2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00046

You Cannot Always Win the Race: Analyzing mitigations for branch target prediction attacks

Alyssa Milburn[§]
Intel[†]
alyssa.milburn@intel.com

Ke Sun[§]
Intel[†]
ke.sun@intel.com

Henrique Kawakami
Intel[†]
henrique.kawakami@intel.com

***Abstract*—Many recent attacks such as Branch Target Injection (BTI, aka Spectre v2) take advantage of speculative execution in modern processors, and in particular the inherent race condition between transient execution of code at the predicted target of a branch and the architectural resolution of the branch. This can create a speculation window in which code can be transiently executed at an unintended target, and mitigations for these attacks often focus on minimizing or removing such windows. By investigating the potential sources of latency that may contribute to such a speculation window, such as pipeline contention and simultaneous multithreading (SMT) activity, we show that an attacker can "win the race" despite the adoption of widely-used mitigations, on a variety of different x86 CPUs. We also show that such speculation windows may be present for predictions of *direct* branches. This enables a new class of BTI-style attacks that do not depend on indirect branches, and bypass the majority of previous mitigations against such attacks.**

1. Introduction

Since the disclosure of Spectre [27], there has been a wide range of research on similar transient execution attacks which rely on speculative execution of the predicted targets of branches [9], [14], [19], [28], [31], [35], [42], [47]. Branch target predictions allow processors to predict the targets of branches before the actual targets are known to the pipeline, steering speculative execution to locations of those predicted branch targets. The resolution of indirect branch targets may be significantly delayed, since they may depend on memory accesses or computation. This latency in calculating the target of indirect branches is a key part of attacks such as Branch Target Injection (BTI, aka Spectre variant 2), where an attacker causes an indirect branch to be predicted to a target of their choice.

Unlike indirect branches, direct branches do not have any data dependency (such as register or memory contents), which means that direct branch targets can be computed based solely on information encoded in the branch instruction, minimizing any potential speculation window. However, even though the targets of direct branches can be computed without such latency, the direction of conditional branches may still be mispredicted ("taken" vs. "not taken"), resulting in attacks such as in Bounds Check Bypass (BCB, known as Spectre variant 1 [19]). Both classes of attack typically involve an attacker attempting to cause the transient execution of code which can reveal sensitive information across security domains via an incidental channel, such as the cache state.

Instructions at the predicted target of a branch may be speculatively fetched, decoded and even executed before an incorrect target prediction can be detected and execution can be redirected (when necessary). Mitigations for branch target prediction attacks attempt to prevent the transient execution of code at attacker-controlled targets. Many of these mitigations focus on suppressing the windows in which code could be transiently executed – which we refer to as the *speculation window*.

Mitigations for indirect branch prediction attacks include software techniques such as retpoline [40] and LFENCE/JMP [3], as well as hardware mitigations such as Indirect Branch Restricted Speculation (IBRS) and the Indirect Branch Predictor Barrier (IBPB) [2], [21]. Although direct branch targets are also predicted on modern processors [15], [24], [47], mitigations typically assume that branch target prediction attacks using direct branches are not possible, although researchers have previously expressed concerns about such assumptions [37].

In this work, we argue that the design and evaluation of existing mitigation approaches may not have assessed whether speculation windows are sufficiently suppressed, and that factors such as *branch latency* and SMT (multithreaded) contention must be considered by these mitigations. To support this claim, we present a systematic methodology for evaluating the effectiveness of such mitigations. Using this methodology, we present three case studies investigating existing BTI mitigations that have been widely deployed on x86 processors. Each of these studies demonstrates that – despite the use of these mitigations – 'universal read gadgets' [33] can be transiently executed within the remaining speculation windows.

Our first case study analyzes the LFENCE/JMP software mitigation (aka the "AMD retpoline" [7]), which was recommended by AMD as an effective retpoline alternative and had been adopted by major OS and hypervisor vendors. We demonstrate that the remaining window for speculative execution on various x86 CPUs from both Intel and AMD, despite use of the LFENCE/JMP mitigation, can still allow the transient execution of disclosure gadgets. We also show that SMT workloads can significantly increase the size of such speculation windows.

Our second case study shows that BTI-style attacks using *direct* branches – which we refer to as Direct Branch Target Injection (DBTI) – *are* possible on some AMD pro-

§. Equal contribution joint first authors.

†. This paper describes work by individual researchers within Intel STORM. It is not intended as documentation or guidance.

DOI 10.1109/EuroSP57164.2023.00046

cessors. This new class of BTI-style attacks (which AMD have now documented as part of Branch Type Confusion (BTC) [8]) are outside the scope of most BTI mitigations as they focus solely on indirect branch predictions. Again, we show that we can expand the speculation window for DBTI using SMT workloads, allowing transient execution of some 'universal read gadgets' requiring two dependent loads. Our analysis also provides insight into the behavior observed by concurrent research on conditional branch predictions on AMD CPUs [41] as well as other Branch Type Confusion [8] attacks such as Retbleed [42] – whose authors later observed that DBTI-style behavior can even occur *without* branches being present [43].

In our third case study, we investigate the IBPB hardware-based mitigation, and show that our systematic approach can be valuable even in cases where a speculation window is already known to be present. We find that, on some CPUs, IBPB is insufficient to mitigate attacks using predicted return targets – potentially exposing software to SpectreRSB [28] attacks – and that SMT workloads can induce the same behavior, even on newer processors.

Finally, we discuss how these identified issues might enable transient execution attacks as well as potential obstacles for practical exploitation. We also present a proof-of-concept attack against the Linux kernel when using the LFENCE/JMP mitigation (via unprivileged eBPF), and discuss alternative approaches for mitigating such attacks. We engaged in coordinated disclosure for all these issues, and updated mitigations have already been deployed.

Our work shows that, where BTI-style attacks are a concern, mitigations for branch target prediction attacks should be evaluated in a more systematic and comprehensive manner.

Contributions. In summary, our contributions include:

- We analyze the design of branch target prediction mitigations, identify additional factors (such as sources of latency) which can render such mitigations ineffective, and propose an approach for evaluating their effectiveness.
- We present three case studies covering different types of branches and mitigations, and show that a variety of branch target prediction attacks remain possible on many x86 CPUs, despite the use of existing mitigations.
- We describe a proof-of-concept attack on the Linux kernel to demonstrate that such attacks are practical, discuss the remaining obstacles for practical exploitation, and review potential alternative mitigations (some of which are now standard).

2. Background

In this section, we provide a brief summary of branch predictors and branch target prediction attacks. We also describe the standard mitigations for such attacks (both hardware- and software-based).

2.1. Branch predictors

Modern processors use several different branch target predictors. Both indirect and direct branch targets are usually predicted using a 'cache' in the form of the Branch Target Buffer (BTB). The BTB stores previous targets for each taken branch, and predicts that later executions of that branch will have the same target. In practice, the BTB is typically indexed by a hashed version of the branch instruction's address ('IP-based'). The Branch History Buffer (BHB) provides similar predictions using a hash of branch history ('history-based'), providing the benefit of context for branches with dynamic targets.

Return branch targets are generally predicted using a specialized stack-based predictor, which maintains a record of previous callsites – when a return instruction is encountered, the predictor 'pops' the target for the corresponding call. Examples include the Return Stack Buffer (RSB) on Intel CPUs, and the Return Address Stack (RAS) on AMD CPUs.

These predictors are typically documented by the optimization guides for modern processors [4], [24], and many of the parameters for specific processors have been determined by previous research [9], [42], [47].

We focus on three categories of branches in this work, all of which are typically predicted by some of these predictors:

Indirect branches: Indirect branch instructions take their target from a register or memory. On x86 platforms, we consider the 'near' forms of JMP and CALL instructions; other forms (e.g., 'far' branches) are not predicted.

Return branches: Although return instructions are typically viewed as a special-case of indirect branches, we consider them separately since they are typically predicted using an specialized branch predictor.

Direct branches: The target of direct branches can be computed based on the immediate offset encoded within the instruction. Again, on x86, we focus on JMP, Jcc (conditional) and CALL instructions.

2.2. Branch target prediction attacks

Speculative execution is inherently not restricted by architectural checks and conditions, and transiently executed code can have persistent and measurable effects (such as cache state changes). Branch prediction attacks typically involve an attacker specifying (or influencing) the branch prediction of a "victim" branch and then inferring data from the victim's domain by speculatively executing a disclosure gadget and leveraging a cache-timing side channel.

Such attacks can be performed using indirect branch target predictions, which form the basis for several well-known attacks, including Spectre Variant 2 [20] (aka BTI), SpectreRSB [28] as well as more recent attacks such as Branch History Injection [9] (aka BHI). We refer to such attacks as 'BTI-style' attacks. A successful BTI-style attack requires several attack primitives. We summarize the standard terminology [34] below.

Speculation Gadget[1]: an indirect branch in the "victim's" security domain, with a branch target prediction which can be specified by the attacker. The original BTI attacks exploited the lack of isolation to directly "inject" indirect predictor targets, while BHI attacks instead rely on influencing target selection based on branch history.

1. Since actual 'gadgets' are not required, we also refer to these concepts more generally as speculation and windowing *primitives*.

This indirect branch also needs to be invoked by the attacker, with sufficient attacker-controlled context.

Windowing Gadget: an operation to delay the architectural execution of the victim indirect branch (typically by delaying the resolution of the branch target), thus opening the window for speculative execution at attacker-controlled location. The most common approach involves evicting the memory containing the branch target address from the cache.

Disclosure Gadget: code which accesses data and conveys it via a side channel. For example, a disclosure gadget may perform a data-dependent cache load, converting data read by transient execution into persistent and measurable cache state changes.

In a typical "universal read" disclosure gadget which can be used to infer the contents of memory, the attacker also needs to control the memory address. This allows such a gadget to load potentially-sensitive data from an attacker-controlled location before transmitting the value using a side-channel.

2.3. Hardware-based mitigations

Both software-based and hardware-based mitigation options exist which can help mitigate attacks such as BTI. We focus on mitigations which are commonly used within the x86 ecosystem and relevant for mitigating BTI-style attacks. Although other platforms (such as ARM) also provide similar mitigations, they are less standardized and often platform-specific.

Both modern Intel and AMD processors support hardware-based mitigations to control the branch prediction behavior of the processor, which can be enabled or disabled by software using MSR writes [2], [21]. In particular, three of these mitigations are widely supported:

"Indirect Branch Restricted Speculation" (IBRS) prevents software from controlling indirect branch predictions in a more privileged domain (e.g., user mode controlling predictions of kernel code, or guest code controlling predictions of hypervisor code). Although this can be implemented by isolating branch predictions in hardware – which appears to be the approach taken by Intel's enhanced IBRS (eIBRS) [9] – other processors appear to instead disable indirect branch predictions entirely [5].

"Single Thread Indirect Branch Predictors" (STIBP) isolates indirect branch predictions between SMT threads (logical processors), preventing one thread from controlling the indirect branch predictions on the sibling thread while it is enabled.

Finally, the "Indirect Branch Predictor Barrier" (IBPB) is intended to prevent previously executed code from controlling the predicted targets of indirect branches executed after the barrier – for example, hardware could implement this by invalidating all branch predictor targets. IBPB is typically used when switching between different security domains, and applies to return branches as well as other indirect branches.

These hardware-based mitigations continue to be improved on newer processors. For example, recent Intel processors also support additional controls which can be used to disable specific types of indirect branch predictions, such BHB-based predictions or alternate RET predictions, and AMD's Zen 4 processor is reported to have support for 'automatic IBRS' [38].

2.4. Retpoline

As an alternative to hardware-based mitigations such as IBRS, Google proposed the "retpoline" mitigation [40], which mitigates indirect branch prediction attacks in software by replacing indirect JMP and CALL instructions with a software sequence. These branches are replaced with a return instruction which is forced to be mispredicted to a "safe" target. Retpoline is widely used, and was believed to be an effective mitigation in most circumstances. However, there are caveats on some processors where alternative predictors (non-RSB) can be used for RET instructions (such as some Intel Skylake-generation processors [22], and AMD Zen, Zen+ and Zen2 processors [8]), and attacks on such cases were demonstrated by recent work [42].

Other software mitigations have also been proposed, such as "randpoline" [39], a non-deterministic software mitigation, as well as LFENCE/JMP.

2.5. LFENCE/JMP

Another software approach for mitigating BTI-style attacks is the "LFENCE/JMP" mitigation. This refers to a code sequence where an LFENCE instruction is used to serialize execution before an indirect branch with a register operand (not a memory address), and thus can be executed without any memory-access latency. The intention is that such a sequence will significantly reduce the potential window for transient execution at the predicted target of the indirect branch, since the architectural target will always be available when the branch instruction is allocated for execution, due to the serialization provided by LFENCE.

As a serializing instruction, LFENCE is widely used as a method for mitigating Spectre Variant 1 vulnerabilities in software. Both Intel and AMD documentation describes that instructions after LFENCE will not be executed until the instructions before LFENCE have completed execution, and the results of those instructions (such as memory loads) are available. We have not observed any behavior incompatible with this definition. We briefly discuss the properties of LFENCE itself in Appendix A.

LFENCE/JMP is presented as an attractive software mitigation option since it may have lower performance impact in some situations, and is simpler to implement compared to alternatives such as retpoline. At the time of our research, LFENCE/JMP was recommended by AMD [3] despite raised concerns (see Appendix B) and was the default BTI mitigation for both the Linux kernel and widely-used hypervisors (e.g., KVM and Xen) on AMD processors. These factors prompted our research into the potential for, and exploitability of, speculation windows – in both LFENCE/JMP and other mitigations.

According to AMD's documentation, the LFENCE/JMP mitigation is described as "convert an indirect branch into a dispatch serializing instruction sequence". It uses this example sequence:

```
jmp *[eax]      ; load+jump to target
```

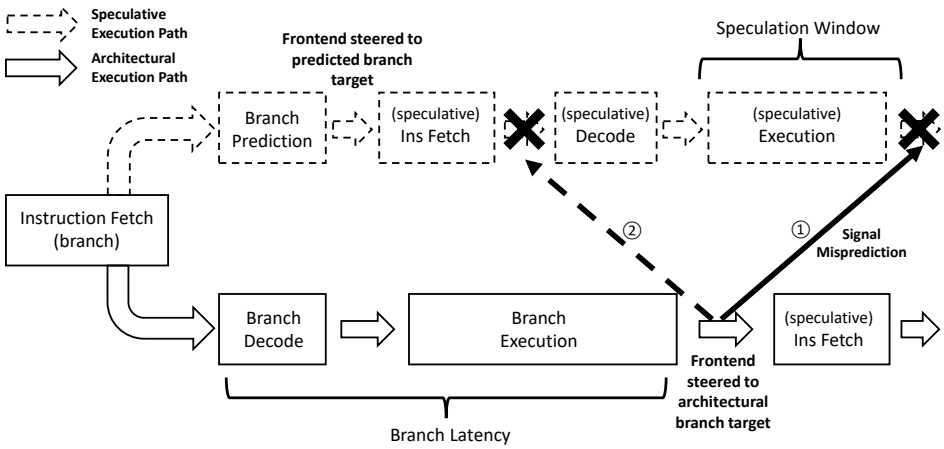

Figure 1: Overview of how branch latency can induce speculation windows. (1) illustrates a case where branch latency is high enough to allow speculative execution of instructions before misprediction is signalled, unlike example (2), where the branch latency is low enough that the misprediction can be signalled before any speculative execution occurs.

And gives an equivalent example using LFENCE/JMP:

```
mov eax, [eax] ; load target address
lfence         ; serialize
jmp *eax       ; jump to target
```

Merely adding an LFENCE before the original code sequence would be insufficient, since the load from memory could act as a windowing gadget. In this example a memory-based indirect branch which loads its target from memory is converted to a load and a register-based indirect branch, with an LFENCE added between to make sure "the load has finished before the branch is dispatched" [3].

3. Branch Latency

The branch prediction unit (BPU) is part of the frontend of the pipeline and is intended to provide branch predictions as early as possible, before the instructions are dispatched for execution – and potentially even before they are decoded. This is so critical to performance that modern processors have multi-stage branch predictors, where predictions are made immediately but may be overridden one cycle later by a more accurate (but higher-latency) predictor [26].

We claim that a key component of BTI-style attacks is the race condition between the speculative execution path and the architectural execution path of the branch – and that this race condition can occur without the need for any data dependency. For an attack to succeed, transient execution of a disclosure gadget at the predicted target must occur before the actual target of the branch is resolved and the misprediction is signaled, creating a speculation window – as illustrated in Figure 1.

3.1. Sources of latency

In our analysis, we focus on the *branch latency* – the time it takes from the point a branch is fetched/predicted, to the point that the architectural branch target is resolved and a misprediction can be detected, the pipeline is cleared and the frontend is steered to the correct target. For direct branches we expect the misprediction to be detected as soon as the branch instruction is decoded. However, for indirect branch instructions (including RETs), a misprediction may only be detected when the branch instruction is executed. In both cases, there is always the potential for a speculation window before a misprediction is signaled and the pipeline is cleared.

Thus, the success of a speculative execution attack heavily relies on branch latency. We identify four categories of possible causes of such latency:

1) **Baseline execution latency**: The inherent latency needed for allocation and execution of the branch (potentially including factors such as the branch type or the alignment of branches).
2) **Data-dependency**: Caused by the delayed availability of the branch target. For indirect branches, this latency may be significant, since it may depend on memory accesses or computation.
3) **Single-thread resource contention**: Latency caused by other instructions executed on the same SMT thread in the out-of-order pipeline.
4) **SMT resource contention**: Resource contention from the sibling thread, which we discuss in detail in subsection 3.2.

3.1.1. Serialization. The standard mechanism to minimize data dependency is to serialize previous instructions with respect to the execution of later instructions. On x86 machines, such serialization is provided by the LFENCE instruction. This minimizes both any data dependency as well as any same-thread resource contention caused by previous instructions. In particular, such serialization can be used after a direct conditional branch to ensure that transient execution does not continue down an unintended path, mitigating attacks such as BCB.

Serialization *before* a branch can be used to minimize branch latency, but it does not affect the baseline execution latency nor any potential SMT resource contention. The LFENCE/JMP mitigation provides an example of how this may be a concern; instructions may still be transiently executed in the window after the serialization of LFENCE,

but before the branch instruction is executed and the misprediction is signaled.

3.2. SMT resource contention

When SMT is in use, certain hardware resources are shared between the sibling threads of the same core. Decoded instructions from both threads are dynamically dispatched and executed on execution ports and may result in resource contention and cause measurable delay in execution of certain instructions.

For analyzing branch execution latency and potential speculation windows, we need to consider the contention for resources which could result in the branch execution on one thread being delayed by activity on the sibling thread. We theorized about three broad categories:

General port contention: Instructions may need to be executed by specific execution ports, which is a known source of contention [1]. Executing instructions that use the same execution ports as branch instructions may delay the execution of a branch on the sibling thread, allowing the speculative execution path to continue and thus expanding the speculation window.

Branch-specific contention: There may be shared execution units and/or resources that are specific to branch instructions (or a specific type of branch instruction). The allocation and execution of branch instructions may depend on the availability of such resources. This means that branches being executed on one thread may delay the execution of the branch on the sibling thread, expanding the speculation window.

Contention from other causes: There are likely causes of contention which are not included above, which we group into a single category. We will discuss concrete examples as part of our case studies.

To analyze the impact in practice, we propose running appropriate SMT workloads on a sibling thread. If such workloads have an impact on branch latency, we would expect them to effectively act as windowing gadgets.

Workloads for most categories could simply execute a specific type of instruction in an unrolled loop – for example, a lengthy sequence of XOR instructions in a loop for evaluating general port contention. For branch-specific contention, we suggest a similar approach but using branch-focused workloads, such as direct JMPs, direct far JMPs, indirect near JMPs and conditional JMPs. Where branch misprediction is possible (e.g., indirect and conditional branches), we propose evaluating workloads with both correctly-predicted and mispredicted branches.

4. Experiment Methodology

In this section, we propose a methodology for designing and running experiments to investigate the speculation window discussed in section 3, and in particular, the effect of branch latency on such windows.

At a high level, each experiment should consist of the following steps, generalizing the approach of previous research:

1) **Training**: executing a trainer branch to populate the relevant branch predictor with a specific target that corresponds to the location of one of the disclosure gadgets described above.
2) **Speculation**: executing a "victim" branch which is intended to consume the target prediction from the training step, and may transiently execute the trained disclosure gadget.
3) **Measurement**: probing the side effects (cache state changes) from the fetching of the trained target and/or the transient execution of the disclosure gadget.

TABLE 1: Disclosure gadgets used in our analysis

	Discloses memory?	High-resolution channel?	Dependency chain size
load-once	✗	✗	1
load-load	✓	✗	2
load-shift-load	✓	✓	3

Where possible, the training step should be executed multiple times before the speculation step, to increase the probability of training the branch predictor correctly and thus observing the expected branch target prediction.

To account for any history-based branch predictors, we normalize the branch history by executing a sequence of branches (or aliasing branches) right before the trainer branch in the training step, and the victim branch in the speculation step.

For indirect branches, we can use the same branch in both training and speculation steps ("self-training") and simply change the target; for direct branches, we can either modify the instruction to change the encoded target between the training and speculation steps, or execute a victim branch at an aliasing branch address (as discussed in subsection 6.1).

Attempting to observe instruction fetches at the predicted branch targets allows the occurrence of branch predictions to be confirmed, and branch predictor address aliasing to be identified. When branch predictions are observed, the next step is to check for transient execution of a minimal *load-once* gadget (see subsection 4.1) on the speculative path. If successful, experiments can then be performed to measure the speculation window size as well as to determine whether longer disclosure gadgets can fit in the relevant speculation window.

If instructions are fetched from the trained branch target but cache side-channels do not observe the side-effects of the load in the gadget, alternative disclosure gadgets can be tested to check for speculative execution (such as a load instruction with an immediate address, or one of the other alternatives discussed above). Finally, other sources (such as performance counters) may also be useful to determine whether any instructions were transiently executed.

In the remainder of this section, we expand upon the details of our proposed methodology.

4.1. Observing branch predictions

When a branch prediction occurs, instructions at the predicted target address may be speculatively fetched and executed. Such activity may be exposed by performance and/or sampling-based counters. However, to detect transient execution, we can simply place a disclosure gadget

at the predicted target address and check if we see side-effects from the execution of this gadget. For our methodology, we consider three forms of disclosure gadgets, all using a cache-based side channel. As summarized in Table 1, the speculation window needed to execute these gadgets increases as they become more widely applicable.

"**Register-disclosure**" or *load-once* gadget: a single load instruction, which carries out a speculative cache load based on data which is already present in a register. Although it does not provide a 'universal read gadget', it could allow an attacker to infer some bits of such data. Since only a single load is required, it may be possible to execute this gadget in a very small execution window:

```
mov ecx, [rsi + rdi]
```

"**Partial memory-disclosure**" or *load-load* gadget: two dependent loads, where the second dependent load directly consumes the value of the first load. This style of gadget does not provide enough resolution for a covert channel to infer all bits, but an attacker with sufficient control of the base address can create a 'universal read gadget' by increasing that address until it crosses cache lines (as discussed in e.g., [42]):

```
mov ebx, [rdi + rdx]
mov ecx, [rsi + rbx]
```

"**Memory-disclosure**" or *load-shift-load* gadget: two dependent load instructions with a shift instruction, providing a full 'universal read gadget' and potentially allowing all bits to be inferred. The first load reads a value from a memory location using a pointer in a register (line 1), which is then shifted to achieve a one-to-one correspondence between data value and cache lines (line 2), and the second load performs a data-dependent load using the shifted value (line 3), which should be visible in the cache after the speculative execution of the gadget:

```
1 mov ebx, [rdi + rdx]
2 shl ebx, 0xc
3 mov ecx, [rsi + rbx]
```

If we do not observe any side-effects from transient execution of any of these disclosure gadgets, we can also consider alternative approaches. For example, we could replace FLUSH+RELOAD with a different cache-based side channel. Evicting cache lines from the L1 cache (rather than flushing them from the entire cache hierarchy) could speed up the second load. The size of the needed speculation window could be further reduced by avoiding evicting lines at all; for example, using cache LRU states [45], which can provide a cache-based side channel for data which is already present in the cache. We may even be able to remove the need for a second dependent load entirely by using a non-cache/memory-based side channel, such as by training the conditional branch predictor in a disclosure gadget and inferring data using the behavior of an aliased branch [16]. However, we focus on the three gadgets above in our work.

Even if no transient execution is observed, we can still observe branch predictions by checking whether instructions are fetched at branch targets. Such fetches can be observed using a cache-based incidental channel (FLUSH+RELOAD [46]), as proposed in [15]. We flush (or otherwise evict) the cache line containing the instructions at the trained branch target, execute the branch, and then measure how long it takes to reload the flushed cache line.

4.2. Speculation window size

To quantitatively characterize the branch latency where a speculation window is present, we add 1-byte NOP instructions to the start of the *load-once* gadget. These NOPs act as spacers and delay allocation or execution of the MOV instruction.

Iteratively adding NOPs until we no longer observe any cache effects from the disclosure gadget's load allows us to determine the point where the speculation window is no longer large enough to reach and execute the MOV instruction. Although NOP instructions may overestimate the size of the speculation window in terms of the "number of instructions" (since they may not actually be allocated or executed), this methodology provides an estimate of the "upper bound" on the potential size of the window, and in practice we did not observe any differences when replacing the NOP instructions with (1-byte) CBW instructions.

4.3. Execution details

It is desirable for these experiments to be performed in a kernel mode environment with interrupts disabled to avoid possible false positives due to interrupts and other system activity, although in practice experiment results may be sufficiently reliable to rule out such sources. Similarly, when SMT is not required for an experiment, it should be run on a single thread with SMT disabled, to avoid unintended contention. Experiment design should also take other potential sources of speculation into account (such as prefetching and store-to-load forwarding) when writing code; alternatively, such features can be disabled using MSR-based controls where present.

5. Case Studies

To illustrate the role of branch latency and the related speculation window in branch target prediction attacks, we provide a detailed case study for each of the three types of branches discussed in subsection 2.1:

- For indirect branches (JMP and CALL), we investigate the LFENCE/JMP software mitigation.
- For direct branches, we investigate whether branch target prediction attacks are possible at all.
- For return branches, which are primarily predicted from the RSB/RAS, we investigate the IBPB hardware-based mitigation.

The focus of our work is to investigate mitigations that rely on suppressing predictions by minimizing the speculation window; as such, we did not evaluate mitigations which attempt to isolate or redirect such predictions, such as retpoline and Intel's enhanced IBRS. We also exclude cases where the RET instruction may use alternate predictors, which were covered by other recent work [42].

For each case study, we conduct experiments corresponding to our proposed methodology in section 3 on a variety of recent x86 CPUs from both Intel and AMD, as listed in Table 2. Specifically, we investigate whether branch predictions may occur and which sources of branch latency are present, determine the size of the corresponding speculation windows, and then explore the impact of SMT activity.

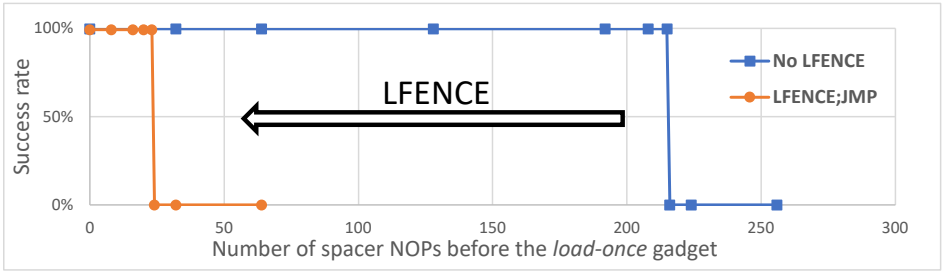

Figure 2: Typical effect of LFENCE/JMP on baseline execution latency

TABLE 2: Processors used in our experiments, along with the corresponding microarchitectures.

Microarchitecture	Tested processor
Goldmont Plus	Intel Pentium Silver N5000
Tremont	Intel Core i5-L16G7 (Lakefield)
Sunny Cove	Intel Core i5-1038NG7 (Ice Lake)
Willow Cove	Intel Core i7-1165G7 (Tiger Lake)
Golden Cove	Intel Core i9-12900K (Alder Lake)
Gracemont	Intel Core i9-12900K (Alder Lake)
Zen	AMD Ryzen 5 2400G
Zen+	AMD Ryzen Threadripper 2990WX
Zen 2	AMD Ryzen 7 4700G
Zen 3	AMD Ryzen 5 5600G

As discussed, one possible issue with conducting these experiments in userspace is that OS context switches may occur (e.g., for LFENCE/JMP, between LFENCE and the JMP), and instructions such as SYSRET are not documented to be serializing on AMD processors. Therefore, we also reproduced the relevant results in a kernel mode environment with interrupts disabled.

All the experiments which did not involve SMT port contention were run with SMT disabled (on a single thread), to avoid unintended contention. We also designed our tests to rule out false positives due to prefetching and other unexpected sources of speculation, and during our SMT tests, we only report results which we consistently observed for the duration of our test.

The scope of this work did not involve finding the best possible SMT workloads for each of the cases; our goal was to cover the instructions and cases needed to cause the desired types of contention during execution, rather than (for example) optimizing the size or reliability of any resulting speculation windows.

5.1. LFENCE/JMP

Our first case study investigates the LFENCE/JMP software mitigation, which is designed to mitigate BTI attacks on indirect branches. While other mitigation options generally rely on restricting the Speculation Primitive of BTI-style attacks, the LFENCE/JMP mitigation aims instead to remove the Windowing Primitive. In other words, while attacker-controlled transient execution can still technically happen, the expectation is that the speculation window will be too small for it to be exploitable.

Since the LFENCE/JMP mitigation essentially relies on a race condition, despite the lack of data-dependency, we still expect there to be a speculative window created by the baseline execution latency.

Figure 2 presents an illustrative example (on Zen 2) of the reduction in speculation window before and after applying LFENCE/JMP to an indirect branch, measured using the NOP method described in subsection 4.2. This visualizes the impact that LFENCE/JMP can have on the speculation window of an indirect branch. The number of instructions which can be transiently executed is greatly reduced, but a residual window clearly still remains (in this example, allowing 23 NOPs to be executed before the gadget).

5.1.1. LFENCE/JMP without SMT. Our first experiment investigates whether disclosure gadgets can be transiently executed within this speculation window. The results can be seen in Table 3.

We see that the *load-once* gadget is transiently executed on all processors (adding the LFENCE decreases the success rate by $< 1\%$), indicating the speculation window is large enough to execute a one-instruction "register-disclosure" gadget with just the baseline execution latency, despite the use of LFENCE/JMP. This confirms our expectation that such a speculation window should be present.

On the other hand, the *load-load* gadget requires a significantly larger speculation window in which two dependent loads can be executed. A sufficient large window only appears to be present on some tested processors – AMD processors prior to Zen 3, and one Intel processor (Gracemont) – supporting AMD's own findings (see the Disclosure section). Finally, executing the *load-shift-load* gadget would require an even larger window, and LFENCE/JMP appears to limit the window sufficiently to mitigate such gadgets on all tested processors.

To quantify these differences between processors, we use the technique from subsection 4.2 to measure the size of the speculation window; specifically, we measured the maximum number of NOPs which can be added in front of the *load-once* gadget before the cache effects were no longer visible – i.e., when we reach the limit of the speculation window. The results are shown in Table 4.

Our experiments show that the impact of LFENCE/JMP on the speculation window size appears to differ significantly between processors. As expected, the window appears to be largest on the processors where we observed transient execution of the *load-load* gadget, and the results support our conclusion that such a gadget fails to transiently execute on other processors, such as Zen 3.

5.1.2. LFENCE/JMP with SMT. We evaluated LFENCE/JMP using different SMT workloads as proposed in subsection 3.2. To evaluate contention from

TABLE 3: Verification of the effectiveness of LFENCE/JMP, and confirmation of the race condition.

Microarchitecture	*Load-shift-load* gadget		*Load-load* gadget		*Load-once* gadget	
	No LFENCE	With LFENCE	No LFENCE	With LFENCE	No LFENCE	With LFENCE
Goldmont Plus	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$\sim 10\%$
Tremont	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$\sim 50\%$
Sunny Cove (ICL)	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$> 99\%$
Willow Cove (TGL)	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$> 99\%$
Golden Cove (ADL)	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$> 99\%$
Gracemont (ADL)	$> 99\%$	0%	$> 99\%$	$> 99\%$	$> 99\%$	$> 99\%$
Zen	$> 99\%$	0%	$> 99\%$	$> 99\%$	$> 99\%$	$> 99\%$
Zen+	$> 99\%$	0%	$> 99\%$	$> 99\%$	$> 99\%$	$> 99\%$
Zen 2	$> 99\%$	0%	$> 99\%$	$> 99\%$	$> 99\%$	$> 99\%$
Zen 3	$> 99\%$	0%	$> 99\%$	0%	$> 99\%$	$> 99\%$

TABLE 4: NOP spacer experiment results with *load-once* "register-disclosure" gadget and LFENCE/JMP (without SMT).

	Goldmont Plus	Tremont	Sunny Cove	Willow Cove	Golden Cove	Gracemont	Zen	Zen+	Zen 2	Zen 3
Max # of NOPs	5	5	2	2	14	23	23	23	23	15

unknown sources, we experimented with a variety of applications, microbenchmarks, and test cases – we present results for the most "interesting" workload we discovered in this category, which consists of repeated calls to the nanosleep system call in a loop, with both parameters set to NULL. Similarly, we only present results for one general port contention workload (XOR), which serves as a representative example.

We only tested a *load-shift-load* gadget, given that the LFENCE/JMP window is typically large enough to allow transient execution of the *load-once* gadget, even in the absence of SMT port contention.

Table 5 shows results for all the tested processors which support SMT; in all cases we observed at least one SMT workload which can induce a large enough speculation window to transiently execute the *load-shift-load* gadget, despite the use of the LFENCE/JMP mitigation.

On AMD microarchitectures prior to Zen 3, we have observed that mispredicted branches (as well as far JMPs) executed on a sibling logical processor appear to be an effective workload to increase latency in the indirect branch execution, which may indicate contention for resources that are specific to mispredicted branches. Similarly, sufficient contention is also seen with our XOR workload on microarchitectures prior to Zen 2, which is likely due to general port contention.

The best SMT workload for the tested Intel CPUs appears to be correctly-predicted indirect JMPs, which may indicate contention for resources specific to indirect branches. The expanded speculation window caused by the nanosleep workload may have the same root cause, since indirect branches will be executed in the kernel on the sibling logical processor. However, this cannot be the only source of contention, since we also observed hits for some other SMT workloads.

On Zen 3, the majority of these SMT workloads do not appear to increase the speculation window after the LFENCE/JMP branches on the sibling logical processor, which may indicate fewer shared resources or differences in the way in which resources are allocated. However, since the nanosleep workload still opens a sufficiently large speculation window to execute the *load-shift-load* gadget, there still appears to be at least some form of resource contention between sibling logical processors on Zen 3. Although it would presumably be possible to root-cause the source of the contention and develop a more focused workload, we did not consider this necessary given the scope of this study.

Note that we did not attempt to optimize these workloads in terms of the effective instruction density, nor to synchronize them with the indirect branch execution on the sibling logical processor. In particular, the mispredicted branch workloads also execute code to cause branch mispredictions, and we did not attempt to optimize the number of mispredicted branches.

5.2. DBTI

As documented by processor vendors [24], previous work [15] showed that branch target prediction can also occur for direct branches (with a IP-based prediction algorithm) by observing instruction fetches. As such, we investigated the potential for speculative execution due to branch prediction of direct branches.

5.2.1. DBTI without SMT. Listing 1 shows our experimental approach, using two direct JMP rel32 instructions with addresses which are known to alias in the structure used to store branch predictor targets (e.g., BTB). We encoded different targets for the "trainer" branch (in this example, an offset of +0x200) and the "victim" branch (in this example, +0x400).

If DBTI is present, the "victim" branch will be predicted using the BTB entry created by the "trainer" branch, and the instructions at offset +0x200 from the victim branch may be transiently executed. Our experiment found this behavior on several AMD processors (including Zen, Zen+ and Zen2), with a speculative window large enough to execute at least one load instruction, allowing us to observe the corresponding cache state change. We saw the same behavior for both conditional and unconditional direct branches, with both 32-bit and 8-bit relative offsets.

We again characterized the speculation window by adding 1-byte NOP instructions before the load instruction in the one-load gadget. We used a 3-byte load instruction with the beginning of the gadget aligned to 64 bytes.

TABLE 5: SMT workloads: *load-shift-load* "memory-disclosure" gadget success rates with LFENCE/JMP

Microarchitecture	No LFENCE	No workload	SMT workloads							
			Direct JMP	Jcc predicted	Jcc mispred	Indirect JMP predicted	Indirect JMP mispred	Far JMP	XOR	nano sleep
Sunny Cove (ICL)	> 99%	0%	0%	< 1%	0%	98%	0%	0%	10%	1%
Willow Cove (TGL)	> 99%	0%	0%	< 1%	0%	95%	0%	0%	10%	2%
Golden Cove (ADL)	> 99%	0%	0%	0%	0%	99%	0%	0%	13%	2%
Zen	> 98%	0%	0%	0%	8%	0%	5%	15%	30%	6%
Zen+	> 99%	0%	0%	0%	12%	0%	5%	23%	43%	8%
Zen 2	> 99%	0%	0%	0%	21%	0%	6%	11%	< 1%	5%
Zen 3	> 99%	0%	0%	0%	0%	0%	0%	0%	0%	1%

```
; this will create a BTB entry
; predicting a jump to +200
6020000: jmp 6020200

; intended target at +200
6020200: ret
```

Trainer code

```
; this branch will be predicted
; using a BTB entry
4000000: jmp 4000400

; target at +200
4000200: mov rax, [rax]
```

Victim code

Listing 1: Basic DBTI test code (using example addresses)

TABLE 6: DBTI results for different branches

	Zen	Zen+	Zen2	Zen3	Intel CPUs
JMP rel32/8	✓	✓	✓	✗	✗
Jcc rel32/8	✓	✓	✓	✗	✗
CALL rel32	✓	✓	✓	✗	✗

TABLE 7: NOP spacer experiment results for DBTI

	Zen	Zen+	Zen2	Zen3	Intel CPUs
Max # of NOPs	7	7	7	N/A	N/A

The results, without any attempt to increase the size of the window (e.g., SMT workloads or branch predictor contention), are shown in Table 7.

Given the lack of a windowing gadget in the single-threaded DBTI attack, it is reasonable to observe consistent numbers across different DBTI-impacted microarchitectures, which likely have similar design in the relevant frontend pipeline stages (as also observed for LFENCE/JMP on these processors).

However, we do not observe a speculation window on any other tested processors. As proposed in subsection 4.1, we can examine whether branch predictions occur by checking for speculative fetches of the predicted target.

The results from this experiment are shown in Table 8, based on the ratio between the time needed to load the cache lines corresponding to both the speculative and architectural targets; a ratio close to 1:1 indicates that a speculative fetch has occurred. We see both branch predictor aliasing and branch predictions on *all* tested processors, from both AMD and Intel. However, on processors unaffected by DBTI, the frontend appears to be able to detect the incorrect prediction and clear the pipeline before any transient execution can occur.

We also observed such instruction fetches *without* the need for a victim direct branch, on all processors. After our initial disclosure, we also successfully reproduced DBTI without the presence of a victim direct branch (e.g., with a NOP at the target) on Zen, Zen+ and Zen2 proces-

TABLE 8: Speculative fetches due to direct branch predictions for aliased and non-aliased branches

Processor	Aliased	Non-aliased
Goldmont Plus	✓	✗
Tremont	✓	✗
Sunny Cove	✓	✗
Willow Cove	✓	✗
Golden Cove	✓	✗
Gracemont	✓	✗
Zen	✓	✗
Zen+	✓	✗
Zen 2	✓	✗
Zen 3	✓	✗

sors; AMD describe this behavior as "BTC-NOBR" [8]. This specific case can be mitigated (on Zen 2) by setting a specific bit in an MSR; in fact, our experiments show that this also suppresses instruction fetches. However, this bit appears to have no effect when a victim branch is present.

IBRS does not appear to affect the observed behavior on any processors. AMD's guidance for BTC states that IBPB flushes both direct and indirect branch prediction targets, thus it can also be used to mitigate DBTI; we confirmed that we do not observe older predictions for direct branches after IBPB has been invoked.

5.2.2. DBTI with SMT. Our LFENCE/JMP analysis showed that speculation windows for indirect branches could be expanded using SMT workloads; the natural question is whether this also applies to direct branches. We repeated our speculation window experiments with SMT workloads. The results are shown in Table 9; specifically, the maximum number of 1-byte NOP instructions that can be added to the start of the one-load gadget and still permit transient execution of the gadget.

Since our load instruction is 3 bytes, the best case of 13 NOPs provides 16 bytes of total speculation window. This limit is likely imposed by the 16-byte decode window, as also mentioned in [41].

Our results show that direct branches are also affected

TABLE 9: NOP spacer experiments with SMT workloads: *load-once* with DBTI

Microarchitecture	No SMT	SMT workloads							
		Direct JMP	Jcc predicted	Jcc mispred	Indirect JMP predicted	Indirect JMP mispred	Far JMP	XOR	nano sleep
Zen	7	7	7	7	8	7	9	8	13
Zen+	7	7	7	7	8	7	9	8	13
Zen 2	7	7	7	7	8	7	9	8	13

by SMT resource contention, although in a differenct way when compared to indirect branches. This is likely due to different hardware components (e.g., execution units) being used for these different branch types.

With SMT contention, we observe that (at least on Zen 2) two dependent loads (a load-load gadget) can be successfully executed speculatively when running the nanosleep workload. This means that memory-disclosure attacks may be possible (as we discuss in section 6), despite AMD's original analysis [8] stating that two dependent loads "will not be able to execute before the pipeline is flushed" (and thus that only register contents can be disclosed). Since Zen and Zen+ processors do not support STIBP, mitigating BTC-RET [8], [42] requires disabling SMT – as such, we did not investigate SMT workloads on these processors.

We believe the root cause of this expanded DBTI window on Zen 2 may be pipeline resource contention due to deep speculation; we developed an optimized workload involving speculative execution of a chain of function calls (including a mispredicted RET) ending in REP STOS, all in the transiently executed shadow of a fault. Our experiments demonstrate this workload obtains a "success rate" of $> 1\%$ – i.e., the frequency at which we observe result from the load-load gadget using the cache side channel. Note that it still has a high ($> 99\%$) signal-to-noise ratio. We also obtained several successes per second with a cross-mode proof-of-concept reading kernel memory – which we believe is sufficient to show that inferring kernel data is practical.

We do not, however, see a signal when adding another instruction (such as a shift) to the dependency chain (as in the load-shift-load gadget), which may indicate limits on the speculation window size. AMD informed us that the first load must be 32- or 64-bit sized, which significantly limits the gadgets which fit inside the window. We leave further investigation of the underlying cause of this workload, as well as whether it may be possible to further expand the speculation window, as a topic for future work.

5.3. IBPB

Our final case study investigates return branches, which are typically predicted using the RSB/RAS. IBPB is recommended when switching between security domains to prevent predictions using previous RSB/RAS entries, to mitigate potential SpectreRSB attacks [28].

However, Intel recently documented "post-barrier RSB predictions", an issue with some Intel processors where a single RSB prediction may be made corresponding to the most recent RSB entry created prior to the IBPB barrier, creating an unexpected speculation window. Unlike LFENCE/JMP and DBTI, the data dependency can contribute to the latency of the branch, since the RET instruction will not be executed until the return address (on the stack) is known.

Since IBPB is only accessible to supervisor mode, this issue is unlikely to be exploitable in practice, since the last RSB entry prior to IBPB will usually be controlled by the OS kernel, rather than an attacker. However, given the existence of this corner case, we applied our methodology to investigate whether there could still be a race condition which could result in a speculation window for predictions based on these 'stale' RSB entries that are expected to be invalidated by the invocation of IBPB.

5.3.1. IBPB without SMT. Unlike previous experiments, the RSB/RAS predictors are based on the most recent CALL instructions, therefore a different "training" step is required. We use a sequence of CALL instructions to populate the RSB with designated speculative return targets (after the CALL instruction). One of them corresponds to a disclosure gadget, and the remainder contain only RET instructions. This results in a speculatively executed RET instruction descending through the targets (from RSB entries) on its speculative path, while allowed by the speculation window.

For our experiment, we first execute this CALL sequence (our training step), and then invoke IBPB (using an MSR write) and switch to an alternate stack (previously flushed from the cache, to cause data dependency) with a different set of return addresses. We then execute a RET instruction, potentially causing speculative execution, and observe (using FLUSH+RELOAD) whether our gadget was transiently executed. By increasing the depth of the CALL instruction corresponding to the disclosure gadget, we can determine the depths at which targets are predicted from the RSB. We performed most of our experiments in kernel code, since only privileged code can execute the WRMSR instruction needed to invoke IBPB.

The results are shown in Table 10. As expected, we observe a single RSB entry on some Intel parts; we confirmed that this is mitigated by Intel's suggested sequence [25]. However, on Zen, Zen+ and Zen2, we observe transient execution for *multiple* RSB entries; in fact, RSB predictions do not appear to be affected by IBPB at all. This behavior was later confirmed by AMD.

5.3.2. IBPB with SMT. Since the speculation window for RET predictions after IBPB is large on the affected processors, unlike our LFENCE/JMP and DBTI studies, there appears to be little value in investigating whether this window could be expanded further using SMT workloads. On other tested processors we did not observe any branch predictions across IBPB. However, the question remains whether SMT workloads could somehow induce a speculation window on these processors.

TABLE 10: Observed RET predictions after IBPB

Processor	RET predictions
Goldmont Plus	None
Tremont	None
Sunny Cove	Most recent
Willow Cove	Most recent
Golden Cove	Most recent
Gracemont	None
Zen	Multiple
Zen+	Multiple
Zen 2	Multiple
Zen 3	None (without SMT)

We ran experiments using the SMT workloads described above, and discovered that a workload consisting of usleep(1) in a loop results in branch target prediction of RETs on Zen3 across IBPB. Although such predictions only occur for a portion of the time (we currently observe $\sim 1\%$), this would be sufficient to mount an attack since there are few false positives (i.e. there is a high signal-to-noise ratio). Moreover, we observe a large speculation window, similar to the results on other affected AMD processors (Zen, Zen+ and Zen2).

Our analysis suggests that this may be caused by sleep state transitions of the sibling logical thread. Success rates are correlated with MWAITX invocations on this sibling thread, and we have not observed this behavior when SMT is disabled. Since AMD state that "Return instructions are always immune to influence by the other thread" [3], we did not investigate potential cross-thread predictions. Given that the root cause appears to be unrelated to branch latency, we leave further analysis to future work.

6. Exploitability

The results presented above alone do not necessarily mean that the demonstrated speculation windows are exploitable by an attacker. In this section, we discuss how practical exploitability may depend on many other factors, even assuming that an attacker has arbitrary code execution in userspace. In particular, we propose potential avenues for exploiting DBTI as well as a full proof-of-concept attack against the Linux kernel using LFENCE/JMP, to demonstrate that the branch latency we have discussed can be a real-world problem. For RET-related attacks that bypass IBPB on AMD processors, we refer to previous work [28] on such attacks.

Limits on the speculation window size may also have other implications; for example, page walks may not be possible, which may limit speculative memory accesses to pages which have their address translations in the Translation Lookaside Buffer (TLB). Similarly, it may not be practical to execute some instructions due to conflicting needs for pipeline resources.

On the other hand, a speculation window which can contain multiple dependent loads – potentially allowing transient execution of a cache-based "universal read" gadget – may raise a higher level of concern. We have shown that – at least in some circumstances – such gadgets can be transiently executed despite the use of LFENCE/JMP, DBTI, or IBPB. Additional results related to alternative disclosure gadgets are provided in Appendix C.

Exploitation in practice also depends on which indirect call sites and disclosure gadgets could be used by an attacker. Assuming a scenario where a userspace attacker attempts to obtain data from kernel mode, we assume Supervisor-Mode Execution Prevention (SMEP) is enabled, which requires the disclosure gadgets to be located in executable kernel memory. If predicted targets are not isolated between modes, such as on older Intel CPUs as well as current AMD CPUs (see below), a userspace attacker can inject targets into indirect branch predictor entries: any bytes in executable kernel code may be a potential gadget. Defenses such as fine-grained Address Space Layout Randomization (ASLR) can be bypassed with the use of cache-timing side channels.

If the userspace attacker cannot directly specify predicted targets for the kernel indirect branches (as may be the case with Intel's eIBRS), and potential locations for transient execution are limited to existing kernel targets, then exploitation also requires identifying a suitable disclosure gadget within this more restricted scope.

6.1. Predictor target aliasing

Our experiments demonstrated (as a side effect) that branch target aliasing is possible on both Intel and AMD processors. However, cross-mode attacks against OS kernels and/or hypervisors are only possible on processors which do not isolate predicted targets between modes. Previous work has shown that older (pre-eIBRS) Intel CPUs do not have such isolation, allowing simple injection of targets, and this is also true on Zen [27]. Such aliasing was also recently documented on Zen 2 processors [42]. Our experiments show that all recent AMD CPUs (including Zen 3) seem to lack mode-based target isolation.

In particular, we experimented with cross-mode aliasing of short indirect branch predictor targets on AMD processors, which was the approach used by the original Spectre Variant 2 attacks on older Intel CPUs without eIBRS [18]. A simpler approach (as discussed in [27]) is to branch to an illegal target and suppress the fault.

Our cross-mode aliasing experiments confirmed that a userspace attacker without the ability to execute code at a linear address with bit 47 set can instead toggle other address bits of a userspace branch to cause collisions with kernel branches due to aliasing in the branch predictor entries. The branches contributing to the BHB (Branch History Buffer, following the terminology from [18]), the trainer indirect branch as well as the branch target itself, can thus be implemented in userspace, allowing an attacker to control the lower bits of the predicted targets of kernel indirect branch addresses. The specific aliasing behavior (and the number of controllable lower bits of the predicted targets) varies between CPU generations, but we reproduced similar aliasing behavior for short targets on all the AMD CPUs tested.

On AMD processors affected by DBTI, we also observe similar cross-mode aliasing behavior and demonstrate that the predicted targets of direct branches in OS kernel (or in a hypervisor) can be trained using aliasing branches from user space (or a guest). This allows DBTI attacks to be performed across security domains, even with SMEP enabled.

```
; bounds check
cmp rax, rbx
; conditional branch; can be trained
ja  out_of_bounds
; speculatively load arbitrary memory
mov rbx, [rcx + rax]
; intervening instructions
nop
nop
nop
nop
; DBTI predicts this to a 2nd load
jmp somewhere
```

Listing 2: Bounds Check Bypass example

6.2. Thread isolation

BTI-style attacks are known to be possible across threads. We also observed that the BTB entries for direct branches are shared between threads when SMT is enabled, allowing DBTI attacks to be performed from a sibling thread. As mentioned earlier, STIBP provides thread isolation for indirect branch predictions, which also applies to our test cases with LFENCE/JMP mitigation; we found that STIBP also applies to direct branches, and mitigates DBTI attacks performed across threads. However, STIBP does not prevent SMT workloads from expanding the DBTI speculation window.

6.3. Bounds Check Bypass + DBTI

BCB vulnerabilities require code that executes (a) a speculative out-of-bounds load of data from memory, and (b) instructions which convey that data using a side-channel. Mitigating BCB depends on identifying such code patterns and adding serialization.

However, with DBTI, we can replace (b) with *any branch instruction*, since DBTI can be used to transiently redirect such a branch to a suitable instruction of the attacker's choosing. If BTC-NOBR has not been mitigated – such as on Zen and Zen+ processors which cannot be configured to suppress predictions for non-branch instructions – this could even be a non-branch instruction. This technique (also mentioned in [43]) provides "universal read" gadgets within a small speculation window, showing that attacks may be possible even without SMT workloads.

Listing 2 shows an example of code which we confirmed is vulnerable to DBTI attacks (with DBTI used to direct prediction to a load using rbx as an offset).

Since such patterns do not require a second load (or other side-channel transmission instruction), but only a code path consisting of a speculative load followed by a direct branch, we found suitable code patterns to be common in OS kernel code. In fact, previous work [36] has shown that kernel code which speculatively dereferences memory specified by userspace-controlled register values may be transiently executed during system calls without the need for deliberate targeting. As such, even disabling SMT may be insufficient to mitigate DBTI attacks. AMD's guidance [8] refers to code which may be vulnerable to these attacks as 'half-v1 gadgets', and recommends mitigating such code using existing techniques.

6.4. Exploiting Unprivileged eBPF

Finally, we describe a proof-of-concept attack against the Linux kernel for one case – LFENCE/JMP – as a demonstration that the speculation windows we found can be a potential issue in real-world scenarios.

After BHI was disclosed to the Linux community, the upstream Linux kernel was updated to inline the indirect branch "thunk" calls where possible, including in JITed eBPF code. At the time of our research, this meant that the default configuration for AMD processors used LFENCE/JMP to protect indirect branches in unprivileged eBPF. To confirm whether the speculation window opened by LFENCE/JMP can be used in a realistic attack, we wrote a proof-of-concept (PoC) exploit which demonstrates that kernel memory contents can be inferred using eBPF on an AMD Zen 2 processor.

The eBPF JIT translates code in a fairly direct manner from eBPF bytecode to x86 assembly, which gives us a certain amount of predictable control over register contents and memory contents. We developed an unprivileged eBPF program which contained an appropriate indirect branch; technical details are provided in Appendix D.

Although the aliasing on AMD CPUs allows a wide range of kernel code bytes to be used as targets, our goal was to create a simple proof-of-concept which did not rely on a specific kernel binary; as such, we did not search existing kernels for bytes to use as disclosure gadgets (as done by similar work [42]). Instead, we created our own disclosure gadgets by embedding constant values inside eBPF code which would be interpreted and executed as x86 code when reached directly from a jump. Although Linux's eBPF constant blinding can mitigate this, it is disabled by default as an optional hardening feature.

We used these constants to construct "universal read" gadgets which would infer a single bit of information from an arbitrary kernel address, and access a cache line based on whether that bit is 0 or 1. By creating one such gadget for every bit, we successfully used this PoC to infer the contents of kernel memory despite the use of the LFENCE/JMP mitigation, on Zen 2 with a suitable SMT workload. Since the technique described above can be used by an attacker to construct arbitrary disclosure gadgets, our experiments imply that exploitation may also be possible without SMT.

We expect that the proof-of-concept BHI attacks [9], which use unprivileged eBPF, would also not be mitigated by LFENCE/JMP on Intel CPUs despite the use of eIBRS; these specific attacks seem likely to require an SMT workload due to the need for a larger speculation window (and 3 dependent loads), but this may not be the case for other BHI attacks.

Note that LFENCE/JMP is no longer the default option even on AMD processors due to our work, and that unprivileged eBPF has been disabled by default in recent kernels as a consequence of the BHI disclosure [9]. Therefore, reproducing this PoC on current Linux kernels would only be possible where unprivileged eBPF has been explicitly enabled, and where the kernel is configured to use the LFENCE/JMP mitigation.

6.5. Mitigations

Finally, we present some options for mitigating or reducing the security impact of the cases we identified where current mitigations may not be (fully) effective.

Alternative mitigations may be an option; LFENCE/JMP can be replaced with other BTI mitigations, such as retpoline or IBRS, and for processors where IBPB does not clear the RSB/RAS by IBPB, software can apply "RSB stuffing" to steer RET predictions to safe targets which prevent attacker-controlled speculation.

DBTI, however, may be more complicated to mitigate. Enabling STIBP (where available) appears to prevent cross-thread control of predictions, but does not prevent an attacker from using an SMT workload to expand the DBTI speculation window. AMD has released updated guidance for BTC which documents that a *load-load* gadget cannot fit in the 'early redirect' speculation window after setting a specific MSR bit (only available on Zen 2). We repeated the relevant experiments with it set, and confirmed that a *load-load* gadget can no longer execute in the DBTI speculation window, even with SMT workloads.

However, neither disabling SMT nor use of this additional MSR bit address scenarios where a *load-once* gadget may be of concern, such as in combination with BCB attacks. AMD recommend developers should "inspect their code for any unmitigated cases", but adding serialization to all locations of concern may not be a practical solution in all cases. Invoking IBPB on domain transitions can be an alternative, but may impact performance; AMD measure IBPB as 10k cycles on Zen, although it is significantly faster on newer processors [8].

An alternative approach is to ensure that secrets belonging to other security domains are not mapped into the address space of privileged code, ensuring that transient execution attacks cannot access them – and thus rendering these mitigations largely unnecessary. Recent research has shown not only that this can be done with low overhead, but also that it can be deployed in practice in a commercial cloud (Azure) [44].

7. Related Work

Our research builds upon the original work on Spectre [27], including the detailed Google Project Zero writeup [18], as well as research on newer variants of BTI-style attacks such as SpectreRSB [28] (aka ret2spec [31]) and Branch History Injection [9] (BHI). The possibility of a speculation window despite use of serialization, as in LFENCE/JMP, was suggested by Paul Turner as part of the motivation for retpoline [40]. Other mitigations for such attacks include randpoline [39] as well as a variety of more targeted software and hardware defenses [12]. As discussed above, Retbleed [42] showed that a speculation window was present for RET instructions on AMD processors due to Branch Type Confusion, and a later addendum [43] briefly discussed the BTC-NOBR case (similar to DBTI) and the potential for BCB-style attacks. Pawel Wieczorkiewicz also published some analysis of the speculation window provided by 'fall-through' transient execution across direct branches [41], which discusses how store-to-load forwarding may provide an alternative method of exploiting small windows.

Research analyzing how BTI-style attacks and other building blocks can be used in practice also provides an improved understanding of how hardware works, what software expects, and what protection mitigations may need to provide. In particular, SGXpectre [14] investigated BTI attacks against SGX enclaves, Zhang et al. [47] provided a detailed analysis of branch predictors and other related microarchitecture details of Intel CPUs, and Mcilroy et al. [33] analyze transient execution vulnerabilities and argue that, at least for Chrome, process isolation is the only realistic mitigation. Speculator [32] presents a framework and case studies using performance counters to analyze processor behavior related to speculative execution. Some formal approaches for analyzing branch target predictor attacks [13], [17] also consider speculation windows, but typically from a different perspective – to determine the *maximum* number of instructions which may be speculatively executed.

Finally, there is a range of recent research focusing on microarchitectural side channels, and how they could be used in disclosure gadgets, such as BranchScope [16] and the LRU work by Xiong et al. [45], which we discussed in subsection 4.1. Other key recent research includes SMoTherSpectre [10] which uses SMT port contention as a side-channel rather than as an attack, and NetSpectre [35], which uses timing differences due to use of AVX2 instructions. For a fairly comprehensive summary, we refer to the survey of transient execution attacks and defenses by Canella et al. [11].

8. Conclusion

We have shown that mitigations against branch target prediction attacks can be compromised due to speculation windows arising from sources of latency which had not previously been considered. Our case studies have demonstrated that several standard mitigations are ineffective, or incomplete in some circumstances.

First, we proved that branch latency can be an issue in practice, demonstrating that the resulting speculation windows can break the LFENCE/JMP mitigation for BTI-style attacks. We then showed that issues with branch latency are not limited to indirect branches, discovering that BTI-style attacks can be possible in the absence of indirect branches on some processors. And finally, we demonstrated that it is also valuable to evaluate mitigations that do not appear to rely on suppressing speculation windows, by finding that some implementations of IBPB may be ineffective for mitigating SpectreRSB-style attacks. The affected vendors have updated (or plan to update) their mitigation guidance for all three of these cases.

In practice, the exploitability of branch target prediction attacks depends on a range of factors beyond the existence of a speculation window, such as whether an attacker is able to execute code locally, as well as the availability of suitable call sites (where relevant) and disclosure gadgets. However, in cases where such attacks are a real concern, our work shows that mitigations for these branch target prediction attacks should be evaluated in a more systematic and comprehensive manner.

Acknowledgements

This work was partially inspired by people involved in the original Spectre response. In particular we would like to thank all the involved people at Google and Intel, many current and previous members of Intel STORM (in particular, Rodrigo Branco, Kekai Hu, Emma Benoit, Igor Chervatyuk, Lisa Aichele, and Thais Moreira Hamasaki), the Vrije Universiteit Amsterdam for their thought-provoking BHI research, the partners who motivated us to pursue this investigation, and finally AMD for their swift and friendly response to our findings involving their processors.

Our work also benefited from AMD's feedback during the disclosure process; in particular, we would like to acknowledge their report of the *load-load* case (without SMT) for LFENCE/JMP on their processors; we expanded our work to also include results for this case.

Disclosure

We engaged in coordinated disclosure for all issues discovered during our research. We reported our LFENCE/JMP findings to AMD in November 2021, originally focused on SMT; they agreed to a March 2022 disclosure date, and assigned CVE-2021-26401. We reported our IBPB findings to AMD in February 2022, which were disclosed in November (as CVE-2022-23824).

Finally, we reported DBTI (including our SMT analysis and the potential for BCB-style exploitation) to AMD in June 2022, who acknowledged that their internal researchers had already discovered most of these issues, which AMD disclosed (as CVE-2022-23825) in July 2022 under the umbrella of Branch Type Confusion. Their BTC guidance was later updated (in November) to cover our SMT findings.

Data Availability

Code to support our paper can be found at github.com/IntelSTORMteam/win-the-race. This does not include all the code needed to reproduce our case studies, since such code also inherently demonstrates how to bypass the relevant mitigations, and we have been asked not to share such proof-of-concept code. However, we believe that our paper provides sufficient information to reproduce all of our results, and we hope that our example code demonstrates how we applied our methodology in practice.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[2] AMD. Amd64 technology indirect branch control extension. Revision 4.10.18, April 2018.

[3] AMD. Software techniques for managing speculation on amd processors. Revision 7.10.18, July 2018.

[4] AMD. Software optimization guide for amd family 17h models 30h and greater processors, August 2019.

[5] AMD. AMD64 Architecture Programmer's Manual, Volume 2, November 2021.

[6] AMD. AMD64 Architecture Programmer's Manual, Volume 3, November 2021.

[7] AMD. Lfence/jmp mitigation update for cve-2017-5715, August 2022.

[8] AMD. Technical guidance for mitiating branch type confusion. Revision 1.0, July 2022.

[9] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, 2022.

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

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[12] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N Khasawneh. Evolution of defenses against transient-execution attacks. In *GLSVLSI*, 2020.

[13] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *S&P*, 2022.

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, 2019.

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

[16] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

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

[18] Jann Horn. Reading privileged memory with a side-channel. Google Project Zero, 2018.

[19] Intel. Analyzing potential bounds check bypass vulnerabilities, January 2018.

[20] Intel. Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088, 2018.

[21] Intel. Speculative execution side channel mitigations. Revision 2.0, May 2018.

[22] Intel. Retpoline: A Branch Target Injection Mitigation. Version 2.0, 2019.

[23] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, December 2021.

[24] Intel. Intel 64 and ia-32 architectures optimization reference manual, February 2022.

[25] Intel. Post-barrier return stack buffer predictions, August 2022.

[26] Daniel A Jiménez, Stephen W Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *MICRO*, 2000.

[27] 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*, 2019.

[28] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT*, 2018.

[29] Tom Lendacky. Re: Avoid speculative indirect calls in kernel. https://lkml.org/lkml/2018/1/4/742, 2018.

[30] Moritz Lipp, Daniel Gruss, and Michael Schwarz. Amd prefetch attacks through power and time. In *USENIX Security*, 2022.

[31] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *SIGSAC*, 2018.

[32] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *ACSAC*, 2019.

[33] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[34] Miller, Matt. Mitigating speculative execution side channel hardware vulnerabilities, March 2018.

[35] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[36] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing of registers: Reviving foreshadow. *arXiv preprint arXiv:2008.02307*, 2020.

[37] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with venkman. *arXiv preprint arXiv:1903.10651*, 2019.

[38] Ryan Smith and Gavin Bonshor. Amd zen 4 ryzen 9 7950x and ryzen 5 7600x review: Retaking the high-end, September 2022.

[39] Ke Sun, Kekai Hu, Henrique Kawakami, Marion Marschalek, and Rodrigo Branco. A software mitigation approach for branch target injection attack. v1.42, Intel STORM, 2019.

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

[41] Pawel Wieczorkiewicz. The amd branch (mis)predictor part 2: Where no cpu has gone before, March 2022.

[42] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[43] Johannes Wikner, Daniel Trujillo, and Kaveh Razavi. Addendum to retbleed: Arbitrary speculative code execution with return instructions, July 2022.

[44] Hongyan Xia, David Zhang, Wei Liu, Istvan Haller, Bruce Sherwin, and David Chisnall. A secret-free hypervisor: Rethinking isolation in the age of speculative vulnerabilities. In *S&P*, 2022.

[45] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. In *HPCA*, 2020.

[46] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[47] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*, 2020.

A. LFENCE

Intel's Software Developer Manual [23] states that "LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes". Although the AMD64 Architecture Programmer's Manual [6] states that LFENCE only "assures that the system completes all previous loads before executing subsequent loads", AMD's software guidance [3] documents an MSR which makes LFENCE dispatch-serializing: "upon encountering an LFENCE when the MSR bit is set, dispatch will stop until the LFENCE instruction becomes the oldest instruction in the machine". AMD also documents a CPUID enumeration for processors where LFENCE is dispatch-serializing by default.

We confirmed that the documented MSR bit is set by default on Linux on the relevant AMD CPUs, and that it was set during our experiments.

B. LFENCE/JMP history

Google's motivation for retpoline explicitly pointed out that serialization was insufficient, since "the speculative execution here is a property of the hardware itself" [40]. This is likely the reason that LFENCE/JMP mitigation was not recommended by Intel, due to the lack of architectural guarantees and the availability of alternative mitigations.

AMD's guidance [3], on the other hand, stated that LFENCE/JMP ("mitigation V2-2") is a suitable mitigation for BTI attacks on "all AMD processors" since "*the speculative execution window is not large enough to be exploited*". Despite concerns from the Linux community given that LFENCE/JMP was "not *quite* good enough" [29] on Intel CPUs, AMD confirmed that the mitigation is sufficient [29].

C. LFENCE/JMP: alternative disclosure gadgets

Our results show that – even when SMT is disabled or unavailable – LFENCE/JMP permits a relatively large speculation window on some processors. However, this window does not appear large enough to execute larger gadgets on many processors. Table 11 show results when attempting to transiently execute some other alternative disclosure gadgets within the LFENCE/JMP window (without SMT). Although we also attempted to implement a BPU-based side channel (as in [16]), we were not able to create a channel within the LFENCE/JMP window.

With SMT workloads, the speculative execution window beyond a LFENCE/JMP sequence and IBPB can be significantly larger than the window needed for a "universal read" disclosure gadget. For these larger speculation windows, a key question remains: whether an attacker would need to control register values at the indirect branch. If a speculation window is large enough to permit a third dependent load, such control would not be necessary, which could significantly increase the number of viable disclosure gadgets. This would allow, for example, forms of "universal read" gadgets which read the address of the desired data by the gadget from the stack, rather than needing it to be in a register.

We implemented an artificial proof-of-concept attack which infers the contents of a string from memory, byte-by-byte, using FLUSH+RELOAD to observe the resulting cache effects. The gadget used is shown below; it is similar to the "memory-disclosure" gadget evaluated above, but the target address is read indirectly from memory rather than taken directly from an attacker-controlled register. We also mask the value with 0xFF, which allows easier inference of memory contents on a per-byte basis (similar results could be obtained in a smaller window by just using one instruction such as MOVZBQ):

```
1    mov rbx, [rbx]
2    mov rdx, [rbx]
3    and rdx, 0xff
4    shl rdx, 0xc
5    mov rax, [rdx+rcx]
```

We combined this attack with the "best-performing" SMT workload (from Table 5) for each processor; Table 12 shows the results for LFENCE/JMP. As can be seen,

TABLE 11: Test success rate for minimal variants of the "memory-disclosure" gadget (without SMT).

	Goldmont Plus	Tremont	Sunny Cove	Willow Cove	Golden Cove	Gracemont	Zen	Zen+	Zen 2	Zen 3
Minimal load	0%	0%	0%	0%	0%	> 99%	> 99%	> 99%	> 99%	0%
Store	0%	0%	0%	0%	0%	0%	> 99%	> 99%	> 99%	0%
Prefetch	0%	0%	0%	0%	0%	> 99%	> 99%	> 99%	> 99%	0%
Flush	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Complex load	0%	0%	0%	0%	0%	> 99%	> 99%	> 99%	> 99%	0%

TABLE 12: Proof-of-concept three-load variant results: does the PoC obtain the secret?

	Goldmont Plus	Tremont	Sunny Cove	Willow Cove	Golden Cove	Gracemont	Zen	Zen+	Zen 2	Zen 3
No SMT	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
SMT workload	N/A	N/A	✓	✓	✓	N/A	✓	✓	✓	✓

TABLE 13: Common mitigations for BTI-style attacks.

Mitigation	Summary	Notes
LFENCE/JMP	Minimize speculation window	
Retpoline	Redirect speculation using RET	Software-based.
IBRS	Restrict indirect branch predictions	
Enhanced IBRS	Isolate indirect branch predictions between modes	Supported by recent Intel processors.
IBPB	Barrier to isolate predictions between security domains	
Predictor controls	Fine-grained disables for specific indirect predictors	Supported by newer Intel processors.

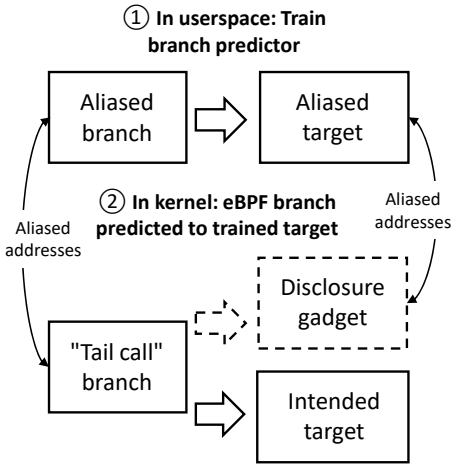

Figure 3: The eBPF proof-of-concept first uses a userspace branch for training, then executes a branch in kernel mode which (due to aliasing) is predicted to a disclosure gadget.

even in the Zen 3 case (where we have yet to isolate the effective instructions in the SMT workload), this more generic disclosure gadget can fit in the speculation window despite the use of the LFENCE/JMP mitigation.

D. eBPF PoC

We obtain the needed indirect branch for the PoC described in Section 6.4 using the "tail call" mechanism, which transfers execution to another eBPF program. Before this indirect branch, a series of branches are executed in eBPF code so that the BHB at the indirect branch is constant, which is used by the indirect predictor to predict the target of the indirect branch. The target of the kernel branch is trained by executing an identical set of branches in a userspace application, with their addresses aliased to the corresponding kernel branches, as shown in Figure 3. We considered obtaining kernel addresses to be out-of-scope (recent academic work [30] has shown that fine-grained KASLR can be bypassed on AMD CPUs, and presumably cache side channels can be used for this on Intel CPUs), and instead used a small setuid program which printed the kernel address of our eBPF program.

When the kernel branch is executed, the JITed code can store any value accessible to our eBPF program (including a kernel pointer) in R8 (eBPF r5), and an arbitrary user-controlled value in R11. In our PoC, R8 contains a pointer to an eBPF map – used as an FLUSH+RELOAD area for our cache-timing side channel – and R11 contains the kernel pointer to transiently read from.

E. Mitigations

Table 13 summarizes common mitigations for BTI-style attacks, as discussed in Section 2.

F. Example SMT workload

Listing 3 shows the core loop used by our XOR SMT workload. Source for this and other workloads can be found in our code repository.

```
xorloop:
xor eax, 0x11111111
xor ebx, 0x22222222
xor ecx, 0x33333333
xor edx, 0x44444444
xor eax, 0x55555555
xor ebx, 0x66666666
xor ecx, 0x77777777
xor edx, 0x88888888
xor eax, 0x99999999
xor ebx, 0xaaaaaaaa
jmp xorloop
```

Listing 3: XOR SMT workload