summary. We showed the conditions under which return instructions can be hijacked in different microarchitectures. In the next section, we describe a framework that we built for identifying these return instructions inside the Linux kernel.

6 Exploitation Primitives

RETBLEED relies on two basic primitives: exploitable return instructions, discussed in Section 6.1, and BTI on kernel memory, discussed in Section 6.2.

6.1 Discovering exploitable returns

Because we assume the attacker can run unprivileged code, they can detect the kernel version. Assuming the victim kernel is not custom-built, the attacker can obtain a copy of victim kernel from the package repository of their Linux distribution. Hence, the following primitives are crafted during an *offline phase*, where the attacker has root privileges.

State-of-the-art speculation primitive scanners are overwhelmingly dedicated to the discovery of vulnerable *conditional branches* [25, 26, 28, 33, 43, 48, 64], since indirect branches are believed to be mitigated by compilers using

retpoline [8, 13, 45]. Return speculation is hardly ever considered as a significant attack vector. Because of the lack of tools, we construct one to discover vulnerable returns in the kernel. We first describe our framework for finding exploitable return instructions in the kernel and then show how to poison the target of these return instructions from user mode.

Exploitability of return instructions depends heavily on the microarchitecture according to our reverse engineering in Section 5. More specifically on Intel CPUs, we need to find return instructions that follow deep call stacks, resulting in RSB underflow conditions that fall back to the BTB. On AMD, even shallow return instructions are potentially exploitable.

To find vulnerable returns, we make use of the ftrace facility inside the Linux kernel [51], which allows us to observe all function call graph when executing a syscall. A *vulnerable return* provides us with arbitrarily control over its target (i.e., the instruction pointer). However, additional control is typically necessary to leak arbitrary secrets, resulting in an *exploitable return*. For example, for a FLUSH+RELOAD covert channel, we need to control two memory pointers, where one references a secret in the kernel, to be leaked, and the other is a buffer pointer (referred to as reload buffer) that can be accessed by both kernel and user. Previous work shows that the kernel's direct mapped memory can be exploited for this purpose [21].

To see whether these vulnerable returns can be exploited with FLUSH+RELOAD, we trace register and memory state at the system call entry and at the point of the vulnerable return using kprobes and kretprobes respectively [34]. Our framework, built around eBPF, ftrace, kprobes and kretprobes, can be broken down into four different steps:

Step 1: Generate test cases. We detect system calls and inputs that result in deep call stacks by running test cases in the form of small compiled binaries that perform system calls. A system call fuzzer, such as Syzkaller[1], could be modified for this purpose. However, an arguably simpler approach is to use a pre-written system call test suite. Linux Test Project[2] includes one that suits our purpose. All test cases are run as unprivileged to exclude the call stacks that require privileged access.

Step 2: Construct kernel function graphs. The Function Graph Tracer [51] is an ftrace feature that constructs call graphs of all kernel functions (with a few exceptions). For each test case, we construct function graphs for every executed system call.

Step 3: Analyze function graphs. We simulate a 16-entry RSB as we traverse the function graphs. Whenever our simulated RSB experiences an underflow condition, when returning from a deep call stack, we consider the offending return and all others that follow it as vulnerable. If there is a

[1] https://github.com/google/syzkaller
[2] https://linux-test-project.github.io/

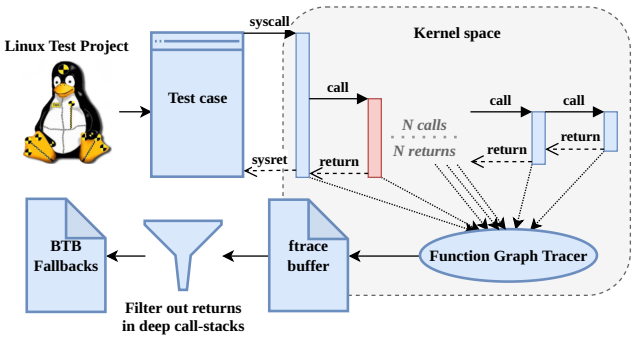

Figure 6: Finding vulnerable returns using kernel tracing tools. Let's assume that we have detected a vulnerable return in the red marked function.

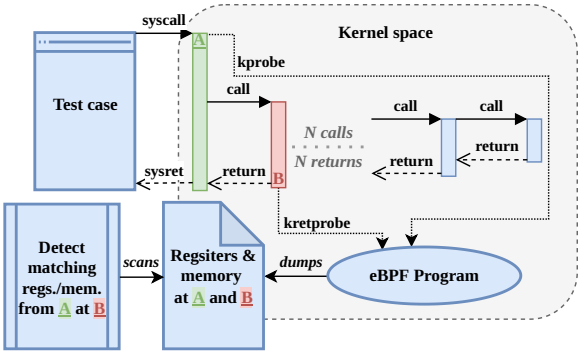

Figure 7: Detecting attacker control of registers and memory. For each vulnerable return found in the previous step, we run the same test case again, only this time, we break the control flow at the vulnerable return (at B) to dump registers and memory. We analyze the information to discover whether anything matches the system call inputs (dumped at A), which we control.

context switch in the call graph, we conservatively skip ahead to the call graph of the next system call. The first three steps are outlined in Figure 6. The function graph also informs us of how many times a particular system call and vulnerable return executed when the RSB underflow occurred.

Step 4: Detecting control for exploitability. If any of the system call inputs are present at a vulnerable return, we assume it is to some degree attacker-controlled. While methods such as symbolic execution or dynamic taint tracking could establish the attacker's control with precision, a more relaxed variant proves sufficient and works without augmenting the kernel. For each vulnerable return, we register a kretprobe at the return and a kprobe at the respective system call handler. For this purpose, we write a eBPF program that we instruct from user mode to register probes at the desired locations, and read register state, as shown in Figure 7.

If a register is a memory pointer, we also dump 96 bytes of memory that it points to. The test case may issue a particular system call more than once, and likewise it may execute the

return in a call stack where it is not vulnerable. We make use of the function graph from Step 3 to find the number of times a particular system call was made and when a function returned under an RSB underflow condition. If we find memory or register values used in the system call inputs that are also present in memory or registers at the vulnerable return, we log them if they can be altered without affecting the call stack depth. For example, we omit system call input arguments such as file descriptors, mode and flag parameters as they provide limited control.

With knowledge about what memory and registers are under the attacker's control, we next need to find an appropriate disclosure gadget which we will discuss in Section 7.1. Our framework is made out of 1301 LoC in user mode and 197 LoC of eBPF code that runs in the kernel.

6.2 BTI on kernel returns

The original Spectre-BTI attack against KVM [36] accomplished BTI by allocating executable memory in the guest kernel address space that would also be mapped in the host kernel. Branching into this address would successfully poison a host indirect branch. An unprivileged attacker does not enjoy this benefit. However, as we will discuss next, allocating executable memory with kernel addresses is not necessary.

In Section 5, we discussed how to create collisions on kernel return instructions from user mode. However, even though we can create collisions, it is not possible to speculatively execute or access unprivileged code, such as the attacker's, from kernel space due to SMEP and SMAP protections. The question is how to inject a branch target that resides in the kernel address space in the BTB.

Previously work noted that illegal branches update the BTB [36], but to the best of our knowledge, this has never been considered in the context of a User–Kernel threat model like ours before. As it turns out, *branch resolution feedback is recorded to the BTB across privilege boundaries* even if the branch target is not architecturally allowed to execute. Consequently, we can inject arbitrary branch targets into the BTB, including ones that are neither mapped nor executable in the current privilege domain. Therefore, a user program can branch into kernel memory and recover from the page fault using a signal handler as BTI primitive.

We evaluate this primitive on different microarchitectures and observe successful BTI on almost all of them (more information in Section 8). Note that, as part of our BTI primitive, we have to use the correct BTB entry. Branch history and BTB collision can be set up using the techniques we described in Section 5.

7 RETBLEED

Armed with the necessary tools to find exploitable return instructions, in this section we develop RETBLEED. For this purpose, we scan the Linux kernel for suitable disclosure gadgets (Section 7.1) and show how we can improve the signal with non-trivial disclosure gadgets that match exploitable return instructions (Section 7.2). We also show how we can adequately set up branch history as needed (Section 7.3) and bypass KASLR as a prelude to our end-to-end exploit (Section 7.4).

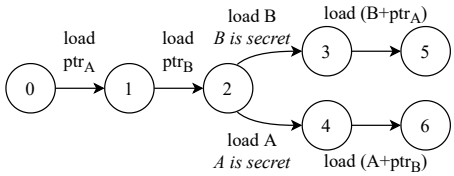

Figure 8: State transitioning of our disclosure gadget scanner

7.1 Scanning for disclosure gadgets

Gadget scanners typically focus on discovering gadgets for code-reuse attacks [30,52,53,56], and Spectre gadget scanners look for speculation primitives [25,26,26,28,33,43,45,48,64]. Because neither fits our scenario well, we construct a basic disclosure gadget scanner, specifically for Spectre-BTI.

We consider two types of gadgets, FLUSH+RELOAD and PRIME+PROBE. A PRIME+PROBE-type gadget requires a controllable input for the secret pointer that it references and uses it in a secret-dependent memory accesses. Alternatively, a FLUSH+RELOAD-type gadget requires the secret-dependent access to be in the memory that is shared with the attacker, thus requiring a reload buffer pointer as second input.

Requirements. Our BTI primitive allows arbitrary speculative code execution. However, SMAP and SMEP protections limit memory accesses and execution of the disclosure gadget to the kernel address space. Arbitrary physical memory, including attacker-owned memory, can be accessed by the kernel via direct mapped memory *physmap*, but physmap is marked as non-executable for the kernel. Hence, for the disclosure gadget, we must reuse code in the executable sections of the kernel. Furthermore, typical disclosure gadgets assume that the secret value pointer and reload buffer (for FLUSH+RELOAD) are already populated in registers. To increase the number of potential gadgets, we further need to additionally support the scenario where such pointers are present through memory dereferences as well.

Implementation. Our basic disclosure gadget scanner uses the Capstone engine[3] to disassemble executable kernel sections and discover FLUSH+RELOAD gadgets. It transitions from an initial state to a final state as shown in Figure 8. We scan for a sequence of up to 20 instructions without branches. To handle memory dereferences, additional instructions are required, which accounts for the first two state transitions. After observing two quad-word memory loads, the scanner

[3] https://www.capstone-engine.org/

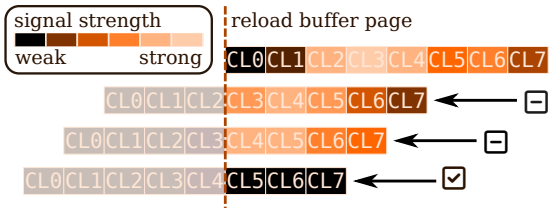

Figure 9: We continuously trigger the secret-dependent access while *sliding* cached CLs over the reload buffer page boundary by offsetting its pointer until we observe no signal. This shows us which CL was the secret-dependent access and which ones that were prefetched

has transitioned to state 2. From there, either pointer *ptrA* or *ptrB* can be dereferenced. The one that is dereferenced first is considered to be the secret pointer and the second, the reload buffer pointer. We then manually go through the discovered gadgets to find the most appropriate. Particularly, we are interested in gadgets that load 2 bytes of the secret at maximum.

7.2 Using non-trivial disclosure gadgets

The gadgets that we discover are non-trivial to exploit. We discuss some of the problems that we encounter and how we overcome them.

Handling small secret-encoding strides. An ideal disclosure gadget multiplies the secret with a value that is large enough to avoid triggering data prefetchers (e.g., 4096). Unfortunately, these are unusual to find in the kernel. A simpler gadget, such as `reload_buf + *secret`, is much easier to find. There are however two problems with such a gadget. First, data prefetchers will access neighboring cache lines (CLs) after observing the secret-dependent access. Second, we can only detect the CL that was accessed, but the lower 6 bits of the secret remain unknown.

We overcome the first problem by exploiting a known property of data cache prefetchers. Because prefetchers usually only operate within page boundaries [55], if we have multiple cached CLs as exemplified in Figure 9, we can detect the secret-dependent access by continuously triggering the disclosure gadget with the same secret while offsetting (i.e., *sliding*) the reload buffer pointer until we observe no signal anymore. As seen on the last row in the example given, all the remaining entries become uncached when *CL*4 was shifted over the page boundary. This means that the secret-dependent access was to *CL*4 and the other cached CLs (2–3 and 5–7) were prefetched.

Considering the second problem, if the secret is one byte, this merely allows us to infer the upper 2 bits. However, if the secret was multiplied by 2, 4, or 8 we infer 3, 4 or 5 bits, respectively. To infer the lower bits of the secret, we shift the reload buffer back until we observe cache hits again. There are 64 possible offsets in the CL where the access could take place. Through binary search, we need to check 16 locations to find the exact offset. If the secret was multiplied by 2, 4, or 8, we only need to check 5, 4, or 3 locations, respectively.

Word-sized secrets. A typical disclosure gadget loads only one byte of the secret, but we also consider gadgets that load word-sized (i.e., 2-byte wide) secrets. For these disclosure gadgets, we can leak only the upper byte $byte_{upper}$, of the secret, without needing to multiply the secret. $byte_{upper}$ is at bit index 8 of the register, meaning the word-sized secret can be expressed as $byte_{lower} + byte_{upper} * 256$. By knowing $byte_{lower}$, we can let $rb' = rb - byte_{lower}$. By passing rb', we will speculatively access $rb + byte_{upper} * 256$, which can be leaked by reloading cache lines with 256 bytes strides. Once we infer $byte_{upper}$, we let $byte_{lower} = byte_{upper}$ to leak the next byte. It's realistic to assume that a portion of the secret is known, for example `root:$` in */etc/shadow* [62].

Increasing the speculation window. Spectre works by winning a race condition against the memory hierarchy. As soon as the architectural branch target is resolved, the CPU stops executing in the wrong speculative path and restart at the correct one. Because our speculation primitive is a return instruction, the correct branch target, according to the Instruction Set Architecture (ISA), is allocated on the stack, which is frequently accessed and therefore is usually present in data caches. Additionally, as shown in Figure 8, non-trivial disclosure gadgets can contain 4 (or more) memory loads, which arguably require a larger speculation window to execute. Hence, we somehow need to increase the speculation window so that our injected branch target gets a chance to execute long enough to leak information.

Whereas the kernel stack pointer varies across process invocations, the lower 12 bits remain consistent, even across reboots. Because CPU cache sets are partially indexed using the lower 12 bits of memory addresses, we can use a cache eviction primitive to evict the kernel stack. Simultaneous Multi-Threading (SMT) allows executing two threads in parallel on the same physical core, where L1 and L2 caches are shared by the threads. To leverage SMT, we pin two threads to same physical core, where one is constantly attempting to evict the kernel stack by accessing memory pages at the same offset as the target stack location, thereby evicting L1d and L2. This allows us to successfully execute the entire disclosure gadget speculatively.

As a final step towards successfully exploiting the vulnerable return, we must also obtain the history of the last 29 control flow edges leading up to the vulnerable return for exploitation on Intel CPUs. Note that ftrace [51] (used in Section 6.1), only provides us with function calls and returns, and we need a different (more heavyweight) mechanism to track all control flow edges.

Restarts. We sometimes observe that the leakage stops at certain page offsets with certain disclosure gadgets. RETBLEED overcomes this problem by forking a new process after detecting it.

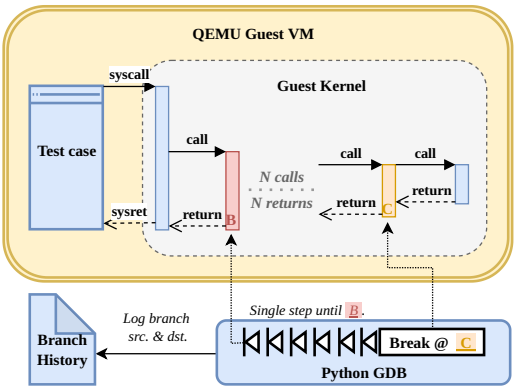

Figure 10: Gather branch history for BTI.

7.3 Obtaining branch history

We need to collect the last 29 control flow edges for exploitation on Intel. Since many of these edges are runtime-dependent, we need find to them dynamically. Intel's Last Branch Records (LBRs) are capable of recording recent control flow edges, but the limited number of LBRs makes it difficult to record all the necessary edges. For example, obtaining the LBRs at the address of a vulnerable return using a kretprobe, contributes unrelated branches that overwrites older ones from the history, before it is possible to stop recording.

Instead, we run the victim's kernel inside a VM which allows us to attach a debugger, as illustrated in Figure 10. Using a debugger, we can single step over instructions and record the source and target addresses of as many control flow edges as desired. Having obtained the function graph trace from a test case that reaches a vulnerable return, we set a breakpoint inside function that is clearly more than 29 control flow edges away from the vulnerable return. From there on, we single-step until we encounter the victim branch (using a gdb-python script), and log all control flow edges.

To avoid crashes or deadlocks, we create a list of locations (e.g., `__sysvec_apic_timer_interrupt`, `asm_call_irq_on_stack`, `psi_task_change`, `try_to_wake_up` and `check_preempt_curr`), where we continue execution until returning to a stable location where we can restart recording control flow edges again. This can cause a gap in the recorded history since prior control flow edges are discarded. Nonetheless, our strategy could always find sufficient control flow edges preceding the speculation primitive.

7.4 Derandomizing KASLR

Several derandomization attacks on KASLR have previously been proposed [11, 12, 18, 23, 27, 31, 38, 62]. These are all side-channel attacks on the CPU hardware. Despite the plenitude of these attacks, the majority is either strictly limited to Intel systems [11, 12, 31] or require further reverse engineering before they could be applied on a modern AMD CPU [18, 27, 38]. We show how our RETBLEED primitives can be used for breaking KASLR on modern AMD CPUs. On Intel, we simply use MDS [62] instead of the following steps for breaking KASLR.

```
1  ADD RCX,qword ptr [page_offset_base] ; physmap
2  MOV RAX,qword ptr [RCX]
3  MOV qword ptr [RDX],RAX
```

Listing 1: Disclosure gadget for breaking KALSR on AMD Zen 1 and Zen 2, located in *init_trampoline_kaslr* (*arch/x86/mm/kaslr.c*). `RCX` and `RDX` are attacker-controlled. Lines 1–2 are used for Steps 1 and 2. Step 3 only uses Line 3.

To use a FLUSH+RELOAD-type gadget, we need both a code pointer of the gadget location in kernel address space and a kernel pointer to our reload buffer via *physmap*, the direct mapped memory address space. Ubuntu by default allows allocation of *transparent huge pages* (thp). Therefore, with 32 GiB of physical memory, according to recent work [38], the cumulative entropy to find our reload buffer using a gadget at the unknown location of the kernel image is $entropy_{kern} + entropy_{physmap} + entropy_{phys_{reloadbuf}} = 9+15+14 = 38$ bits, which is too large to bruteforce directly. Instead, we will reduce this entropy in the following three steps. For the speculation primitive on AMD, we target a vulnerable victim return inside the `mmap` system call handler, which we discuss in Section 8.3.

① **Derandomizing kernel image.** We conduct a PRIME+PROBE attack on a single L1d set. The goal is to evict memory from the set by misprediction to the disclosure gadget in Listing 1. Since both the victim return and disclosure gadget are relative to the kernel image, correctly guessing the kernel image consequently forces the victim return to mispredict into the disclosure gadget.

For each guess, we poison the BTB entry of the victim return with the disclosure gadget, prime the cache set and make the system call. When the system call returns, we probe our primed cache set to detect the memory access. Knowing the kernel image location reduces the entropy by 9 bits.

② **Leaking a physical memory pointer.** We use FLUSH+RELOAD to detect the physical address of our reload buffer. Our disclosure gadget dereferences physmap using an attacker-controlled register as offset. We allocate our reload buffer as a transparent huge page so that it is aligned on a 2 MiB boundary in physical memory. Using the attacker-controlled register, we try all 2 MiB aligned physical memory addresses that the reload buffer can be assigned to.

For each guess, we flush the first reload buffer cache line, poison the BTB entry of the victim return, execute the system call, and time the access to the same reload buffer cache line. A cache hit means we have guessed the correct physical address. Knowing the physical address of the reload buffer reduces the entropy by 14 bits.

③ **Finding physmap.** All we need is a single-instruction

Table 1: RETBLEED primitives and leakage rate with ideal gadgets on various Intel and AMD microarchitectures.

CPU	Microarch.	Microcode	Year	Defense [c]	Primitive			Bandwidth	Success rate
					RET-BTI [a]	U → K [b]	RETBLEED		
AMD Ryzen 5 1600X	Zen 1	0x8001138	2017	retpoline$_{AMD}$	✓	✓	✓	18.4 kB/s	99.5 %
AMD Ryzen 5 2600X	Zen 1+	0x800820d	2018	retpoline$_{AMD}$	✓	✓	✓	22.5 kB/s	99.8 %
AMD Ryzen 5 3600X	Zen 2	0x8701021	2019	retpoline$_{AMD}$	✓	✓	✓	20.6 kB/s	96.7 %
AMD Ryzen 7 PRO 4750U	Zen 2	0x8600106	2019	retpoline$_{AMD}$	✓	✓	✓	17.1 kB/s	99.0 %
AMD EPYC 7252	Zen 2	0x8301038	2019	retpoline$_{AMD}$	✓	✓	✓	14.8 kB/s	99.9 %
AMD Ryzen 5 5600G	Zen 3	0xa50000c	2020	retpoline$_{AMD}$	✗	✗	✗	–	–
Intel Core i7-7500U	Kaby Lake	0xea	2016	retpoline	✓	✓	✓	3.6 kB/s	77.0 %
Intel Core i7-8700K	Coffee Lake	0xea	2017	retpoline	✓	✓	✓	5.9 kB/s	83.1 %
Intel Core i9-9900K	Coffee Lake Ref.	0xea	2018	eIBRS	✓	✗	✗	–	–
Intel Core i9-12700K	Alder Lake	0xd	2021	eIBRS	✓	✗	✗	–	–

[a] BTI on returns; *out-of-place* [10]; [b] User → Kernel boundary BTI; [c] AMD-style retpoline is referred to as `RETPOLINE_LFENCE` since Kernel 5.14.

disclosure gadget that dereferences a register that we control. We follow the same procedure as in Step 2, but this time, we guess the possible locations of physmap by passing $guess_{physmap} + phys_{reloadbuf}$ in our register. In each attempt, we load our reload buffer. Detecting a cache hit means we have guessed the correct physmap address and reduces the remaining entropy to 0.

Results. We can break KASLR in 60 ms with 97.5% accuracy. We find the physical address of the reload buffer at physical address 6 GiB into physical memory after 16 seconds. Finding the physmap base address takes about 1 second. After breaking KASLR and finding the physmap address of our reload buffer, we can leak memory using a FLUSH+RELOAD-type disclosure gadget.

8 Evaluation

We first evaluate the attack primitives we designed in Section 6 on a variety of microarchitectures from Intel and AMD running on Ubuntu 20.04 with Linux kernel 5.8. We then evaluate the end-to-end RETBLEED attack on Intel Coffee Lake (bypassing generic retpoline) and AMD Zen 2 (bypassing AMD retpoline) microarchitectures.

8.1 RETBLEED primitives

We verify the existence of two required RETBLEED primitives on different microarchitectures. First primitive RET-BTI, is to perform BTI on returns. The second primitive U → K BTI, is to inject branch targets on kernel branches from user mode. To accurately evaluate these, we construct a simple kernel module with an ideal victim return instruction and disclosure gadget. For end-to-end exploitation in the next sections, we use returns and disclosure gadgets within the kernel image.

As seen in Table 1, Intel Kaby Lake and Coffee Lake microarchitectures are susceptible to both primitives. On Coffee Lake Refresh and Alder Lake, while we can hijack return instructions, we did not observe a signal using our U → K BTI primitive, even after disabling eIBRS. Therefore,

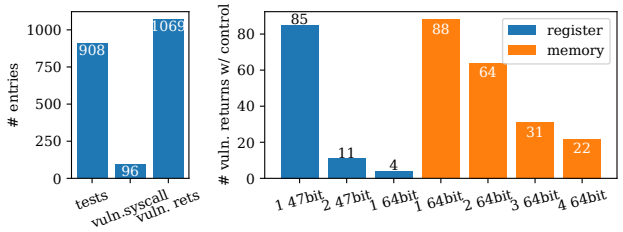

Figure 11: Breakdown of the vulnerable returns discovered. Totally 908 tests cases were run. We found 1069 vulnerable returns across 96 system calls.

we cannot confirm whether eIBRS stops our primitive or if it is ineffective for other reasons. On AMD platforms, our primitives are successful on Zen 1, Zen 1+, and Zen 2 microarchitectures. On AMD Zen 3, we were only able to observe BTI through indirect branches in the same privilege mode, thus is not susceptible using our method (see Section 5).

When repeatedly leaking 4096 bytes of randomized memory with an ideal victim return and disclosure gadget, we measure the covert channel bandwidth and accuracy of 5.9 kB/s and 83.1 % on Intel Coffee Lake, and 20.6 kB/s and 96.7 % on AMD Zen 2, respectively.

8.2 RETBLEED on Intel

Vulnerable returns. We used our framework to analyze the entire syscall test suite from Linux Test Project to search for vulnerable return instructions. Out of 1363 test cases, an unprivileged user can run 908. The results are shown in Figure 11. We discovered 1069 vulnerable return instructions reached from 96 different system calls in total. For four vulnerable returns, we had partial control over a full 64 bit register. For three of which, the register contained a file name, thus refusing NULL (`0x00`) and forward slash characters (`0x2f`), and the fourth case could not be reached consistently. For 85 different vulnerable returns, we had control over 47 bits of a register with the upper bits unset, constrained to a valid user-space pointer. For 11 cases, we had control over two such registers.

We could detect substantially more control if we also resolved pointers to compare memory referenced by registers at vulnerable returns. We found 88 vulnerable returns with control over at least a 64 bit block of memory through one indirection. For 64 for which we had control over 2 or more 64 bit blocks, which satisfies the requirements of a FLUSH+RELOAD gadget. We decided to target the *sendto* system call, since in the test case *recvmsg02*, it led to two vulnerable returns, where the entire message buffer was present through `R14+0x8`. These returns were discovered in functions in `ip6_local_out` *(net/ipv6/output_core.c)* and `ip6_send_skb` *(net/ipv6/ip6_output.c)*.

```
MOV   RAX,qword ptr [R14 + 0x28]
MOV   RDX,qword ptr [R14 + 0x20]
MOV   dword ptr [RDX + 0x1c],0xffffffff ; bogus
MOVZX EAX,byte ptr [RAX + 0x14]
ADD   RAX,0x1 ; secret + 1
MOV   ECX,dword ptr [RDX + RAX*0x4 + 0x58]
```

Listing 2: Disclosure gadget found in `mb_mark_used` *(fs/ext4/mballoc.c)*. `&R14[8]` holds the second argument, `*buf`, of the syscall `sendto`. The secret and reload buffer pointers are loaded into `RAX` and `RDX` respectively.

Disclosure gadget. With the fairly loose constraint of having control over buffer contents pointed to by `R14`, we run our disclosure gadget scanner. We discover a compatible gadget that loads a byte-sized secret, shown in Listing 2. Looking at Listing 2, the following memory address is accessed

$$rb' + (secret + 1) * 4 + 0x58$$

Because we freely control the reload buffer address rb', offsetting it by $-0x5c$ the memory address becomes $rb + 4 * secret$. This gadget is non-trivial to exploit, since the secret is only multiplied by 4. To leak arbitrary data using it, we use the sliding method discussed in Section 7.2.

Results. We leak 1 kB of kernel memory 10 times with an average accuracy and covert channel bandwidth of 98 % and 219 bytes per second respectively. We furthermore verified that RETBLEED is capable of locating and leaking the root password hash from */etc/shadow* in physical memory in 28 minutes on a Coffee Lake system with 16 GiB of RAM.

8.3 RETBLEED on AMD

The AMD Zen 1 and Zen 2's branch predictors are easier to mistrain than their Intel counterparts. Neither underflowing the RSB, nor generating a perfectly matching branch history is necessary. This means that virtually all returns are vulnerable.

Vulnerable returns. Because of the looser constraints for AMD, our framework finds many vulnerable returns. Specifically, we found suitable vulnerable returns on early error paths in system calls. Because these error paths are easily reached (e.g., by providing an invalid argument), the syscall inputs are often untouched. In particular, `mmap` and `mremap` have all six arguments present in the registers at the vulnerable return instructions. Using one of these returns, we can fully control `RDI`, `RSI`, `RDX`, `RCX`, `R08` and `R09`. With such control, finding disclosure gadgets is trivial.

Disclosure gadget. With the `mmap` syscall, the attacker can control 6 registers. We found a disclosure gadget that loads a word-sized secret, as shown in Listing 3. It uses the `RDI` and `RSI` registers, which are among the 6 attacker-controllable registers. As before, the secret is not shifted enough to leak the secret. However, by using the method described in Section 7.2, exploiting this gadget becomes practical.

```
MOVZX EDX,word ptr [RDI + 0x7c]
MOV   R14D,dword ptr [RDI + 0x70] ; bogus
MOV   R13,qword ptr [RSI + RDX*0x8 + 0x900]
```

Listing 3: Disclosure gadget found in `tun_net_xmit` *(drivers/net/tun.c)*. `RDI` points to the secret and `RSI` points to the reload buffer

Results. We leak 4 KiB of kernel memory 100 times. 20 times the attack was unsuccessful due to noise. For the remaining 80 times, we leak with a median accuracy and bandwidth of >99 % and 3.9 kB per second respectively. We furthermore verified that RETBLEED is capable of locating and leaking the root password hash from */etc/shadow* in physical memory in 6 minutes on a Zen 2 system with 64 GiB of RAM.

9 Mitigation

We saw that retpoline-protected Intel and AMD CPUs are vulnerable to RETBLEED. We will now discuss the possible directions of mitigating RETBLEED, specifically we consider mitigations by means of preventing speculation and isolation. Furthermore, Intel and AMD have shared with us their current mitigation strategies, which we also discuss and evaluate.

9.1 Preventing speculation

Retpoline, as a Spectre-BTI mitigation, fails to consider return instructions as an attack vector. While it is possible to defend return instructions by adding a valid entry to the RSB before executing the return instruction, treating every return as potentially exploitable in this way would impose a tremendous overhead. Previous work attempted to conditionally refill the RSB with harmless return targets whenever a per-CPU counter that tracks the call stack depth reaches a certain threshold, but it was never approved for upstream [35]. In the light of RETBLEED, this mitigation is being re-evaluated by Intel, but AMD CPUs require a different strategy.

AMD's mitigation. AMD proposed a mitigation, called *jmp2ret*, which prevents speculation by replacing returns in the kernel with direct jumps to a return thunk. This is used to

Table 2: Unixbench overhead with AMD and Intel mitigations.

Microarch.	Coffee Lake	Zen 1+	Zen 1+ (no SMT)	Zen 2
Overhead	16.84 %	5.78 %	27.90 %	11.94 %

return from all function calls, and it narrows down the attack surface to a single return located in the return thunk. To mitigate the remaining attack surface, an *untrain* procedure, executed at kernel entry, jumps to a byte that immediately precedes the return in the return thunk. Consequentially, this byte gets encoded by the CPU as a different instruction, which invalidates the BTB entries associated with the return instruction. Unlike IBPB, jmp2ret leaves the remaining BTB intact. Furthermore, to prevent BTI from a sibling thread, right after untrain, on affected AMD systems that do not support STIBP [16] (e.g., Zen 1+), SMT must be disabled.

9.2 Isolation

We observed that eIBRS systems seem protected from RETBLEED. IBRS, which is the alternative for earlier systems, was considered too expensive to use in practice when Spectre was introduced in 2018. Flushing the entire BTB through IBPB upon kernel entry would arguably be even more expensive.

Intel's mitigation. Despite the performance cost of IBRS, it mitigates RETBLEED on vulnerable Intel CPUs. Hence, IBRS will selectively be enabled on systems that exhibit RSB-to-BTB fallback behavior that do not support eIBRS.

We initially considered conditional IBPB as a feasible alternative. The key component of RETBLEED is the $U \rightarrow K$ BTI primitive, which triggers a page fault in the kernel caused by executing a kernel address that is outside the range of valid user program memory. At this stage we "untrain" by issuing IBPB. Because invalid memory accesses to kernel memory that originate in user mode is exceptional behavior for normal applications, the performance implication is negligible. Unfortunately, this mitigation turned out to be incomplete, since the page fault of the $U \rightarrow K$ BTI primitive can be suppressed by training speculatively. We verified on Coffee Lake that such a variant of RETBLEED indeed works.

9.3 Security and performance of mitigations

We verified that the proposed patches indeed stop RETBLEED on vulnerable Intel and AMD systems. To evaluate the performance overhead, we use Unixbench [4].

Table 2 shows the incurred performance overhead derived from the geometric mean of the median of 10 invocations of each workload in multi-threaded mode. While the performance overhead is generally significant across the board, Zen1(+) systems suffer the most due to the need for disabling SMT.

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

Table 3: Cross privilege domain Spectre attacks.

Attack	**Variant [10]**	**Domain**	**Assumption**
KVM Attack [36]	BTB-OOP	Guest → Host	Privileged guest
eBPF Attack [36]	PHT-IP	User → Kernel	Unprivileged eBPF
Blindside [21]	PHT-IP	User → Kernel	Memory corruption
BHI [7]	BTB-OOP	User → Kernel	Unprivileged eBPF
RETBLEED	BTB-OOP	User → Kernel	Unprivileged user

10 Related work

Attacking the BPU. Initial security research used branch predictors for leaking secret keys [3, 4]. Later work by Evtyushkin et al. [18, 19] explored alternative attacks for breaking ASLR or leaking secrets from SGX enclaves. This past work laid out the stepping stones for Spectre attacks.

Spectre. Kocher et al. published Spectre [36], with extensive BTB reverse engineering by Horn [32]. While their contributions include speculative execution on top of conditional and indirect branches, they also correctly predicted that further attacks can be designed through mispredicted return instructions, via the RSB, as was later shown in [37, 40, 67]. Conversely, RETBLEED uses returns to target the BTB, which has never been shown in previous work.

There are only a few Spectre attacks where the attacker is in a lower privilege domain than the victim [7, 21, 36]. We enumerate these in Table 3. The original Spectre attack features Spectre-BTI in a guest-to-host attack on KVM [36]. Göktas et al. [21] showed that speculative probing can be combined with a memory corruption vulnerability, allowing them to break fine-grained KASLR and craft a code-reuse attack against the kernel. In contrast to their threat model, RETBLEED assumes no software bugs while providing arbitrary leakage of kernel data.

Concurrently to our work, Branch History Injection (BHI) [7] explored the limitations of eIBRS and found that indirect branch speculation can be hijacked to run previously executed indirect branch targets in the kernel. Whereas RETBLEED targets Intel CPUs without eIBRS and certain AMD CPUs, BHI targets Intel CPUs with eIBRS and ARM. Moreover, because of the limited number of disclosure gadgets available to BHI, unprivileged eBPF is required for practical exploitation, which consequentially has been disabled as a mitigation. It is possible to combine RETBLEED with BHI. While a future RETBLEED-BHI appears to be mitigated for platforms that support RRSBA controls (Restricted RSB Alternative) [29], evaluating this attack vector on other systems remains an interesting direction for future research.

AMD-specific Spectre research, concurrent to ours, has

shown that the BTB indexes conditional branch targets [65], and that AMD CPUs are vulnerable to Straight-line Speculation attacks [6, 66].

Closely related to Spectre are fault-based transient execution attacks, such as Meltdown [39], which along with Spectre [36], were the first transient execution attacks able to leak arbitrary privileged memory. Fault-based transient execution attacks that exploit CPU faults rather than branch predictors are plenty [9, 11, 49, 50, 54, 61, 62]. Canella et al. [10] provided a systematization of all types of transient execution attacks. They classify different Spectre attacks by their training methods. For example, training is either on the same virtual memory address as the victim branch *in-place* (IP), or on a different one *out-of-place* (OOP), where IP is more constrained than OOP. Our work exclusively focuses on OOP training.

Spectre vulnerability detection. Locating code vulnerable to transient execution attacks, such as Spectre, is ongoing research. Initial work included compiler passes to detect and remove speculation primitives [8, 13, 28, 45, 64]. Other methods such as using symbolic execution [25, 26], dynamic taint analysis [48] or fuzzing [43] have also been explored. Recent work, Kasper [33], combines fuzzing with dynamic taint analysis to detect transient execution primitives. Despite the variety of contributions in this category, focus on Spectre-BTI is lacking. Our work complements these by discovering return instructions that can hijack the speculative control flow. Moreover, our disclosure gadget scanner finds branch targets that can be used to disclose memory.

11 Conclusion

We showed how return instructions can be hijacked to achieve arbitrary speculative code execution under certain microarchitecture-dependent conditions. We learned these conditions by reverse engineering the previously-unknown details of indirect branch prediction on Intel and AMD microarchitectures and its interaction with the RSB. We found many vulnerable returns under these conditions, using a new dynamic analysis framework which we built on top of standard Linux kernel testing and debugging facilities. Furthermore, we showed that an unprivileged process can control the destination of these kernel returns by poisoning the BTB using invalid architectural page faults. Based on these insights, our end-to-end exploit, RETBLEED, can leak arbitrary kernel data as an unprivileged process running on a system with the latest Linux kernel with all deployed mitigations enabled. Our efforts led to deployed mitigations against RETBLEED in the Linux kernel.

Acknowledgments

We thank our reviewers for their feedback, particularly Yuval Yarom for shepherding our paper. Finn de Ridder contributed to the initial version of our gadget scanner. Jean-Claude Graf evaluated a speculative version of RETBLEED's BTI primitive. We also want to thank AMD, Intel and Linux kernel developers for working transparently with us and sharing insights into their mitigation strategies.

References

[1] The Chromium Projects: Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation/. Accessed on 29.1.2022.

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

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

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*. Springer, 2007.

[5] AMD. Amd64 technology indirect branch control extension. https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf, 2018. Accessed on 7.6.2022.

[6] ARM. Straight-line speculation. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Security%20Update%2008%20June%202020/Straight-line_Speculation-v1.0.pdf, 2020. Accessed on 7.6.2022.

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

[8] Richard Biener. LWN.net: GCC 7.3 released. https://lwn.net/Articles/745385/, 2018. Accessed on 7.6.2022.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *SEC*. USENIX, 2018.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A

Systematic Evaluation of Transient Execution Attacks and Defenses. In *SEC*. USENIX, 2019.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*. ACM, 2019.

[12] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *AsiaCCS*. ACM, 2020.

[13] Chandler Carruth. Introduce the "retpoline" x86 mitigation technique for variant 2 of the speculative execution... https://reviews.llvm.org/D41723, 2018. Accessed on 7.6.2022.

[14] Intel Corp. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, 2018. Accessed on 7.6.2022.

[15] Intel Corp. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf, 2018. Accessed on 7.6.2022.

[16] Intel Corp. Speculative Execution Side Channel Mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, 2018. Accessed on 7.6.2022.

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

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

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

[20] Anders Fogh. In debt to Retpoline. https://cyber.wtf/2018/02/13/in-debt-to-retpoline/, 2018. Accessed on 7.6.2022.

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

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

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *CCS*. ACM, 2016.

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[25] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *S&P*. IEEE, 2020.

[26] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. In *ICSE*. ACM, 2020.

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *S&P*. IEEE, 2013.

[28] Open Source Security Inc. Respectre: The state of the art in spectre defenses. https://grsecurity.net/respectre_announce, 2018. Accessed on 7.6.2022.

[29] Intel. Branch history injection and intra-mode branch target injection / cve-2022-0001, cve-2022-0002 / intel-sa-00598. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html, 2022. Accessed on 7.6.2022.

[30] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *SIGSAC*. ACM, 2018.

[31] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *CCS*. ACM, 2016.

[32] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018. Accessed on 7.6.2022.

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

[34] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel Probes (Kprobes). https://www.kernel.org/doc/html/latest/trace/kprobes.html, 2021. Accessed on 7.6.2022.

[35] Andi Kleen. LKML.ORG: [PATCH 1/4] x86/retpoline: Add new mode RETPOLINE_UNDERFLOW. https://lkml.org/lkml/2018/1/12/609, 2018. Accessed on 7.6.2022.

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

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

[38] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *EuroS&P*. IEEE, 2020.

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

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

[41] Matt Miller, Anders Fogh, and Christopher Ertl. Wrangling with the Ghost: An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities. BlackHat, 2018.

[42] 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 15.5.2022.

[43] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *SEC*. USENIX, 2020.

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

[45] Andrew Pardoe. Spectre mitigations in msvc. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/. Accessed on 7.6.2022.

[46] Filip Pizlo. What spectre and meltdown mean for webkit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018. Accessed on 7.6.2022.

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

[48] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 2021.

[49] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *SEC*. USENIX, 2021.

[50] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*. IEEE, 2021.

[51] Steven Rostedt. Linux kernel: ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. Accessed on 29.1.2022.

[52] Jonathan Salwan. Ropgadget-gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/, 2011. Accessed on 7.6.2022.

[53] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *CCS*. ACM, 2020.

[54] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[55] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *CCS*, pages 131–145. ACM, 2018.

[56] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *S&P*. IEEE, IEEE, 2016.

[57] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient

high-performance x86-64 microprocessor core. In *ISSCC*. IEEE, 2020.

[58] Linus Torvalds. LKML.ORG: Re: [PATCH 0/7] IBRS patch series. https://lkml.org/lkml/2018/1/4/720, 2018. Accessed on 7.6.2022.

[59] Linus Torvalds. LKML.ORG: Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lkml.org/lkml/2018/1/21/192, 2018. Accessed on 7.6.2022.

[60] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. Accessed on 7.6.2022.

[61] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*. IEEE, 2020.

[62] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[63] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018. Accessed on 7.6.2022.

[64] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[65] Pawel Wieczorkiewicz. The amd branch (mis)predictor: Just set it and forget it! https://grsecurity.net/amd_branch_mispredictor_just_set_it_and_forget_it, 2022. Accessed on 7.6.2022.

[66] Pawel Wieczorkiewicz. The amd branch (mis)predictor part 2: Where no cpu has gone before (cve-2021-26341). https://grsecurity.net/amd_branch_mispredictor_part_2_where_no_cpu_has_gone_before, 2022. Accessed on 7.6.2022.

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

[68] Dan Williams. LKML: [PATCH v6 02/13] array_index_nospec: sanitize speculative array dereferences. https://lore.kernel.org/lkml/151727414808.33451.1873237130672785331.stgit@dwillia2-desk3.amr.corp.intel.com/, 2018. Accessed on 7.6.2022.

[69] David Woodhouse. LWN.net: Re: [RFC 09/10] x86/enter: Create macros to restrict/unrestrict Indirect Branch Speculation. https://lwn.net/Articles/745113/, 2018. Accessed on 7.6.2022.

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

A Collisions detected on AMD Zen 1, 2 and 3

We denote folding bits as *functions*. A collision occurs for two addresses A and B when $f_i(A) = f_i(B)$ for all i.

Zen 1. We obtain the following functions.

$$
\begin{array}{ll}
f_1 = b_{39} \oplus b_{30} \oplus b_{21} & f_2 = b_{40} \oplus b_{31} \oplus b_{22} \\
f_3 = b_{41} \oplus b_{32} \oplus b_{23} & f_4 = b_{42} \oplus b_{33} \oplus b_{24} \oplus (b_{15} \wedge b_9) \\
f_5 = b_{43} \oplus b_{34} \oplus b_{25} \oplus (b_{16} \wedge b_{10}) & f_6 = b_{44} \oplus b_{35} \oplus b_{26} \oplus (b_{17} \wedge b_{11}) \\
f_7 = b_{45} \oplus b_{36} \oplus b_{27} \oplus b_{18} & f_8 = b_{46} \oplus b_{37} \oplus b_{28} \oplus b_{19} \\
f_9 = b_{47} \oplus b_{38} \oplus b_{29} \oplus b_{20} &
\end{array}
$$

Zen 2. With the following, patterns we observed BTB collisions on Zen 2.

$$
\begin{array}{ll}
f_1 = b_{36} \oplus b_{24} & f_2 = b_{37} \oplus b_{25} \\
f_3 = b_{38} \oplus b_{26} & f_4 = b_{39} \oplus b_{27} \oplus (b_{15} \wedge b_{10}) \\
f_5 = b_{40} \oplus b_{28} \oplus (b_{16} \wedge b_{11}) & f_6 = b_{41} \oplus b_{29} \oplus b_{17} \\
f_7 = b_{42} \oplus b_{30} \oplus b_{18} & f_8 = b_{43} \oplus b_{31} \oplus b_{19} \\
f_9 = b_{44} \oplus b_{32} \oplus b_{20} & f_{10} = b_{45} \oplus b_{33} \oplus b_{21} \\
f_{11} = b_{46} \oplus b_{34} \oplus b_{22} & f_{12} = b_{47} \oplus b_{35} \oplus b_{23}
\end{array}
$$

Zen 3. For Zen 3, we found the following patterns. Note that b_{47} is not included because we were unable to create cross privilege domain BTI on this microarchitecture. Moreover, we only f_{8-12} under certain conditions. More work is necessary to fully understand Zen 3 BTB index generation.

$$
\begin{array}{ll}
f_1 = b_{44} \oplus b_{32} \oplus b_{20} & f_2 = b_{45} \oplus b_{33} \oplus b_{21} \\
f_3 = b_{46} \oplus b_{34} \oplus b_{22} & f_4 = b_{35} \oplus b_{23} \\
f_5 = b_{36} \oplus b_{24} & f_6 = b_{37} \oplus b_{25} \\
f_7 = b_{38} \oplus b_{26} & f_8 = b_{27} \wedge b_{15} \wedge b_6 \\
f_9 = b_{40} \oplus b_{28} \oplus (b_{16} \wedge b_7) & f_{10} = b_{41} \oplus b_{29} \oplus (b_{17} \wedge b_8) \\
f_{11} = b_{42} \oplus b_{30} \oplus (b_{18} \wedge b_9) & f_{12} = b_{43} \oplus b_{31} \oplus (b_{19} \wedge b_{10})
\end{array}
$$

Zen 3 was more difficult to infer the exact patterns on than Zen 1 and Zen 2. For example, we only found f_{1-7} to work independently. For f_{8-12}, we only found them working in combination with other functions as shown below.

$$
\begin{array}{lll}
f_8 \wedge (\neg f_{11} \wedge \neg f_{12}), & f_9 \wedge \neg f_{12}, & f_{10} \wedge \neg f_1, \\
f_{11} \wedge \neg f_2, & f_{12} \wedge \neg f_3 &
\end{array}
$$