InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2

Sander Wiebing* Alvise de Faveri Tron* Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

Abstract

Spectre v2 is one of the most severe transient execution vulnerabilities, as it allows an unprivileged attacker to lure a privileged (e.g., kernel) victim into speculatively jumping to a chosen *gadget*, which then leaks data back to the attacker. Spectre v2 is hard to eradicate. Even on last-generation Intel CPUs, security hinges on the unavailability of exploitable gadgets. Nonetheless, with (i) deployed mitigations—eIBRS, no-eBPF, (Fine)IBT—all aimed at hindering many usable gadgets, (ii) existing exploits relying on now-privileged features (eBPF), and (iii) recent Linux kernel gadget analysis studies reporting no exploitable gadgets, the common belief is that there is *no* residual attack surface of practical concern.

In this paper, we challenge this belief and uncover a significant residual attack surface for cross-privilege Spectre-v2 attacks. To this end, we present *InSpectre Gadget*, a new gadget analysis tool for in-depth inspection of Spectre gadgets. Unlike existing tools, ours performs generic *constraint analysis* and models knowledge of advanced *exploitation techniques* to accurately reason over gadget exploitability in an automated fashion. We show that our tool can not only uncover new (unconventionally) exploitable gadgets in the Linux kernel, but that those gadgets are sufficient to bypass all deployed Intel mitigations. As a demonstration, we present the first *native* Spectre-v2 exploit against the Linux kernel on last-generation Intel CPUs, based on the recent BHI variant and able to leak arbitrary kernel memory at 3.5 kB/sec. We also present a number of gadgets and exploitation techniques to bypass the recent FineIBT mitigation, along with a case study on a 13th Gen Intel CPU that can leak kernel memory at 18 bytes/sec.

1 Introduction

As the community slowly comes to grips with various forms of transient execution attacks [16, 32, 34, 35, 38, 54], Spectre v2 or Branch Target Injection (BTI) [2] remains one of the most severe ones, able to transiently divert the control flow of a program. If attackers can find a snippet of code that encodes secret data into the microarchitectural state, i.e., a (disclosure) *gadget*, they can force a victim program, e.g., the kernel, to transiently jump to it. Even in the face of hardware mitigations such as eIBRS, researchers have shown that Spectre v2 can still leak secret data across privilege levels on Intel systems through what is known as Branch History Injection (BHI) [16].

However, neither academia [16] nor industry [8] ever found an exploitable "*native*" gadget and the only existing exploit relies on a gadget injected by the authors themselves using eBPF. Since then, new advanced mitigations such as Indirect Branch Tracking (IBT) and its recent fine-grained counterpart FineIBT, have reduced the set of usable gadgets even (much) further. As a result, it is a common belief that deployed mitigations such as eIBRS and privileged eBPF (now default in all popular Linux distributions) are sufficient to eliminate the cross-privilege Spectre-v2 attack surface—and even more so in combination with the opt-in (Fine)IBT mitigations.

To challenge this belief, our key observation is that current techniques to identify such gadgets either *overfit* "standard" patterns—identifying only gadgets that look like a handful of known Spectre examples and ignoring less conventional patterns—or grossly *overapproximate*—identifying many *potential* gadgets of which exploitability is highly uncertain. Examples of the latter are approaches that identify gadgets based on their high-level data flow, leaving exploitability to manual analysis [16, 24, 31]. Examples of the former include all pattern-based gadget scanners [40, 42, 43], but also all simplifying and self-limiting gadget definitions [8]. Overly constraining the definition of a gadget is dangerous, because even snippets that do not meet all the preconditions of a standard gadget can still leak data with advanced exploitation techniques [24, 31, 54]—leaving a gap between what attackers need for exploitation and what vendors and developers consider for mitigation.

In this paper, we present *InSpectre Gadget*, an in-depth Spectre gadget inspector that uses symbolic execution to accurately reason about exploitability of usable gadgets. To this end, our tool explicitly models data *constraints* and knowledge of advanced *exploitation techniques*. This strategy relaxes the common preconditions of standard gadgets, while still avoiding the common overapproximations that would otherwise report many unexploitable gadgets. Moreover, it provides the analyst with insights into exploitability characteristics, such as the exploitation techniques required, the constraints to be met, the values that can be leaked, etc.

Scanning the Linux Kernel, InSpectre Gadget finds 1,565 gadgets leading to secret transmission, a significant residual attack surface. Furthermore, it uncovers hundreds *dispatch gadgets*, i.e., gadgets containing an indirect branch to an attacker-controlled target and, as we will show, providing the attacker with a variety of interesting capabilities for exploitation in face of deployed mitigations. Examples include increasing control over registers, expanding the set of reachable gadgets, or crafting disclosure gadgets by chaining multiple loads. To demonstrate the practicality of our findings, we use the reported gadgets to implement the first *native* Spectre-v2 exploit against the Linux kernel on last-generation Intel CPUs. Our exploit is based on the BHI variant and is able to leak kernel memory at 3.5 kB/sec without eBPF.

Furthermore, in contrast to previous findings, we show the recent (Fine)IBT mitigations still allow transient execution of between 4 and 6 (unchecked) dependent loads. To exploit the resulting attack surface, we showcase a number of gadgets and exploitation techniques, along with a case study on a 13th Gen Intel CPU, leaking kernel memory at 18 bytes/sec.

Contributions. To summarize our contributions:

1. We build InSpectre Gadget, a gadget inspector that evaluates exploitability of potential gadgets, incorporating data constraints and knowledge of advanced exploitation techniques. InSpectre Gadget is available at https://github.com/vusec/inspectre-gadget, along with a database of gadgets found for Linux kernel v6.6-rc4.

2. We present the first native (no-eBPF) BHI exploit on the latest Linux kernel and last-generation Intel CPUs (CVE-2024-2201).

3. We analyze the effectiveness of both the IBT and FineIBT mitigations and demonstrate they are insufficient to hinder native BHI exploitation.

4. We uncover a large presence of dispatch gadgets in the kernel, showing how an attacker can abuse them to further the attack surface and bypass deployed mitigations.

2 Background

2.1 Transient Execution Attacks

Modern CPUs rely on a multitude of speculation mechanisms to achieve better performance. For example, whenever a CPU encounters a control flow instruction like a conditional branch or an indirect branch, the correct target might not be known yet. To avoid stalling, the CPU uses a set of internal structures, called *predictors*, to determine the next instruction to fetch, and starts *speculatively* executing instructions. If the speculation is later revealed to be incorrect, the CPU will undo the effects of the *transient* execution and restart from the correct jump target. However, traces of the transient computation can still be observed in the shared microarchitectural state, e.g. in the cache. An attacker can then measure these traces with techniques such as FLUSH+RELOAD [55] and PRIME+PROBE [36] and recover the values of registers and memory used during the transient computation.

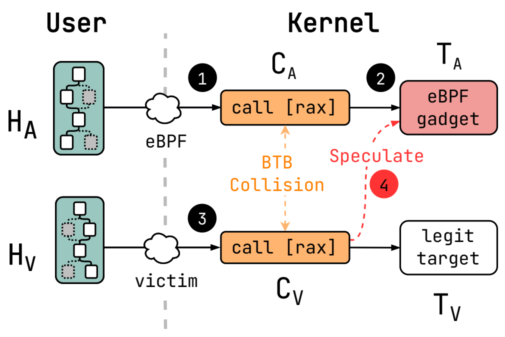


Figure 1: **The BHI attack.** The attacker first triggers the branch C_A with history H_A ①, which inserts the target T_A into the BTB ②, then triggers the victim branch C_V with history H_V ③. The histories are crafted so that C_A and C_V share the same BTB entry, so from C_V the CPU speculates to T_A ④.

2.2 Spectre v2

In 2018, the disclosure of Spectre [32] famously demonstrated how speculation can be used to leak data across security domains. One variant presented in the paper, originally known as Spectre v2 or Branch Target Injection (BTI), shows how speculation of indirect branches can be used to transiently divert the control flow of a program and redirect it to an attacker-chosen location. The attack works by poisoning one of the CPU predictors, the Branch Target Buffer (BTB), which is used to decide where to jump on indirect branch speculation. Initially, mitigations were proposed at the software level and, later, in-silicon mitigations such as Intel eIBRS [6] an ARM CSV2 [14] were added to newer generations of CPUs to isolate predictions across privilege levels.

2.3 Branch History Injection

In 2022, Branch History Injection (BHI) [16] showed that, despite mitigations, cross-privilege Spectre v2 is still possible on latest Intel CPUs by poisoning the Branch History Buffer (BHB). Figure 1 provides a high-level overview of the attack.

In summary, by executing a sequence of conditional branches (H_A and H_V) right before performing a system call, an unprivileged attacker can cause the CPU to transiently jump to a chosen target (T_A) when speculating over an indirect call in the kernel (C_V). This happens because the CPU picks the speculative target for C_V from a shared structure, the BTB, that is indexed using both the address of the instruction *and the history* of previous conditional branches, which is stored in the Branch History Buffer (BHB). Finding the right combination of histories that will result in a collision can be done with brute-forcing.

To ensure the injected target, T_A, contains a disclosure gadget, the original BHI attack relied on the presence of the extended Berkeley Packet Filter (eBPF), through which an unprivileged user can craft code that lives in the kernel.

2.4 Defenses

As a recommended mitigation for BHI, Intel has advised to disable unprivileged eBPF, which is now disabled by default in the Linux kernel, and to mitigate potential disclosure gadgets by prepending a `LFENCE` instruction to them [1].

Attack Surface Analysis. The original BHI paper presented an initial estimate of the BHI attack surface beyond eBPF using simple data-flow analysis and a loose definition of disclosure gadget [16]. The analysis pinpointed 1,177 *potential* gadgets in the Linux kernel, but with no insights into their exploitability. Later, Intel researchers statically analyzed the Linux kernel [8], adopting a more refined data-flow-based approach and finding an order of magnitude more potential gadgets. Again, with the analysis unable to automatically reason over exploitability, Intel researchers resorted to manually assessing exploitability of the 8 simplest (linear) gadgets. No gadgets were deemed exploitable (6 due to reachability issues, 2 due to leakage constraints). Ultimately, with many potential gadgets uncovered by both scanners but no evidence of practical exploitability, no additional mitigations were deployed.

Hardware mitigations. On the hardware front, Intel has proposed a hardware mitigation, the `BHI_DIS_S` indirect predictor control, which prevents the CPU from selecting BTB entries based on history coming from lower security domains. As the performance overhead is nontrivial, future CPUs might come with an optimized version, namely `BHI_NO`. At the time of writing, while Alder Lake and Raptor Lake Intel CPUs support `BHI_DIS_S` after applying a microcode update, the Linux Kernel does not have support to enable this feature.

Advanced software mitigations. Additional software mitigations, like Retpoline [10] or a software BHB-clearing sequence [1], have been proposed as spot fixes. However, they come with a prohibitive performance cost, discouraging practical deployment. For Linux, in particular, the software BHB-clearing sequence recommended by Intel has not been implemented at all, as developers rely on unprivileged eBPF being "the only known real-world BHB attack vector" [11].

IBT. Indirect Branch Tracking (IBT) [48] is a defense for code-reuse attacks, such as return-oriented programming [46] and jump-oriented programming [19], which ensures that indirect branches always jump to an intended target. Intel CPUs implement a coarse-grained version of IBT in hardware, which ensures that every indirect branch lands on a special instruction (`endbr32` or `endbr64`), which has to be inserted by the compiler. While IBT was originally designed to address architectural control-flow hijacking, it is also part of Intel's mitigation guidance for BHI [1]. This is to provide defense-in-depth against speculative control-flow hijacking, limiting—somewhat similarly to the existing eIBRS—the possible Spectre-v2 disclosure gadget locations to the beginning of *any* indirect branch target in the kernel. Support for IBT was added to Intel processors with the Tiger Lake series [7] and is enabled by default from Linux kernel v6.2 [4].

FineIBT. Researchers have recently proposed a finer-grained IBT variant in software, called FineIBT [23]. The idea is to instrument the caller of each IBT-guarded indirect call to load a unique value into a register as well as the callee to check said value. If the value is different from the expected one, the execution path is directed to an illegal instruction to abort execution.

This further restricts indirect calls to (architecturally or speculatively) target only compliant callees. Support for FineIBT was recently introduced in Linux kernel v6.2 [5]. Since FineIBT relies on Clang-instrumented kernels [23], to our knowledge, it is not yet enabled by default in any Linux distribution (unlike its IBT building block).

3 Threat Model

We consider a traditional cross-privilege Spectre-v2 threat model, with a local unprivileged attacker seeking to disclose information from a privileged victim, such as the operating system kernel or the hypervisor. We specifically focus on a victim Linux kernel, running with all default Spectre-v2 mitigations on last-generation Intel CPUs such as eIBRS [6] and privileged eBPF. Finally, we assume other classes of vulnerabilities (e.g., memory errors) are subject of orthogonal mitigations and not part of the attack surface under study.

4 Overview

To perform a Spectre-v2 attack against the kernel on last-generation Intel systems, one must target an indirect branch in the kernel that jumps to a disclosure gadget. Using BHI, one can then speculatively hijack a victim branch to the chosen gadget. InSpectre Gadget aids the analyst in choosing a suitable gadget, following the workflow depicted in Figure 2.

As shown in the figure, the analyst provides InSpectre Gadget with a kernel image and a list of candidate gadgets in input. Our tool inspects each candidate for a fixed number

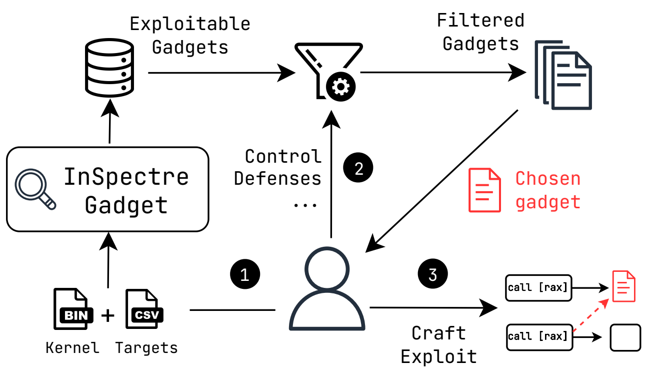


Figure 2: **InSpectre gadget workflow.** The analyst provides a kernel image and a list of target addresses to InSpectre Gadget ①, which performs in-depth inspection to find gadgets that can leak secrets and output their characteristics. The gadgets can be filtered ② based on the available attacker-controlled registers and the mitigations enabled, and used to craft Spectre-v2 exploits against the kernel ③.

of basic blocks with symbolic execution and returns a list of gadgets that lead to the transmission of a secret. Along with the gadgets, our tool outputs a number of gadget characteristics: the advanced exploitation techniques required (if any), the constraints that have to be met, the registers that have to be controlled, and the values that can be leaked. Such characteristics are stored into a database, which the analyst can later filter according to what targets are reachable, what registers are controlled when the speculative hijack happens, and what mitigations are enabled for a given target. Additionally, an annotated assembly file is generated for each gadget to give the analyst a quick overview of the gadget, as shown in Appendix C. The resulting gadgets can then be exploited to mount end-to-end Spectre-v2 kernel attacks.

In the next sections, we first elaborate on how InSpectre Gadget models gadgets (constraints, exploitability, etc.) and how it then performs exploitation-aware gadget inspection. Next, we demonstrate how an attacker can use its output to mount an end-to-end BHI attack against the Linux kernel.

5 InSpectre Gadget

A recurring problem in transient execution attacks is evaluating if a given instruction sequence can leak a secret via a—typically cache [27, 33, 36, 41, 47, 55]—covert channel. To leak a secret through the cache, an attacker needs to open a *speculation window* and accommodate a *disclosure gadget*. In this section, we explain that practical disclosure gadgets extend well beyond those that fit existing narrow definitions. Next, we show how InSpectre Gadget uses symbolic execution to analyze the candidate gadgets for Spectre-v2 exploitability.

Listing 1: Standard Spectre disclosure gadget.

```
1 // Load from attacker-controlled address
2 uint64_t secret = *attacker;
3 // Mask the loaded value
4 uint8_t secretByte = (secret & 0xFF);
5 // Shift the result
6 uint32_t tsecret = secretByte << 9;
7 // Use transmission secret as index.
8 uint64_t transmission = *(tbase + tsecret);
```

5.1 Standard Gadgets

Today's tools typically concentrate on finding speculation windows and overapproximating exploitability—leaving the analysis of disclosure gadgets to the human analyst. Unfortunately, the complexity of such exploitability analysis is daunting and, unsurprisingly, analysts generally look for *standard* Spectre disclosure gadgets, such as the one in Listing 1.

In a standard (or "perfect") gadget, the CPU loads a *secret* from an attacker-controlled address. The loaded secret must be sufficiently small, either by nature, or as the result of a bitmask (Line 4), to serve as an index into a second buffer—ideally shared with the attacker. This second buffer is known as the *reload buffer*. Moreover, to ensure that each value of the secret corresponds to a different cache line, the gadget should *shift* the value left by some stride (Line 6). The result is known as the *transmitted secret*. As the gadget subsequently adds it to a second attacker-controlled value which we refer to as the *transmission base* and dereferences the resulting value, it inadvertently establishes a *transmission* through a cache covert channel. Specifically, by iterating over the reload buffer with the same stride while timing the accesses, attackers can infer that the secret value is the index of the buffer element for which the access is fast (because it is in the cache).

5.2 Exploitation-Aware Gadget Analysis

Standard gadgets are the most intuitive to understand and the most straightforward to exploit, but they are by no means the only exploitable ones. While limiting the analysis to standard gadgets reduces the complexity, attackers are under no obligation to respect such restrictions. BLINDSIDE [24], KASPER [31], PACMAN [44], and RETBLEED [54], for example, all exploit gadgets that deviate from such narrow definitions. Moreover, besides leaking information through the cache, attackers may avail themselves of a myriad of other covert channels [18, 22, 25, 45, 52]. In this section, we relax the assumptions for standard gadgets, describe the additional challenge posed by each relaxation, and where appropriate, explain how we can meet the challenge and still leak secrets.

C1. Base not controlled. To perform FLUSH+RELOAD an attacker needs to be in control not only of the secret address, but also of the transmission base, so that the transmission load falls inside of the reload buffer. However, if the base is not

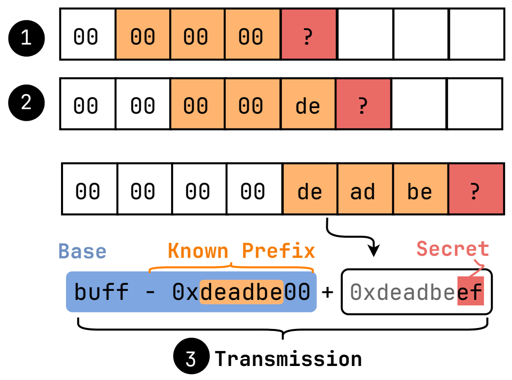


Figure 3: **Known-prefix technique.** The attacker first points the secret address to some known data ①. Then, by shifting the address ②, small portions of the secret are revealed. With that, one can adjust the base of subsequent transmissions ③.

controlled, attackers can still perform PRIME+PROBE [36].

C2. Secret entropy too big. If the code does not mask the secret prior to its transmission, the entropy impedes its recovery by the attacker who would have to probe too many memory locations. However, if the attackers control the transmission base, they can still exploit such gadgets by repurposing a technique pioneered in earlier attacks [24, 52, 54], which we call the *known-prefix* technique. First, the attacker chooses a location in memory near the secret that contains a small known value, and uses that as the secret address. Then, by increasingly shifting the address, the attacker makes sure that only a few bytes of the secret are unknown at any given transmission, and adjusts the transmission base according to the bytes that are already known (see Figure 3).

C3. Max secret too high. Another problem that may occur when leaking a large secret value is that the transmission address might end up outside the valid address space. However, if enough bits of the transmission base are under attacker control, one can adjust the base to overflow the secret value, ensuring that the transmission always occurs in the valid address space. We call this technique *base adjusting* and it is often used in conjunction with the known-prefix technique.

C4. Secret too small. If the gadget does not shift the secret value prior to transmission, the lower bits cannot be recovered through cache covert channels, since nearby addresses belong to the same cache line. However, if at least one byte of the secret is above cache-line granularity, we can use the known-prefix technique to leak the uppermost byte in each iteration. Otherwise, an attacker may use the *sliding* technique of RETBLEED [54], which exploits the fact that prefetchers generally do not prefetch cache lines across pages [49]. An attacker may now adjust the base address to be near a page boundary, so that even a one-bit difference in the secret value will result in the transmission being observed on another memory page. Doing so eliminates any cache and prefetcher noise that would otherwise make two different values indistinguishable.

C5. Base aliasing. In some cases, even if the transmission base is attacker-controlled, the base cannot be chosen without influencing the secret address—a phenomenon we refer to as *aliasing*. This may occur, for instance, if the base is computed from a value that is also used to compute the secret address. In this case, an attacker can still leak values using PRIME+PROBE. However, not all secret addresses may be targeted as the probe region must reside within mapped memory. Specifically, since the probe region typically follows the secret address, the final secret addresses result in a non-mapped probe region. Other, more complex dependencies between the base and the secret address out of scope. InSpectre Gadget marks these cases as not (easily) exploitable.

C6. Non-linear gadgets. An implicit assumption of most static analysis techniques for gadget scanning is that all the instructions have to be in the same basic block. Since modern CPUs all support nested speculation, this is not a limitation in practice. The attacker can either train branches or simply use static prediction to ensure that the transmission gadget is reached during speculative execution. An attacker can also use SMT contention, as demonstrated by prior work [39], to delay branch resolution and create large speculation windows that can accommodate multiple basic blocks.

C7. Other transmitters. Finally, a plethora of covert channels exist in modern CPUs outside of caches (e.g., the TLB [37]) and one should flexibly support different transmitters. For brevity, we focus our main analysis on classic secret-dependent data load/store transmitters and later show our tool can be easily extended to support other vectors such as the recent SLAM covert channel [29].

5.3 Design

While advanced exploitation techniques relax the assumptions for standard gadgets, each has its own requirements and constraints for applicability. To reason over the constraints and to assess if the code satisfies them, InSpectre Gadget employs *symbolic execution*—expressing variables as *symbols*, and exploring all the possible directions of a program's control flow at the same time, while recording the corresponding *symbolic constraints*. To this end, we build our tool on top of ANGR [50], a symbolic execution engine widely used in the field of software security and reverse engineering.

Although symbolic execution quickly leads to *state explosion* in the general case, InSpectre Gadget explores only a small fraction of the program's control flow. In particular, since the number of instructions executed during a speculation window is limited, symbolic execution can easily explore multiple paths, while keeping the number of explored states small. All the steps described below are performed automatically by InSpectre Gadget.

Tracking attacker control. We start our analysis by substituting all the values stored in registers and on the stack with

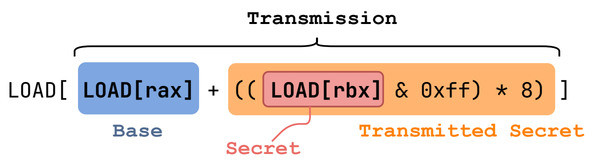


Figure 4: **Anatomy of a transmission.** Once a potential transmission is identified through symbolic execution, InSpectre Gadget dissects its symbolic expression into components, which are then analyzed to reason about exploitability.

symbolic variables, which for now we assume are *attacker-controlled*, and marked as such. Next, we symbolically execute code starting from a given location. On each load, we produce a new symbolic value with a label attached to it. If the load comes from an attacker-controlled symbol, it is marked as a *potential secret*. If the address comes from a potential secret, it is marked as a *potential transmission*. Note that *potential secret* implies *attacker-controlled*, since it is any value loaded from an attacker-chosen location. Similarly, *potential transmission* implies *potential secret*. This strategy allows us to track of attacker control through complex chains of loads.

Store-to-Load Forwarding (STL). We model Store-To-Load Forwarding by keeping a list of all the symbolic stores, and checking this list for each of the loads encountered by the symbolic execution engine. If the symbolic expression of the load address aliases with that of a previous store, we forward the stored value to the load. Store-to-Load forwarding can be enabled or disabled with a runtime flag.

Potential transmissions. We let the symbolic execution engine run for a configurable number of basic blocks, recording all the constraints that might be added, for instance by `cmove` or branch instructions. We also record the symbolic expression of each load. Finally, after the scanning phase is finished, the scanner reports a list of potential transmissions, i.e., loads whose symbolic address has been marked as a potential secret.

In a second phase, we inspect the AST of the symbolic expression of each potential transmission, identifying the transmission base, transmitted secret, and secret address. A high-level example is shown in Figure 4. Finally, we check if the base depends on any value used to construct the secret address, perform a range analysis to infer the minimum, maximum and stride of all the transmission components, and perform an *inferable-bits* analysis to infer which bits of the secret end up in the transmission and at which position. This comprehensive analysis is crucial for accurate exploitability reasoning. For instance, data-flow information alone is insufficient to determine the controllability requirements to be met.

Gadget reasoning. With this information, we can finally reason about each gadget. All the properties found during the analysis, along with a list of registers that the attacker needs to control for each gadget, are saved in a database. We now use a *reasoner* to model exploitation techniques with database queries. The reasoner inserts columns indicating which gadgets can leak a secret, and, if needed, which techniques are required. For instance, if a gadget has a high secret entropy (i.e., number of transmitted bits > 16), the reasoner checks if we can perform the known-prefix technique (i.e., secret address is attacker-controlled with granularity <= 16 bits).

5.4 Evaluation

To evaluate the ability of InSpectre Gadget to uncover a new attack surface, we analyzed the Linux kernel version 6.6-rc4 (latest at time of writing) with the default configuration. By listing all the code locations that contain an `endbr` instruction and looking at the symbol table, we found a total of 35,212 indirect call targets (have a symbol), and 7,562 indirect jump targets (have no associated symbol). InSpectre Gadget took approximately 14 hours to analyze all indirect branch targets, running on the i9-13900K Intel CPU with 20 cores. We count each unique transmission address as a gadget, so multiple paths leading to the same transmission count as 1 gadget.

Additionally, while InSpectre Gadget does not directly output information about *reachable* gadgets (i.e., those that can be user-triggered through a syscall), we estimate reachability for call targets by cross-referencing the labels with the coverage report generated by Syzkaller [13], a state-of-the-art kernel fuzzer. We use the openly-available results from the Syzbot project [12], which runs Syzkaller for 24 hours, finding a total of 14,391 reachable targets. This approach underapproximates the number of reachable gadgets, as the completeness is subject to fuzzing coverage, but it is useful to provide an estimate.

Table 1 summarizes the gadgets we uncovered. We found a total of 955 and 610 gadgets in kernel indirect call and jump targets (respectively). We manually analyzed 40 randomly sampled gadgets from the list by manually inspecting the assembly code (in approximately 3 hours). After manual analysis, we considered 35 to be exploitable. This shows InSpectre Gadget provides good accuracy, with the few misses caused by imprecise controllability modeling of our current prototype (Section 5.5).

Figure 5 shows statistics that we can use to estimate the size of the required speculation window, e.g., the number of instructions and branches present in the gadget, or how many dependent loads have to fit in the window to leak a secret.

Finally, Table 2 presents the number of potential gadgets that were *not* deemed exploitable by our tool. Gadgets with an invalid base are gadgets where the attacker does not completely control the transmission base, and in particular, for some values of the secret, the transmission address would be invalid, without the attacker being able to compensate by adjusting the transmission base. Gadgets classified as having an invalid secret address do not allow the attacker to choose an arbitrary memory location to leak, and gadgets where no

Table 1: The number of exploitable gadgets found by InSpectre Gadget in indirect call targets and indirect jump targets of the kernel, grouped by technique needed for exploitation.

Technique	*Call Targets*		*Reachable*		*Jump Targets*	
	Load	*Store*	*Load*	*Store*	*Load*	*Store*
None	14	2	1	1	0	0
Prime+Probe	199	126	64	51	126	17
Sliding	69	30	34	7	208	50
Known Prefix	399	121	107	19	135	110
Base Adjust	7	1	4	1	0	0
(Base Adjust + Known Prefix)	(79)	(30)	(28)	(9)	(88)	(43)
Train In-Place	467	210	174	68	153	52
Train OOP	53	10	25	2	130	77
Total	738	288	230	78	471	179

bits of the secret survive before being transmitted, e.g., if the secret is XOR-ed with itself, are classified as *Not Inferable*. Finally, the labels *Secret Too Big*, *Secret Too Small* and *Base Alias* refer to the problems mentioned in Section 5.2.

As mentioned earlier, InSpectre Gadget can be easily extended to support new covert channels. As a demonstration, we added support for both the recent SLAM covert channel [29] and the code-load (i.e., secret-dependent function pointer dereference) covert channel [45]. With our tool, we found ~4x more exploitable gadgets than SLAM's simple scanner, primarily due to our ability to reason about the exploitability of complex gadgets. The code-load covert channel further revealed over 2,000 SLAM gadgets, although no new traditional gadgets were found. For a more detailed analysis, we refer the reader to Appendix A.

5.5 Limitations

As opposed to tools like KASPER [31], InSpectre Gadget is designed to analyze only the content of *speculation windows* (e.g., call and jump targets for Spectre-v2), whose entry points have to be provided by the analyst. Other aspects needed for end-to-end exploitation, such as the *reachability* of the target and the presence of a suitable *victim branch*, are not part of the tool's output. Nonetheless, in Section 6.1, we show how to construct an end-to-end attack based on the results of our analysis. Moreover, InSpectre Gadget cannot completely prove the *absence* of gadgets in a given snippet of code.

Regarding exploitability results, our current prototype has a number of limitations potentially impacting accuracy. First, our tool is based on ANGR and relies on both its disassembler (CAPSTONE) and constraint solver (Z3). Whenever an error occurs in one of these components, e.g., on unsupported instructions, we have to bail out from the analysis, leaving some symbolic states unexplored. Second, transmissions that contain a complex symbolic expression, e.g., two independently-controlled loads used in a XOR operation, cannot be easily unpacked into a *base* and a *transmitted secret*, but might still leak a value. We mark these cases as *complex* and approximate their ranges by querying the SAT solver for the minimum and maximum values of the whole expression and of each sub-expression. We also use this approach when performing range analysis on expressions with complex (symbolic) constraints, whose values cannot be easily reduced to an interval or a small set. For these cases, besides reporting the minimum and the maximum values, we also check if certain bits are always 0 or 1 to approximate the stride.

Finally, at the moment our tool models the attacker's control over complex chains of loads as a binary condition (*controlled* or *not controlled*), as opposed to reporting the degree of control as done for transmission components. For complex aliasing cases, this can introduce imprecision (e.g., inaccurate classification of the required exploitation techniques).

Table 2: The number of potential gadgets marked as "not exploitable" by InSpectre Gadget, broken down by the reason for which they were deemed unexploitable.

Problem	*# of Gadgets*		*# Reachable*	
	Load	*Store*	*Load*	*Store*
Base Alias	1322	730	502	151
Invalid Base	32651	5619	9270	1391
Invalid Secret Address	412	78	104	11
CMOVE *Alias*	773	171	246	50
Secret Not Inferable	114	5	59	1
Invalid Transmission	268	56	51	9
Secret Too Big	2045	439	282	159
Secret Too Small	561	67	198	28
Total	34825	6035	9609	1530

6 Native BHI

In the previous section, we saw that InSpectre Gadget was able to uncover in the Linux kernel many gadgets that can lead to the transmission of a secret. In this section, we demonstrate how an attacker can use such gadgets to mount an end-to-end Spectre-v2 exploit against the kernel, by presenting the first native BHI attack (without the need of unprivileged eBPF).

6.1 Preliminaries

To perform native BHI, an attacker must first trigger the gadget via a syscall to insert its entry in the BTB. As discussed in Section 5.4, we use the reports from Syzkaller to determine which target can be triggered, and how. Note that, since we use valid indirect call targets, the attack is completely unaffected by the eIBRS and IBT mitigations.

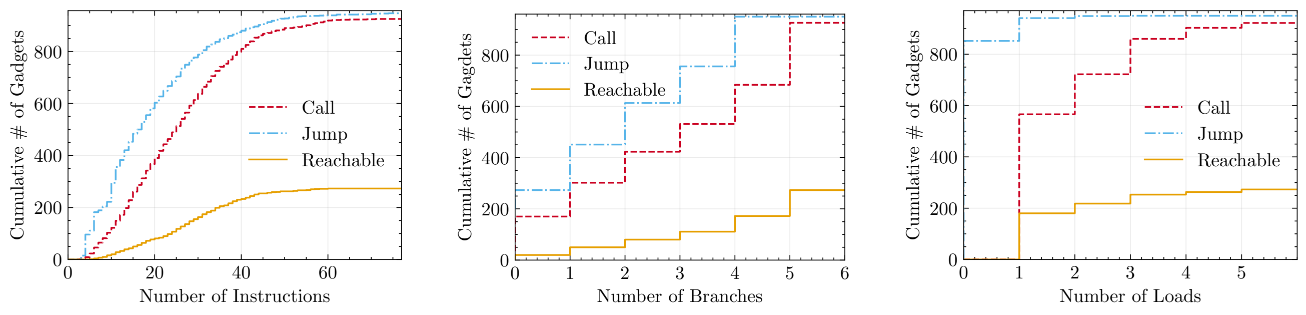


Figure 5: Cumulative distributions of the number of instructions, branches and dependent loads found in disclosure gadgets.

In addition, we must choose a victim indirect branch. Attackers need to ensure they control the registers and memory required by the gadget, when the victim call is triggered. To find potential victims, we use ANGR to search for indirect branches from the syscall entry point. Upon detecting an indirect branch, we perform range analysis on attacker-controlled registers. To find attacker-controlled values that require an extra dereference with a small offset, we first query the solver to determine if the expression stored in a register has a single solution. If so, we examine the memory at this address and the adjacent 256 bytes for attacker-controlled values and perform range analysis on them if found.

We identified 21 indirect branches across 11 syscalls, each with at least one register under sufficient attacker control to pass a kernel pointer.

6.2 End-To-End Exploit

In this section, we show how we craft an end-to-end exploit against the Linux kernel to leak the content of `/etc/shadow` on the latest Intel CPUs in under two minutes. Figure 6 shows a general overview of the attack.

Disclosure gadget and victim branch. As a first step, we search for a FLUSH+RELOAD gadget to provide an efficient covert channel, by filtering the database generated by InSpectre Gadget. In particular, we select `cgroup_seqfile_show`, shown in Listing 2, as our gadget. The corresponding annotated assembly file, generated by InSpectre Gadget, is included in Appendix C.

As our victim branch, we choose the kernel syscall handler. When executing this branch, all of the (attacker-controlled) syscall arguments have already been pushed on the stack, while `rdi` points to the location of these arguments. Since `cgroup_seqfile_show` loads a value from `rdi + 0x70` and uses it to compute all other values, attackers need to control just the first syscall argument when the misprediction occurs.

Inserting the BTB entry. To ensure that the CPU mispredicts to our gadget, its address must be injected into the BTB before calling the victim. To this end, we trigger the gadget from userspace exactly once. Manual analysis shows that we

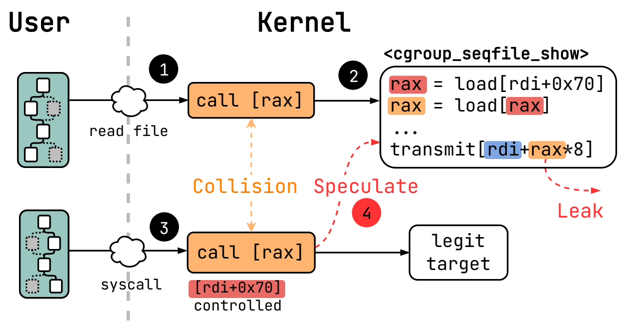


Figure 6: **Workflow for the native BHI exploit.** The attacker first triggers an indirect call in kernel ①, which jumps to `cgroup_seqfile_show` ②. Then, a colliding history is executed, and the syscall dispatcher is invoked ③, which expects the user-provided syscall argument at address `rdi+0x70`. Before executing the intended target, the CPU transiently jumps to `cgroup_seqfile_show` ④, which dereferences `rdi+0x70` and uses its value to construct a transmission.

can trigger `cgroup_seqfile_show` by reading a file in the cgroup directory (`/sys/fs/cgroup`) from userspace.

Increasing the transient window size. The induced transient window must be sufficiently large to fit our transmission. There are various ways to achieve this. In our case, we opted for evicting from the cache the syscall table entry that contains the legitimate target of our victim, making the victim call slow. Our experiments show that evicting the entry from the L2 data cache provides a window that is large enough.

Finding the reload buffer. To use FLUSH+RELOAD, there needs to be a shared buffer between the attacker and the victim. We can obtain such a reload buffer by allocating a user page and identifying its corresponding address in the kernel's direct map of physical memory. However, the map address is not known in advance, and must be leaked.

In principle, an attacker can brute-force this address by exploiting BHI to transiently jump to an attacker-controlled load, and check if the load brought the user page into the cache. However, two significant sources of entropy stand in

Listing 2: Assembly of the `cgroup_seqfile_show` gadget. Linux kernel 6.6-rc4, default configuration.

```
endbr64
push   rbp
mov    rax, QWORD PTR [rdi+0x70]; load user rdi
mov    r8,  rsi
mov    rbp, rdi
mov    rax, QWORD PTR [rax]     ; indirect load
mov    rsi, QWORD PTR [rax+0x60]; indirect load
mov    rdx, QWORD PTR [rax+0x8] ; indirect load
mov    rax, QWORD PTR [rsi+0x58]; load secret addr
mov    rdi, QWORD PTR [rdx+0x60]; load tbase
test   rax, rax
je     <cgroup_seqfile_show+55>
movsxd rax, DWORD PTR [rax+0x9c]      ; load secret
add    rax, 0x2e
mov    rdi, QWORD PTR [rdi+rax*8+0x8]; transmit
```

our way: the kernel direct map address (subject to KASLR and allocator entropy) and the victim/target branch histories (which need to collide). In other words, a cache hit for the user page can only be observed by the attacker if both the kernel direct map address of the user page is correctly guessed and the history correctly collides with the target branch.

To reduce the entropy and speed up brute forcing, we first find the start of the physical map and break KASLR. To do so, we use a modified version of the prefetch attack [26].

Next, instead of looking for a specific collision, as one would do when performing BHI, we perform a *parallel history collision* search by injecting the address of multiple gadgets in the BTB before calling the victim, and randomizing both the target and victim branch history.

Combining these optimizations allows us to identify the kernel direct map address for our user page within a minute on both Intel Comet Lake and Intel Raptor Lake CPUs.

Finding the histories. Once we have found the user page address, we follow the original BHI method to find a single reusable history for our victim branch colliding with the target branch [16]. Specifically, we randomize the victim history while keeping the target history constant until our user page gets loaded by the kernel.

Leaking kernel memory. To leak arbitrary kernel memory, we first have to find a known signature to use the known-prefix technique. It can be any value, including zeros. In our attack, to leak the shadow file, we first bring it into memory by calling `passwd -s`. Next, we start leaking from the start of the physical map and check if the signature 'root:' is present at the start of a 4k page [24].

Results. The end-to-end exploit time to leak the root password hash from the shadow file is on average 45s and 120s for the Intel i7-10700K and i9-13900K, respectively. The exploit runs longer on Raptor Lake due to the larger BHB size which therefore requires more collision iterations. The leakage rate after initialization is 4.5 KBps and 3.5 KBps on the i7-10700K and i9-13900K, respectively, with an accuracy of >99.9%

7 Dispatch Gadgets

A native BHI attacker is normally restricted to syscall-reachable disclosure gadgets. However, we found another type of gadget in *many* indirect kernel targets, i.e., one with an indirect branch to an attacker-controlled address (Figure 8)—which we refer to as *dispatch gadget* (or *dispatcher*).

To find such gadgets, we adapted InSpectre Gadget to also report such cases, which correspond to jumps to a symbolic address during symbolic execution. Most code logic for dispatch gadgets is shared with that of disclosure gadgets, as supporting dispatch gadgets primarily required adding an extra hook for symbolic branches and saving all the information from the analysis. With such support, we found as many as 2,039 dispatch gadgets residing within the first 6 basic blocks of an indirect call target of the Linux kernel, of which 477 are reported to be reachable by Syzkaller. We also found 462 others at the beginning of indirect *jump* targets. We manually analyzed 20 randomly sampled dispatch gadgets and found that all 20 are indeed exploitable. Figure 7 shows the depth of the dispatch gadgets found by InSpectre Gadget, measured in number of instructions, number of branches, and maximum number of dependent loads.

Dispatch types. Dispatch gadgets are particularly interesting because they allow an attacker to divert control flow to effectively *any* address. Use cases of interest are for example:

Dispatch-to-Call. The attacker jumps to a valid indirect call target. The target might either not be (easily) reachable via syscall, or the attacker uses the dispatcher to increase the number of controlled registers before jumping to the target.

Dispatch-to-Jump. Similar to Dispatch-to-Call, the attacker jumps to an indirect *jump* target. Jumps targets are of special interest as, with FineIBT, they are instrumented with an `endbr` instruction but not with the call-specific FineIBT instrumentation. We discuss this primitive in the next section.

Dispatch-to-Any. In case IBT is disabled, the attacker can jump to any code location, at direct call/jump targets, in the middle of functions, or even in the middle of instructions.

Dispatch-to-Dispatcher. The attacker chains two or more dispatchers to increase control over registers or to craft a disclosure gadget by chaining multiple load sequences together.

Chaining strategies. From an exploitation perspective, dispatch gadgets can be used to jump to a gadget within the same speculation window. With this *1-stage chaining* strategy, the dispatcher can easily reach unreachable code, increase controlled registers, etc. For example, if at the time of misprediction only `rax` is controlled, an attacker might jump to a dispatcher that moves `rax` to `rbx` before jumping to the final target. This allows the attacker to use any gadget that requires control of `rbx`—not previously possible. To demonstrate this approach, we repurposed our exploit to use `common_timer_delete` as the dispatcher to jump to the `of_css` disclosure gadget—a valid indirect call target not normally reachable from an unprivileged attacker's workload.

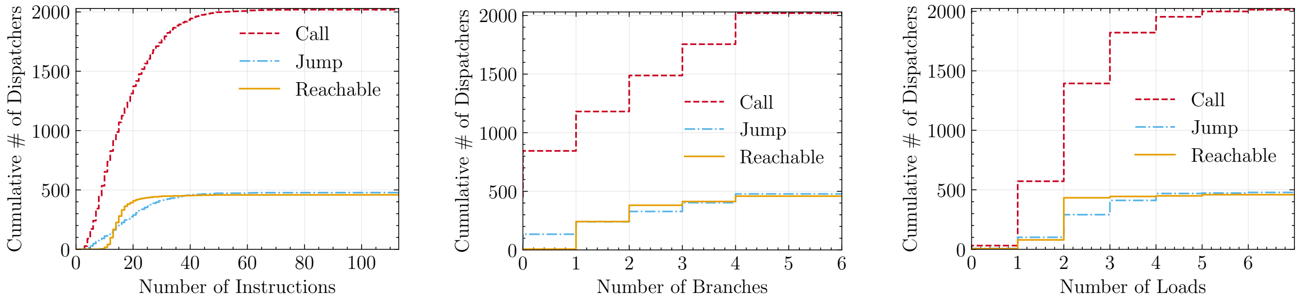


Figure 7: Cumulative distribution of number of instructions, branches, and dependent loads of the attack surface for dispatch gadgets found within indirect call (*Call*), indirect jump (*Jump*), and reachable indirect call (*Reachable*) targets.

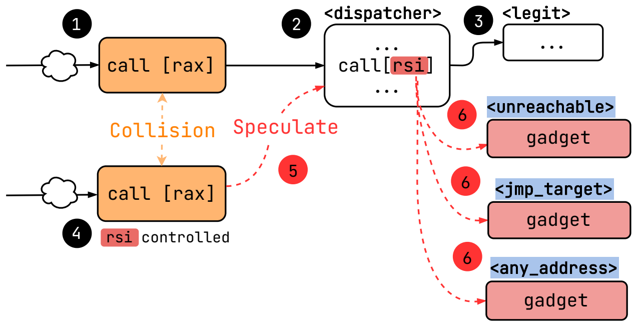


Figure 8: **Dispatch gadgets.** Instead of transiently jumping to a disclosure gadget, an attacker can jump to a gadget that calls a controlled function pointer. From that point, the attacker can divert control flow to less restrictive gadgets.

Listing 3: IBT test snippet.

```
ind_target:
    .rept 16
        mov rax, QWORD PTR [rax]
    .endr
```

We achieved a leakage rate of 3.3 KB/sec on the i9-13900K.

Alternatively, the attacker could opt for a *2-stage chaining* strategy , i.e., using the dispatcher to inject a controlled BTB entry within the transient window [51] and later collide with another victim. This strategy offers two benefits. First, dispatch and disclosure gadgets are executed in separate speculation windows, accommodating a larger number of instructions. Second, this strategy allows the attacker to exploit different victims with different controllability characteristics for dispatcher and disclosure gadgets. To demonstrate this approach, we repurposed our exploit to use `m_show` as the dispatcher to inject the address of the `of_css` disclosure gadget into the BTB and, next, start a new history colliding phase to collide with the BTB entry just inserted. We achieved a leakage rate of 2.9 KB/sec on the i9-13900K.

8 FineIBT Analysis

Having shown that our discovered dispatch/disclosure gadgets offer many options to mount cross-privilege Spectre-v2 attacks against the kernel, defeating deployed mitigations that are currently believed to hinder exploitation, we now evaluate the impact of the recent FineIBT mitigation on these attacks. We first analyze the effective speculation window(s) and the impact of Simultaneous Multithreading (SMT) resource contention, before discussing the residual attack surface.

8.1 IBT Speculation Window

Since FineIBT builds on IBT, any IBT-induced speculation window is of concern. For Intel CPUs supporting IBT, the CET-tracker transitions to the `WAIT_FOR_ENDBRANCH` state upon execution of an indirect branch. If the subsequent instruction is not an `endbr` instruction, speculation is limited. To be more precise, Intel specifies that IBT limits speculative execution to 7 instructions (with a maximum of 5 loads) in early implementations, while later versions should completely block speculative execution after a missing `endbr` [30].

We evaluate the limits of IBT in the context of speculative execution by opening a transient window in a kernel module and executing an indirect call to a target snippet which omits the `endbr` instruction (Listing 3).

The snippet executes a chain of dependent loads, which ends with a load from a reload buffer. Prior to each iteration, we flush the reload buffer and, after executing the snippet, we test if the reload buffer entry is cached. We ran the experiment over 100 separate runs, with each run performing 1 million iterations for different sizes of the load chain. The test snippet is positioned between two pages filled with zeros, which prevents interference from other `endbr` instructions. We performed our experiment on the three most recent Intel CPU generations: Rocket Lake, Alder Lake, and Raptor Lake. Table 3 presents our results.

We observed that the Intel i7-11800H CPU features an early implementation of IBT, which allows for up to 5 dependent loads in the speculation window. We measured an

Listing 4: FineIBT instrumentation.

```
__cfi_ind_target:
    endbr64
    sub r10d, 0x8baaa714
    je <ind_target>
    ud2
    nop
ind_target:
```

average hit rate of 34% for the last load. For every NOP instruction inserted before the load chain, we observed one less load being speculatively executed. Finally, we tested whether the alignment of the caller or target impacts the speculation window. The caller's alignment impacts the hit rate, and in the least optimal alignment, only 3 dependent loads can be fitted in the window before speculation is killed by IBT.

For the i9-12900K and i9-13900K CPUs, we can execute exactly 1 load in the IBT window. Adding a NOP instruction results in the reload buffer no longer being cached. We observed no influence from the alignment of the caller or target.

8.2 FineIBT Speculation Window

As a fine-grained extension of IBT, FineIBT introduces a speculation window of its own. This is due to the (SID check) conditional branch part of the software instrumentation. Nonetheless, such branch has been explicitly designed as "low-latency", performing a comparison between a register value and an immediate operand. As a result, FineIBT is currently considered an effective countermeasure against speculative control-flow hijacking attacks [1, 23].

Listing 4 shows the FineIBT instrumentation as it appears in the Linux kernel. The caller of an indirect branch is required to insert the correct SID into the `r10` register and call the CFI variant of the target function. Inside the callee, the correct SID is subtracted from `r10` and, if the outcome is zero, the instrumentation jumps to the actual call target. Conversely, if the SID is incorrect, the UD2 operation is executed, resulting in an invalid opcode exception.

To evaluate the speculation window size, we instrument the test snippet from Listing 3 with FineIBT. Next, we train the branch predictor by transiently executing the snippet with the correct SID, and later perform a call with an incorrect SID. To ensure that the same PHT entry is used for both the training phase and the malicious call, we prime the branch history by executing 200 branches after we differentiate between a training and a malicious call, which is in line with previous results [56]. Our experiments showed that this successfully trains the branch predictor on the three selected tested CPUs for this experiment. Table 3 presents our results.

As shown in the table, on all the tested CPUs, we can complete 1 load from RAM into memory with a high hit rate on average, from 79% up to 85%. This contrasts with the findings

Listing 5: FineIBT contention snippet.

```
1 .align 32
2 do_contention:
3     .rept REPEAT
4         sub eax, 0x1
5         .rept N_JE
6             je do_contention
7         .endr
8     .endr
9     jne do_contention
```

of the original FineIBT paper [23], which observed a considerably lower hit rate (17 hits out of 10 million iterations) when testing for Spectre resilience. However, we observe similar results without prior PHT entry massaging. As such, we hypothesize that the branch predictor may have distinguished between test and training runs in their case, thereby correctly predicting the SID check for the test runs. Namely, their test setup, despite conducting 10 training runs, jumps to a location dependent on whether it is a training or test run just prior to executing the FineIBT check. This jump difference can likely be inferred by the branch predictor.

8.3 Impact of SMT Contention

When SMT is enabled, a number of resources are shared between two sibling logical processors sharing the same core. Prior research [39] has shown that resource contention from a sibling processor can cause nontrivial delays in the other processor, thereby expanding the speculation window sizes.

To assess the impact of SMT contention on the (Fine)IBT speculation windows, we use the same test setup as before. However, we now introduce a contention workload on the sibling core. We seek to induce a delay in the FineIBT check (i.e., subtract operation, conditional branch evaluation, speculation rollback) or more generally in the victim logical processor (accounting also for the IBT window).

We evaluated the effects of SMT contention using different workloads and observed that the behavior of each workload strongly varies across different runs, although it remains stable during the run. We focus primarily on finding the most efficient workload on average, which is shown in Listing 5.

Branch contention. To delay the FineIBT check, the main ingredient is conditional branch contention. Changing the branch type in the branch sequence (line 6) to a direct jump/call or indirect jump/call to the next instruction does not yield a higher hit rate compared to no SMT workload. Our experiments also showed that nontaken conditional branches are more effective on average.

Arithmetic operations. We tested workloads with `sub`, `add`, `mul`, and `cmp` operations. A workload with only arithmetic operations does not seem to increase the hit rate. We also experimented with adding arithmetic operations in combination with conditional branches. Our experiments showed

Table 3: Hit rates for different load chain sizes while racing against the IBT and FineIBT mitigations.

Intel Core	*SMT*	*IBT Hit Rates*					*FineIBT Hit Rates*									
		1 LD	*2 LD*	*3 LD*	*4 LD*	*5 LD*	*1 LD*	*2 LD*	*3 LD*	*4 LD*	*5 LD*	*6 LD*	*7 LD*	*8 LD*	*9 LD*	*10 LD*
i7-11800H	None	100%	100%	100%	59%	57%	82%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Contention	99%	99%	74%	49%	11%	100%	96%	55%	14%	<1%	0%	0%	0%	0%	0%
*i9-12900K**	None	100%	0%	0%	0%	0%	79%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Contention	100%	0%	0%	0%	0%	100%	95%	65%	39%	20%	8%	1%	0%	0%	0%
*i9-13900K**	None	100%	0%	0%	0%	0%	85%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Contention	100%	0%	0%	0%	0%	100%	79%	57%	33%	16%	7%	4%	<1%	<1%	<1%

* Performance core

Table 4: The SMT workload configuration per CPU model

Intel Core	*N_REPEAT*	*N_JE*
i7-11800H	2	5
i9-12900K	1	9
i9-13900K	2	8

that placing one `sub` or `add` instruction before the branch sequence yields a higher hit rate.

Alignment. We observed an alignment of 32 bytes for the contention workload to be generally the most effective. However, for some test runs, we observed the highest hit rates on different, unpredictable alignments. We suspect this is due to side effects from branch aliasing.

Workload size. During the experiments, we observed that the most effective configuration of the workload for each tested CPU differs in the number of sequential conditional branches (`N_JE`). Additionally, repeating the snippet before jumping back to the entry point can also affect the effectiveness (`N_REPEAT`). We experimentally derived the best configuration for the tested CPUs (Table 4).

We repeated the (Fine)IBT speculation window experiments with our fine-tuned SMT workload. Table 3 reports our results. As shown in the table, for the IBT window, our SMT workload does not increase the window size. For the FineIBT window, on the other hand, our SMT workload heavily impacts the number of loads one can fit across microarchitectures. Specifically, we observed cache hits up to 5, 7, and 10 loads for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. Noteworthy, on the i9-13900K CPU, a contention workload composed solely of taken branches showed varied results: 80% of the runs had hits only up to 2 loads, 15% had a 90% hit rate for 6 loads, and 4% even reached a 90% hit rate for 9 loads.

9 FineIBT Bypass

We showed that the (Fine)IBT speculation windows still leave room to transiently fit a gadget with illegal control flow. In particular, depending on the microarchitecture, our results show that a FineIBT attacker can transiently execute 4-6 dependent loads with a hit rate of at least 7%. As such, attackers can use a variety of techniques to bypass FineIBT. To find gadgets for these techniques, the analyst can issue a SQL query to filter the InSpectre Gadget database for gadgets that satisfy the corresponding requirements.

Racing against the FineIBT window. An attacker may race against the FineIBT window with a disclosure gadget, and, optionally, use Dispatch-to-Call to increase reachability of disclosure gadgets (either via 1-stage or 2-stage chaining). As shown in Figures 5 and 7, the ability to fit 4 loads in the transient window is sufficient to have a wide selection of gadgets. More specifically, if we conservatively filter for reachable dispatchers with a maximum of 3 dependent loads within 10 instructions, we are left with 44 gadgets. If we filter for disclosure gadgets with at most 15 instructions and 4 dependent loads, we obtain 88 reachable gadgets and 548 non-reachable gadgets, which can be reached through a dispatcher.

Racing against the IBT window. Instead of using indirect call targets as disclosure gadgets, an attacker can also opt to use Dispatch-to-Any to reach any executable code for disclosure. However, the attacker then needs to race against both the FineIBT and the (nested) IBT window.

On CPUs with an early IBT implementation, an attacker can use a dispatcher to jump to a disclosure gadget containing at most 5 loads. With 1-stage chaining, the dispatch gadget must fit in the FineIBT window, while the transmission races against both the FineIBT and IBT window. With 2-stage chaining, the FineIBT window is avoided by inserting the transmission gadget's address in the BTB.

Later IBT implementations allow for exactly one speculative load after the illegal jump. While at a first glance this may seem unhelpful, attackers can still exploit any dispatch gadget that loads an attacker-controlled value and use the additional load as a transmission. InSpectre Gadget found 115 dispatch gadgets with at least one load preceding (but not controlling) the dispatch call.

Using JMP targets. IBT requires both indirect call targets and indirect jump targets to land on an `endbr` instruction. In contrast, FineIBT only guards indirect call targets. As a result, an attacker can target indirect *jump* targets—which do start

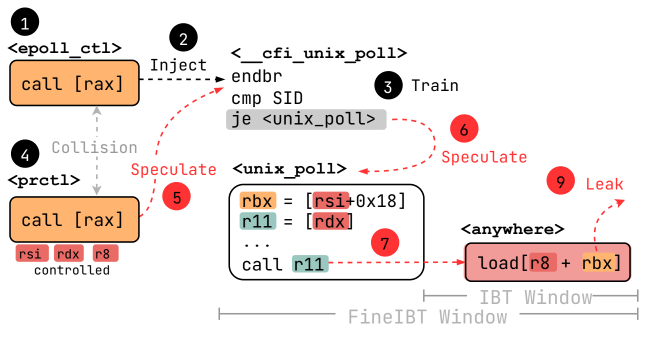


Figure 9: **FineIBT bypass case study.** The attacker invokes `epoll_ctl` ① to inject the address of `unix_poll` ② and train its FineIBT check ③. Then, `prctl` is invoked ④, which speculatively jumps first to the target's FineIBT check ⑤ then to its body ⑥. Inside `unix_poll`, a secret is loaded, then the function jumps to a single, final load ⑦ which discloses the secret ⑧.

with an `endbr` instruction but are not instrumented with a SID check—to bypass FineIBT. Hence, the attacker can use a disclosure gadget reached via indirect jump without racing against the FineIBT window. InSpectre Gadget found 462 dispatch gadgets and 610 viable disclosure gadgets at indirect jump targets in the Linux kernel.

Case study. As a case study, we demonstrate how a BHI attacker can bypass FineIBT and leak kernel memory in practice on a 13th Gen Intel CPU. At a high level, as depicted in Figure 9, we rely on a Dispatch-to-Any primitive with 1-stage chaining. This is to divert control flow to a dispatcher gadget (valid according to IBT, invalid according to FineIBT) and then, in turn, to an arbitrary final target (invalid according to IBT, unchecked by FineIBT). To this end, we need to race against two nested speculation windows (other than the initial BHI one): (i) the FineIBT window checking for the invalid (SID-violating) control-flow transfer to the dispatcher gadget; (ii) the IBT window checking for the invalid (`endbr`-bypassing) control-flow transfer to the final target.

From our InSpectre Gadget results, we need to first select a dispatch gadget that can race against the main FineIBT window. To this end, we select `unix_poll` (Appendix B), a short-lived dispatcher that loads an attacker-controlled value into a first register before a call via a second, attacker-controlled register. Next, we need to select a final target that can race against the nested IBT window. Recall from Section 8.2 that, on 13th Gen Intel IBT, we can still execute exactly one instruction without a preceding `endbr`. As a result, we need to select as final target a load instruction disclosing the secret in the first register. To this end, we scanned the Linux kernel image for loads with matching register. We found over 900 suitable instructions and selected one as our final target.

To inject the address of the dispatch gadget into the BTB, we invoke the `epoll_ctl` syscall. As our victim branch, we selected an indirect call in the `prctl` syscall which, at call time, grants the attacker full control over `rsi` and `rdx`, required by the `unix_poll` gadget. Finally, to successfully race against the main FineIBT window, we need to correctly train the SID branch (Section 8.2). To this end, one can in principle just execute `epoll_ctl` when inserting the BTB entry. However, this is alone insufficient due to the different path histories of the training and test runs (potentially distinguishable by the branch predictor). As a result, upon the misprediction of the SID check during the test run, a new entry is inserted into the level-1 PHT, indexed by the test run's path history [56].

To evict the test run's PHT entry, we build a jump chain of conditional branches. Eviction of the level-1 PHT requires us to initially take all branches, resulting in an entry insertion in the local base predictor. Then, we need to cause a misprediction by not taking the branches, leading to an entry inserted into the level-1 PHT and potentially evicting the entry of the test run. For this purpose, we used a simple (but far from optimal) eviction strategy walking 8k branches. Therefore, we alternate between eviction sets during the colliding phase and, after we find a collision, we randomize eviction sets until we observe a hit rate exceeding our lower bound.

This strategy, in conjunction with our fine-tuned SMT workload (Section 8.3), allowed us to achieve a hit rate of between 50% and 70% and a leakage rate of 18 B/sec on the i9-13900K CPU. It should be noted that an attacker can certainly optimize our PHT eviction strategy to gain a higher leakage rate.

10 Mitigations

To mitigate the gadgets found by InSpectre Gadget, an option is to add a speculation barrier (i.e., `LFENCE` instruction) at the function entry point of all the disclosure and dispatch gadgets. However, possible kernel performance degradation aside, this strategy cannot guarantee the absence of residual exploitable gadgets—especially with an ever-evolving Linux kernel. A more general strategy is to hinder BHI exploitation, e.g., with `BHI_DIS_S` controls [1] (which is only supported from Alder Lake and Sapphire Rapids CPUs onwards) or a software BHB-clearing sequence [1] (which is, however, costly). This strategy is still insufficient to stop other (intra-mode BTI) variants. To fully close the attack surface, we need new in-silicon mitigations (e.g., decoupling history and IP matching logic) or more costly ones, such as retpoline [1] (which is, however, vulnerable on some microarchitectures [54]) or `IPRED_DIS` controls [1] (which are, again, only supported from Alder Lake and Sapphire Rapids CPUs onwards).

Vendors' response. In response to our disclosure, Intel acknowledged our findings and updated their BHI mitigation guidance [1]. AMD and ARM confirmed that their existing mitigations are sufficient. Given the significant presence of exploitable gadgets revealed by our findings, Intel now recommends software vendors to enable BHI_DIS_S on CPUs that support it and to insert a software BHB clearing sequence at privilege boundaries on other CPUs. For future CPUs that enumerate BHI_NO, no additional mitigations are required. On the Linux kernel, Intel engineers have developed patches to incorporate the recommended mitigations [9] and kernel developers have decided to replace the syscall handler's indirect call with a switch statement [3]. While this mitigates Spectre-v2 on this specific call site, many other easily reachable indirect branches remain (e.g., the indirect call used in our FineIBT Bypass PoC). Moreover, since conditional direct calls are vulnerable to Spectre-v1, an attacker might still be able to confuse syscall targets [15]. Finally, our endbr analysis uncovered a bug in Clang (causing the compiler to occasionally emit spurious endbr instructions), which Intel engineers promptly patched.

11 Related Work

Spectre gadget scanners. Spectre gadget scanners documented in literature mostly focus on Spectre v1. With exceptions [43], such scanners typically rely on dynamic analysis. SpecFuzz [40] uses fuzzing to detect out-of-bounds accesses on a speculative path. SpecFuzz marks every out-of-bound access as a gadget, without modeling attacker controllability. In response, SpecTaint [42] requires the secret address to be tainted with attacker input, as determined via dynamic taint analysis (DTA). Kasper [31] also relies on DTA for gadget characterization, but generalizes the fixed patterns used by SpecTaint. Unlike InSpectre Gadget, all these solutions identify gadgets solely based on their data flow, an overapproximation that leaves their exploitability uncertain. Like ours, other gadget scanners are based on symbolic execution, but typically focus on other use cases (i.e., verification [28] or early detection [53]) with no exploitability analysis.

Spectre-v2 attacks. Besides work exclusively focusing on gadget scanning, prior Spectre-v2 attack efforts also described gadget analysis campaigns. For instance, the Retbleed authors [54] used static data-flow analysis to identify basic 3-load gadgets for a native Spectre-RSB-to-BTI exploit. This simple strategy is sufficient as Retbleed exploits (similar to Inception [51]) vulnerable older-generation CPUs with no eIBRS. As such, they can speculatively hijack control flow to arbitrary code locations with no restriction.

The BHI authors [16], also using data-flow-based gadget analysis, presented evidence eBPF=off exploits were at least potentially feasible on modern eIBRS-enabled platforms—but with no attempt to reason about exploitability. Intel later presented a similar gadget analysis campaign along with manual exploitability analysis of "the most promising" gadgets, reporting no exploitable ones [8]. This shows the difficulty of performing exploitability analysis—even from experts *manually* inspecting the gadgets—without a general framework to rule out self-limiting assumptions.

In contrast, by analyzing one (v2) transient window at the time, InSpectre Gadget can leverage the full power of symbolic execution to deeply characterize gadgets and reason about their exploitability for the first time, without running into the limitations of traditional symbolic execution tools [20, 21], such as scalability and state explosion. As a result, our analysis was not only able to automatically uncover several exploitable native gadgets, but even gadgets that can bypass newer mitigations (FineIBT).

Crucially, our tool models knowledge of advanced exploitation techniques including sliding and gadget chaining. Chaining itself has been proposed before to increase the Spectre-v2 attack surface [17, 51]. With InSpectre Gadget, we demonstrate for the first time its crucial role in cross-privilege Spectre-v2 attacks, countering mitigations with widespread dispatch gadgets in modern kernels.

12 Conclusion

In this paper, we showed that, by relaxing the requirements on "standard" exploitable Spectre gadgets and using in-depth gadget inspection, it is possible to generically reason about gadget exploitability. To substantiate this claim, we presented InSpectre Gadget and applied its exploitation-aware gadget analysis to uncover a significant residual attack surface for cross-privilege Spectre-v2 attacks against the Linux kernel. Specifically, we revealed several new gadgets and showed that they can be exploited by a BHI attacker not only to leak kernel memory in native end-to-end exploits, but also bypass all deployed mitigations including the recent Fine(IBT).

13 Disclosure

We disclosed the end-to-end native BHI exploit to Intel in May 2023, and disclosed our full analysis to Intel, AMD, ARM and the Linux kernel in October 2023, which further notified other vendors. We converged to a public disclosure date of April 9, 2024, providing time for vendors to roll out mitigations. A number of vendors (Intel, Microsoft, Google, Xen) have also explicitly requested access to InSpectre Gadget for internal attack surface analysis, which we granted under embargo. Native BHI has been assigned CVE-2024-2201.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback, Enrico Barberis for his support on the Native BHI exploit, and Andrew Cooper for his early feedback and contributions to the tool. This work was supported by Intel Corporation through the "Allocamelus" project, by NWO through project "INTERSECT" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] BHI and intra-mode BTI. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[2] Branch target injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html.

[3] Don't force use of indirect calls for system calls. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e3ad78334a69b36e107232e337f9d693dcc9df2.

[4] Enable kernel IBT by default. https://lore.kernel.org/lkml/166764458451.4906.10224019690835731804.tip-bot2@tip-bot2/T/.

[5] Implement FineIBT. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=931ab63664f0.

[6] 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.

[7] Indirect branch tracking for Intel CPUs. https://lwn.net/Articles/889475.

[8] Intel research on disclosure gadgets at indirect branch targets in the Linux kernel. https://www.intel.com/content/www/us/en/developer/articles/news/update-to-research-on-disclosure-gadgets-in-linux.html.

[9] Merge tag 'nativebhi'. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2bb69f5fc72183e1c62547d900f560d0e9334925.

[10] Retpoline: A branch target injection mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[11] Spectre side channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html#spectre-variant-2-branch-target-injection.

[12] Syzbot. https://syzkaller.appspot.com/.

[13] Syzkaller. https://github.com/google/syzkaller.

[14] Vulnerability of speculative processors. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[15] Nadav Amit, Fred Jacobs, and Michael Wei. Jump-Switches: Restoring the performance of indirect branches in the era of Spectre. In *USENIX ATC*, 2019.

[16] 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.

[17] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. SpecROP: Speculative exploitation of ROP chains. In *RAID*, 2020.

[18] 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.

[19] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *ASIACCS*, 2011.

[20] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *PLDI*, 2020.

[21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter-efficient relational symbolic execution for Spectre with haunted relse. In *NDSS*, 2021.

[22] 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.

[23] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P Kemerlis. FineIBT: Fine-grain control-flow enforcement with indirect branch tracking. *ACM 2023*, 2023.

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

[25] Ben Gras, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida, et al. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[27] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack, 2016.

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

[29] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. Leaky address masking: Exploiting unmasked Spectre gadgets with noncanonical address translation. In *IEEE S&P*, 2024.

[30] Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes. 2019.

[31] 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.

[32] 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 *IEEE S&P*, 2019.

[33] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.

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

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: reading kernel memory from user space. In *USENIX Security*, 2018.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, 2015.

[37] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *USENIX Security*, 2021.

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

[39] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You cannot always win the race: Analyzing mitigations for branch target prediction attacks. In *IEEE EuroS&P*, 2023.

[40] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security*, 2020.

[41] Colin Percival. Cache missing for fun and profit. 2005.

[42] 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.

[43] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: Exploiting and mitigating speculative race conditions. In *USENIX Security*, 2024.

[44] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking ARM pointer authentication with speculative execution. In *ISCA*, 2022.

[45] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead µops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, 2021.

[46] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[47] Dag Arne Osvik Shamir, Adi and Eran Tromer. Cache attacks and countermeasures: the case of AES, 2005.

[48] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *S&P*, 2019.

[49] 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 *ACM CCS*, 2018.

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

[51] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Security*, 2023.

[52] 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 *IEEE S&P*, 2019.

[53] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESPECTRE: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM TOSEM*, 2020.

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

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

[56] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution. In *IEEE S&P*, 2023.

A Extra Covert Channels

To demonstrate the flexibility of InSpectre Gadget, we extended the tool to include two extra covert channels, namely the code-load covert channel [45], which requires a secret-dependent function pointer, and the recent SLAM covert channel [29]. SLAM leverages Intel's recently announced Linear Address Masking (LAM) feature, as well as microarchitectural race conditions present on some existing AMD CPUs. Via SLAM, an attacker can leak data via a straightforward dereference of a 64-bit secret, typically a noncanonical address, by transiently bypassing canonicality checks.

Supporting SLAM required modifications only in the reasoner, with 83 line changes. The code-load covert channel required even fewer modifications, with only 7 line changes in the scanner. We ran the extended implementation of InSpectre Gadget with the same setup described in Section 5.4. Table 5 presents our results. We count each entry point as a single gadget, in line with SLAM's definition.

The SLAM paper reports a large potential attack surface when scanning the Linux kernel's indirect call targets (16,046 *potential* gadgets), but practical exploitability is approximated by pattern-matching simple cases, resulting in 4,194 exploitable gadgets. In contrast, by reasoning on complex gadgets as well, InSpectre Gadget is able to uncover 13,648 SLAM gadgets that pass all exploitability tests in indirect call targets, and a total of 16,287 exploitable SLAM gadgets when considering also indirect jump targets and code-loads. While we did not uncover any new traditional gadgets via the code-load covert channel, we identified 2,856 SLAM gadgets that are exploitable via the code-load covert channel.

B FineIBT Case Study Gadget

Listing 6 presents the assembly code of the unix_poll_gadget, the gadget used in the FineIBT bypass case study. The gadget loads a secret from memory with attacker-controlled register `rsi` as address. Next, the gadget loads the call target from the attacker-controlled address in `rdx` and calls it subsequently. The secret is transmitted by the instruction at the call target, selected by the attacker, that performs a load with the secret as an argument and an attacker-controlled value as the base. We selected the instruction `movzx ebx, BYTE PTR[r8+rbx]` found in the `uuid_string` function.

Table 5: The number of exploitable SLAM gadgets found by InSpectre Gadget in indirect call targets and indirect jump targets of the kernel, grouped by technique needed for exploitation. Counted by the number of indirect entry points with at least one gadget.

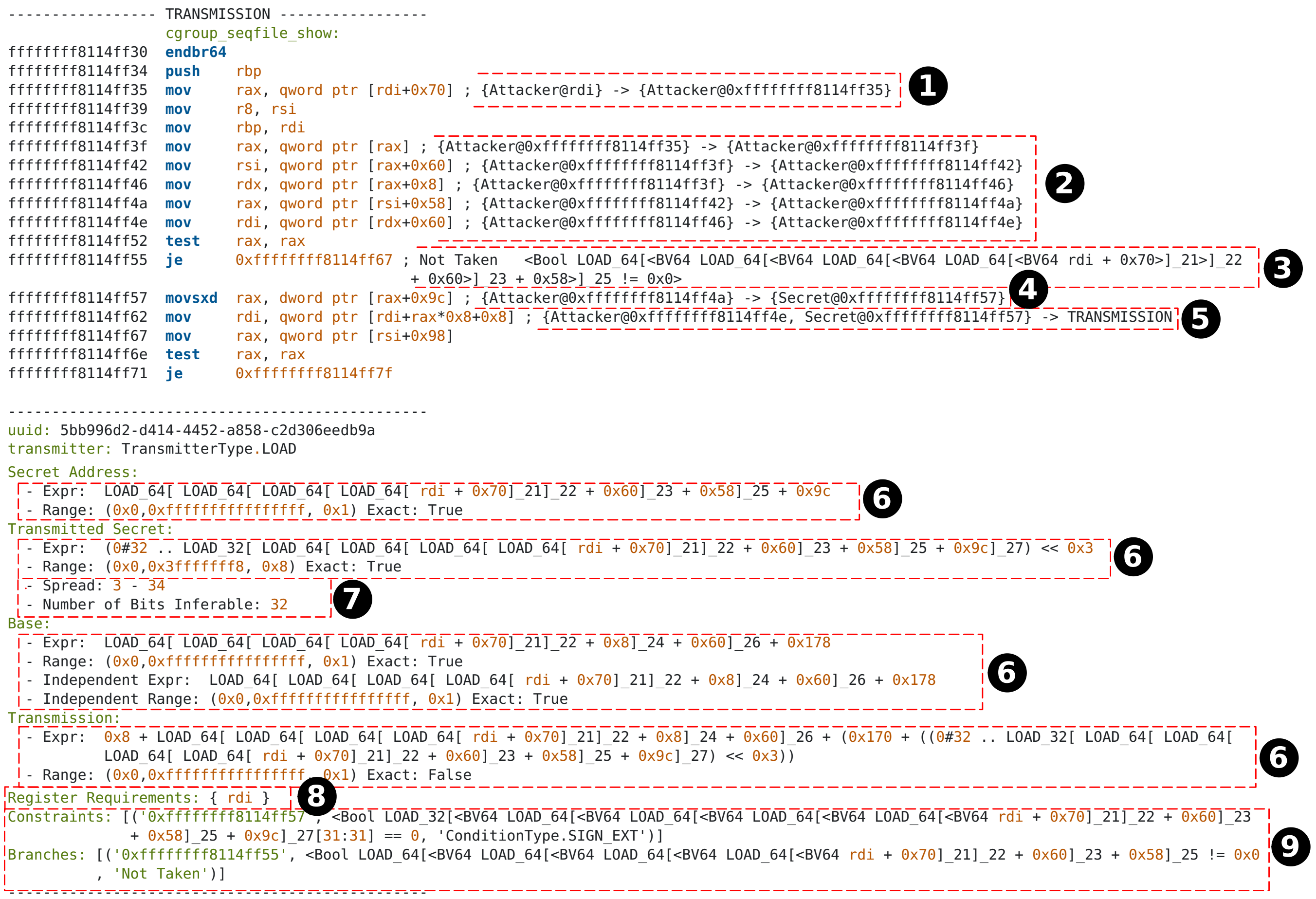
Technique	***Call Targets***			***Jump Targets***		
	Load	***Store***	***Code-load***	***Load***	***Store***	***Code-load***
Known Prefix	14,840	3,470	2,440	1,981	831	474
Train In-Place	5,285	1,796	1,230	531	308	49
Train OOP	363	174	75	1,132	430	424
Total	14,847	3,485	2,440	1,987	832	474

Listing 6: Assembly of the `unix_poll` gadget. Linux kernel 6.6-rc4, FineIBT enabled.

```
__cfi_unix_poll:
  endbr64
  sub    r10d,0x1eb58ddc
  je     <unix_poll>
  ud2
  nop
unix_poll:
  nop    WORD PTR [rax]
  push   rbp
  push   r14
  push   rbx
  mov    rbx, QWORD PTR [rsi+0x18]; load secret
  test   rdx, rdx
  je     <unix_poll+55>
  mov    r11, QWORD PTR [rdx]; load call target
  test   r11, r11
  je     <unix_poll+55>
  add    rsi, 0x40
  mov    r10d, 0xd0facb91
  sub    r11, 0x10
  nop    DWORD PTR [rax+0x0]
  call   r11; call attacker chosen target
```

C Annotated Assembly Output

Figure 10 shows the annotated assembly output of the (`cgroup_seqfile_show`) gadget used in our exploit.



```
---------------- TRANSMISSION ----------------
                  cgroup_seqfile_show:
ffffffff8114ff30  endbr64
ffffffff8114ff34  push    rbp
ffffffff8114ff35  mov     rax, qword ptr [rdi+0x70] ; {Attacker@rdi} -> {Attacker@0xffffffff8114ff35}
ffffffff8114ff39  mov     r8, rsi
ffffffff8114ff3c  mov     rbp, rdi
ffffffff8114ff3f  mov     rax, qword ptr [rax] ; {Attacker@0xffffffff8114ff35} -> {Attacker@0xffffffff8114ff3f}
ffffffff8114ff42  mov     rsi, qword ptr [rax+0x60] ; {Attacker@0xffffffff8114ff3f} -> {Attacker@0xffffffff8114ff42}
ffffffff8114ff46  mov     rdx, qword ptr [rax+0x8] ; {Attacker@0xffffffff8114ff3f} -> {Attacker@0xffffffff8114ff46}
ffffffff8114ff4a  mov     rax, qword ptr [rsi+0x58] ; {Attacker@0xffffffff8114ff42} -> {Attacker@0xffffffff8114ff4a}
ffffffff8114ff4e  mov     rdi, qword ptr [rdx+0x60] ; {Attacker@0xffffffff8114ff46} -> {Attacker@0xffffffff8114ff4e}
ffffffff8114ff52  test    rax, rax
ffffffff8114ff55  je      0xffffffff8114ff67 ; Not Taken   <Bool LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 rdi + 0x70>]_21>]_22
                                               + 0x60>]_23 + 0x58>]_25 != 0x0>
ffffffff8114ff57  movsxd  rax, dword ptr [rax+0x9c] ; {Attacker@0xffffffff8114ff4a} -> {Secret@0xffffffff8114ff57}
ffffffff8114ff62  mov     rdi, qword ptr [rdi+rax*0x8+0x8] ; {Attacker@0xffffffff8114ff4e, Secret@0xffffffff8114ff57} -> TRANSMISSION
ffffffff8114ff67  mov     rax, qword ptr [rsi+0x98]
ffffffff8114ff6e  test    rax, rax
ffffffff8114ff71  je      0xffffffff8114ff7f

----------------------------------------------
uuid: 5bb996d2-d414-4452-a858-c2d306eedb9a
transmitter: TransmitterType.LOAD
Secret Address:
 - Expr:  LOAD_64[ LOAD_64[ LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x60]_23 + 0x58]_25 + 0x9c
 - Range: (0x0,0xffffffffffffffff, 0x1) Exact: True
Transmitted Secret:
 - Expr:  (0#32 .. LOAD_32[ LOAD_64[ LOAD_64[ LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x60]_23 + 0x58]_25 + 0x9c]_27) << 0x3
 - Range: (0x0,0x3fffffff8, 0x8) Exact: True
 - Spread: 3 - 34
 - Number of Bits Inferable: 32
Base:
 - Expr:  LOAD_64[ LOAD_64[ LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x8]_24 + 0x60]_26 + 0x178
 - Range: (0x0,0xffffffffffffffff, 0x1) Exact: True
 - Independent Expr:  LOAD_64[ LOAD_64[ LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x8]_24 + 0x60]_26 + 0x178
 - Independent Range: (0x0,0xffffffffffffffff, 0x1) Exact: True
Transmission:
 - Expr:  0x8 + LOAD_64[ LOAD_64[ LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x8]_24 + 0x60]_26 + (0x170 + ((0#32 .. LOAD_32[ LOAD_64[ LOAD_64[
          LOAD_64[ LOAD_64[ rdi + 0x70]_21]_22 + 0x60]_23 + 0x58]_25 + 0x9c]_27) << 0x3))
 - Range: (0x0,0xffffffffffffffff, 0x1) Exact: False
Register Requirements: { rdi }
Constraints: [('0xffffffff8114ff57', <Bool LOAD_32[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 rdi + 0x70]_21]_22 + 0x60]_23
               + 0x58]_25 + 0x9c]_27[31:31] == 0, 'ConditionType.SIGN_EXT')]
Branches: [('0xffffffff8114ff55', <Bool LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 LOAD_64[<BV64 rdi + 0x70]_21]_22 + 0x60]_23 + 0x58]_25 != 0x0
          , 'Not Taken')]
----------------------------------------------
```

Figure 10: **Annotated assembly file generated by InSpectre Gadget.** Right to the assembly instructions, we output the annotations attached to the source and destination operands (source →destination). The annotations include the origin of the value, which is either the instruction pointer of the source load or a register (e.g., `@rdi`). As we generate for each detected gadget an annotated assembly file, we do not print annotations that are irrelevant to the gadget flow and we replace all secret annotations with an attacker annotation that are, for this specific gadget, not used as a secret but as an attacker-controlled value. In the case of a branch instruction, we show the branch condition to hold instead.
As shown by the annotations, the attacker-controlled value `rdi` is used in the first load ①, followed by a series of loads whose controllability is tracked ②. The encountered branch condition is recorded ③. Subsequently, the secret value is loaded from an attacker-controlled value ④ and transmitted using an attacker-controlled value as the base ⑤. We output key details after the assembly code, including the symbolic expression and the range—i.e., (min, max, stride)—for each transmission component ⑥, insights about the transmitted secret bits ⑦, details about which registers an attacker should control to exploit the gadget ⑧ as well as the constraints and branches encountered ⑨.

USENIX Security '24 Artifact Appendix: InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2

Sander Wiebing* Alvise de Faveri Tron* Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

Our paper presents InSpectre Gadget, an in-depth Spectre gadget analyzer that can be used to identify viable disclosure gadgets on large codebases. We used such analyzer to uncover the residual attack surface for cross-privilege Spectre v2 in the Linux Kernel. We demonstrated the significance of such an attack surface by showing the first end-to-end Spectre v2 attack against the Linux kernel that does not require eBPF to craft gadgets (Native BHI). We also showed how the abundance of both disclosure and dispatch gadgets can be used to bypass all modern mitigations, including FineIBT. Finally, we measured the size of speculation windows for both IBT and FineIBT on a variety of microarchitectures, and the effects of SMT contention on such windows. This artifact contains all the code needed to reproduce the analysis of the Linux Kernel, as well as the PoCs for both Native BHI and the FineIBT bypass and the experiments on (Fine)IBT speculation windows.

A.2 Description & Requirements

There are four main artifacts that are relevant for this evaluation: 1) Linux Kernel analysis; 2) Native BHI; 3) Speculation-window experiments; 4) FineIBT bypass. Different requirements may apply for the different artifacts.

A.2.1 Security, privacy, and ethical concerns

Running the Linux Kernel analysis does not pose any risk, and the PoCs for both Native BHI and the FineIBT bypass are designed to only leak data locally.

A.2.2 How to access

All of our artifacts are available at the following url https://github.com/vusec/inspectre-gadget/releases/tag/v1.1, under the `experiments/` folder.

A.2.3 Hardware dependencies

There are no hard requirements for running Linux kernel analysis, however, the amount of available memory can have a minor impact on the number of reported gadgets due to execution paths being killed prematurely. For the evaluation reported in the paper, we used a 13th Gen Intel i9-13900K with 32 cores and 64GB of RAM.

For the other artifacts, the following CPUs are required for evaluation.

- **CPU: 13th Gen Intel(R) Core(TM) i9-13900K.** Required for Native-BHI, FineIBT bypass, and speculation-window experiments.
- **CPU: 12th Gen Intel(R) Core(TM) i9-12900K.** Required for speculation-window experiments.
- **CPU: 11th Gen Intel(R) Core(TM) i7-11800H.** Required for speculation-window experiments.

A.2.4 Software dependencies

The tool itself requires `python3`, and the test scripts are written in `bash`. The only requirement for the Linux Kernel analysis is to have `docker` installed. The analysis was tested on Ubuntu 23.04.

For Native-BHI, Fine-IBT Bypass and FineIBT Speculation Window the following requirements apply:

- Linux kernel 6.6.0-rc4, custom build. Source code available here: https://github.com/torvalds/linux/archive/refs/tags/v6.6-rc4.tar.gz.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To access the artifacts, first clone https://github.com/vusec/inspectre-gadget and checkout to the `v1.1` tag.

```
git clone \
git@github.com:vusec/inspectre-gadget.git

cd inspectre-gadget
git checkout v1.1
```

To run the scanner locally, you will need to install the following:

```
sudo apt install python3 python3-pip bat
pip3 install -r requirements.txt
```

If `bat` is not available, you can download it from https://github.com/sharkdp/bat.

Finally, you can navigate to the `experiments/` folder. Here you will find a folder for each of the artifacts to evaluate.

Linux Kernel analysis. All scripts can be found under the `scanner-eval/` folder. The only requirement is to have `docker` installed on the system.

Native BHI. All scripts are under the `native-bhi/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../native-bhi/kernel
   ./build_kernel.sh
   ```

Speculation window experiments. All scripts are under the `speculation-windows/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../speculation-windows/kernel
   ./build_kernel.sh
   ```

   Note: If your architecture is not supported by the Ubuntu config, create a config via `make localmodconfig` after you extract the Linux source code. Although we did not test it, the tests should not be dependent on a kernel configuration.

3. Please isolate two performance sibling cores by adding `isolcpus=` to the kernel boot paramters (.e.g, `isolcpus=2, 3`). Adjust the core numbers in the file `src/targets.h`. Please reboot the system and verify if the cores are isolated:

   ```
   # Print the isolated cores
   cat /sys/devices/system/cpu/isolated
   ```

FineIBT Bypass. All scripts are under the `fineibt-bypass/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel with the tested configuration. The required PoC patch will be applied. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../fineibt-bypass/kernel
   ./build_kernel.sh
   ```

A.3.2 Basic Test

The `tests/test-cases` folder contains a set of simple cases to verify how the scanner behaves in various situations (cmove, branches ecc.). A single test can be ran for example as:

```
cd tests/test-cases
make
./run-single.sh used_memory_avoider
```

This should result in the scanner reporting two potential transmissions, both at address `0x4000014`. The two transmissions come from the same expression, but a different part of the expression is considered secret each time.

It is recommended to install `batcat` to visualize the resulting annotated assembly with `batcat output/asm/*`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We analyzed the Linux kernel version 6.6-rc4 (latest at time of writing) with the default configuration. We found a total of 922 and 589 exploitable gadgets in kernel indirect call and jump targets (respectively). Section 5.4 of the paper reports results of our analysis (specifically, Table 1 and Table 2). Moreover, the cumulative distributions of the gadgets found by the scanner are shown in Figure 5 and Figure 7 of the paper. Experiment (E1) reproduces the analysis on the Linux kernel and reports all the relevant numbers.

(C2): We demonstrated the Native BHI PoC can leak arbitrary kernel memory at 3.5 kB/sec on the i9-13900K CPU. This is proven by experiment (E2) which tests the leakage rate and leaks the shadow file from memory.

(C3): We show that a speculation window is present at the IBT and FineIBT defense mechanisms. We claim that we can fit up to 5, 1 and 1 dependent loads in the IBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. We claim that we can fit up to 5, 7 and 10 dependent loads in the FineIBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. This is proven by experiment (E3).

(C4): We demonstrated that the FineIBT BHI PoC can leak kernel memory at 18 B/sec on the i9-13900K CPU with a FineIBT-enabled kernel. This is proven by experiment (E4) which tests the leakage rate.

A.4.2 Experiments

(E1): [Linux Kernel analysis] [10 compute-hours + 15GB disk]: This experiment builds an image of the Linux kernel version 6.6-rc4, extracts all the call and jump targets, then runs the scanner on the kernel image and extracts a set of possible gadgets, which are then saved in a database.
Preparation: Navigate to `experiments/scanner-eval`.
Execution: Run `run.sh`. This will create a docker container, install dependencies, and automatically run the analysis (in particular, `scripts/run-eval.sh`).
Results: The results of the analysis are available in the `results` subfolder after the run. In particular, `stats.txt` contains the numbers used for tables and throughout the evaluation section, while the `figs/` subfolder contains the cumulative distribution plots. `gadgets.db` contains the database of all the gadgets, which is queried through the queries contained in `analysis/queries`.

(E2): [Native BHI] [10 compute-minutes + 15GB disk]: This experiment runs the Native BHI PoC on the custom-build kernel.
Preparation: Navigate to `native-bhi/src`. Test the working of the PoC by first skipping the huge-page finding phase:

```
sudo ./run.sh -p
```

Execution: Test the leakage rate:

```
sudo ./run.sh test_rate
```

Test and time shadow leak:

```
time sudo ./run.sh leak_shadow
```

Results: The results (leakage rate and shadow leak) are printed to the terminal.

(E3): [Speculation window experiments] [8 compute-hours + 15GB disk]: This experiment runs the speculation window experiments on the custom-build kernel.
Preparation: Navigate to `speculation-windows`. Install the kernel module:

```
cd kernel_modules/
    ibt_tests_kernel_module
sudo ./run.sh
```

Execution: Run the experiment. Replace the CPU-NAME with the CPU name (consult `lscpu`).

```
cd src
sudo ./run_test.sh CPU-NAME
```

Results: The results are available in the folder `src/results`. To analyze the results:

```
./analyze_all.sh src/results/
```

(E4): [FineIBT Bypass] [10 compute-minutes + 15GB disk]: This experiment runs the FineIBT Bypass PoC on the custom-build kernel.
Preparation: Navigate to `fineibt-bypass/src`.
Execution: Test the leakage rate:

```
sudo ./run_fast.sh
```

To test the full PoC, including the collision finding phase. Note that the collision-finding phase can take up to 5 minutes

```
sudo ./run.sh
```

Results: The leakage rate is printed to the terminal.

A.5 Notes on Reusability

While the tool has been used specifically to target Spectre-v2 in the Linux Kernel, its structure is general enough to be applicable also to other targets. Full documentation and examples of how to use the tool can be found in `docs/index.html` or at https://vusec.github.io/inspectre-gadget/.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.