

# Branch Target Reuse: Practical Spectre-v2 Attacks in JIT Engines via Stale Branch Prediction Entries

Sander Wiebing\*

Vrije Universiteit

Amsterdam

Amsterdam, Netherlands

[s.j.wiebing@vu.nl](mailto:s.j.wiebing@vu.nl)

Yuhui Zhu\*

Scuola Superiore Sant'Anna

Scuola IMT Alti Studi Lucca

Pisa & Lucca, Italy

[y.zhu@sssup.it](mailto:y.zhu@sssup.it)

Alessandro Biondi

Scuola Superiore Sant'Anna

Pisa, Italy

[al.biondi@sssup.it](mailto:al.biondi@sssup.it)

Cristiano Giuffrida

Vrije Universiteit

Amsterdam

Amsterdam, Netherlands

[giuffrida@cs.vu.nl](mailto:giuffrida@cs.vu.nl)

## Abstract

Self-Modifying Code (SMC) is the foundation of dynamic code generation in commodity JIT engines. Modern processors provide coherence mechanisms that ensure stale, overwritten code cannot be executed architecturally, but their guarantees during *microarchitectural* execution remain less understood. The only previously known SMC-based transient execution primitive (SCSB), is limited to x86 and relies on a very short instruction/data-cache coherence window, shaping the common view that SMC-based transient execution attacks are impractical in real-world JIT engines.

In this paper, we overturn this assumption and introduce *Branch Target Reuse (BTR)*, the first practical *in-place* Spectre-v2 attack. The key insight is that, while modern CPUs restore architectural code coherence after self-modification, they do not necessarily invalidate stale indirect branch prediction entries (i.e., *branch targets*). In JIT engines, these stale targets can outlive the original code and later be *reused* when the code cache is repopulated, yielding a transient *execute-after-free* primitive. This allows attackers to hijack transient control flow to newly generated code at obsolete offsets, bypassing software hardening or reaching misaligned gadgets. We evaluate BTR across SpiderMonkey, GraalVM, and the Linux kernel's cBPF JIT, and show that it is a cross-engine issue whose exploitability depends on code-cache management and garbage-collection noise. Focusing on the most vulnerable target, cBPF, we build two end-to-end exploits against the Linux kernel that leak the root password hash in minutes on Intel systems despite deployed mitigations and even when enabling cBPF's constant blinding defense.

## CCS Concepts

• Security and privacy → Operating systems security; Browser security; Side-channel analysis and countermeasures.

## Keywords

Microarchitectural security, Transient execution attacks, Spectre-v2 attacks, JIT engine exploitation, Linux kernel exploitation

## ACM Reference Format:

Sander Wiebing, Yuhui Zhu, Alessandro Biondi, and Cristiano Giuffrida. 2026. Branch Target Reuse: Practical Spectre-v2 Attacks in JIT Engines via Stale Branch Prediction Entries. In *Proceedings of the 2026 ACM SIGSAC*

\*Equal contribution joint first authors.



This work is licensed under a Creative Commons Attribution 4.0 International License. CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11

<https://doi.org/10.1145/3830454.3846672>

Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846672>

## 1 Introduction

Spectre-v2 attacks exploit indirect branch prediction to transiently hijack control flow and disclose sensitive information [37]. In classic injection-based variants [6, 37, 62], the attacker poisons the branch predictor so that a victim branch will speculate to a chosen target. In practice, speculating to exploitable targets typically requires *out-of-place* attacks, where speculation is steered toward a target injected via another branch. This model, however, is increasingly constrained by modern defenses, which restrict the set of speculation targets through stronger isolation (e.g., eBRS) and control-flow enforcement (e.g., FineIBT), ideally confining the attacker to *in-place* Spectre-v2 attacks—where speculation is constrained to the architectural target set of the victim branch. This strategy rests on the widely held assumption that in-place Spectre-v2 attacks are impractical [63]. In this paper, we challenge this assumption and present *Branch Target Reuse (BTR)*, the first practical in-place Spectre-v2 attack. BTR targets JIT engines and arises from the interplay between Self-Modifying Code (SMC) and indirect branch prediction. To demonstrate its practicality, we present two end-to-end exploits against the Linux kernel's cBPF JIT that recover the root password hash in minutes.

**SMC.** JIT engines routinely emit machine code into writable buffers, make it executable, and later reclaim and reuse code-cache pages as functions or traces are recompiled or freed. From the processor's perspective, these transitions are instances of self-modifying code (SMC): bytes first created or modified via the data path are later consumed again by the instruction-fetch path. Modern architectures therefore provide mechanisms to restore coherence after such updates. On x86, dedicated SMC handling automatically re-synchronizes the L1 instruction/data caches with the updated bytes. On ARM, software must issue the appropriate cache-maintenance barriers to obtain the same effect. These provisions are fundamental to JIT execution because they ensure that architectural execution always observes the rewritten code rather than stale instructions lingering in the cache hierarchy.

**Abusing SMC for transient execution attacks.** Prior work has shown that SMC can also induce transient execution. In particular, *Speculative Code Store Bypass (SCSB)* demonstrated that x86 SMC handling, implemented via microcode assists, creates a transient execution window in which stale instructions may briefly execute after code is rewritten but before the processor completes recovery and restarts from a safe point [49]. However, SCSB concluded that this window is too narrow for practical transient execution attacks

against real-world JIT engines. As a result, the prevailing intuition has been that, although SMC can momentarily desynchronize architectural and microarchitectural views of code, it does not ultimately yield a practical transient execution primitive.

**Branch Target Reuse.** In this paper, we introduce *Branch Target Reuse (BTR)* and show that JIT engines *do* expose exploitable transient-execution opportunities induced by SMC for the first time. Our key insight is that, while the instruction cache is repaired after self-modification, other microarchitectural structures may retain stale state. Specifically, we show that the indirect branch predictor typically preserves stale entries (i.e., stale branch targets) when dynamically generated code is rewritten or freed. As a result, stale targets can outlive the original code, and even when the corresponding code-cache region is later reused with different content, these stale entries may still be used transiently. By carefully masaging the JIT code cache, BTR turns this behavior into a transient *execute-after-free* primitive, which we then escalate into a *control-flow hijacking* one: transient execution jumps to an address that was valid in the old code but now lands inside the new code at a different offset. Because the bytes at that address now decode differently, the transient path can execute misaligned gadgets or skip Spectre hardening instructions that would have run along the intended entry path.

We analyze BTR across SpiderMonkey, GraalVM, and the Linux kernel’s cBPF JIT, and find that all three engines are affected, albeit with markedly different exploitability characteristics and leakage rates. A key determinant is garbage collection: code reclamation and sweeping introduce timing and placement noise, whereas more deterministic code-cache reuse makes BTR substantially easier to exploit. Among the engines we studied, cBPF is the highest-impact target, has the most favorable reuse behavior, and, upon disclosure, developers were reluctant to deploy mitigations without hard proof of exploitability (Appendix B). Hence, we focus on cBPF for end-to-end exploitation. To this end, we build two end-to-end in-place Spectre-v2 exploits against the Linux kernel that recover the root password hash within minutes on Intel systems.

The first exploit targets default mitigations on stock Ubuntu and executes a misaligned microarchitectural gadget. The second exploit targets a hardened cBPF configuration, demonstrating it is possible to circumvent cBPF’s constant blinding defense. We further analyze the impact of other Spectre-v2 countermeasures on our exploits. We find that lightweight countermeasures such as FineIBT are insufficient in isolation, but, when combined with constant blinding, they offer strong, though not future-proof, protection. Heavyweight countermeasures such as IBPB and (selective) retpoline are effective, but also impose higher complexity/performance costs.

**Contributions.** We make the following contributions:

- (1) We present *Branch Target Reuse (BTR)*, the first practical *in-place* Spectre-v2 attack. BTR targets JIT engines and breaks prior assumptions of deployed mitigations.
- (2) We evaluate the practicality of BTR across three real-world JIT engines: SpiderMonkey, GraalVM, and cBPF.
- (3) We build two end-to-end exploits against the Linux kernel that bypass all the mitigations present in the default Ubuntu configuration on recent Intel systems, and, additionally, even bypass the (non-default) cBPF’s constant blinding hardening.

- (4) We analyze other relevant Spectre-v2 countermeasures and derive practical recommendations to defend against BTR.

## 2 Background

### 2.1 Branch prediction

Branch prediction is a core optimization in modern CPUs to hide memory latency. Before a branch is resolved, the Branch Prediction Unit (BPU) chooses the next instruction to speculatively execute. The BPU typically uses a Branch Target Buffer (BTB) to predict branch target addresses. The BTB may be indexed by the instruction pointer, or combined with the branch history (e.g., a Branch History Buffer, BHB) to predict multi-target indirect branches. When the CPU sees a branch, it probes the BTB and begins executing the predicted target. If the prediction is wrong, the processor discards the *transient* execution and resumes from the correct path, but microarchitectural effects such as cache state persist.

### 2.2 Spectre v2

Spectre v2 exploits indirect branch prediction. An attacker trains the predictor so that a victim indirect branch speculatively jumps to a chosen target. The goal is to reach *disclosure gadgets* and leak sensitive data into microarchitectural state, which can be recovered through covert channels such as FLUSH+RELOAD [69].

Early Spectre-v2 defenses blocked indirect speculation (e.g., ret-polines [28, 56]) or isolated BTB entries across privilege boundaries (Intel eIBRS [26], ARM CSV2 [3]). Later, Branch History Injection (BHI) [6] showed that BTB isolation alone is insufficient, because the branch history also affects predictions and can be manipulated from user space. Modern systems therefore isolate the branch history as well. For example, on Intel, older platforms clear the history in software; newer Intel CPUs disable history-based prediction via BHI\_DIS\_S, and Lion Cove CPUs add BHI\_NO to allow only *privileged* history-based prediction [63].

Spectre v2 can be launched out-of-place or in-place [6]. Out-of-place attacks use separate training and victim branches, while in-place attacks use the same branch. Prior work on Spectre v2 [6, 62, 63, 65] focused primarily on out-of-place training, since targeting a disclosure gadget from a single branch is generally harder. Similarly, most prior Spectre-v2 attacks rely on cross-domain training, since self-training attacks are more challenging [63].

### 2.3 JIT engines

Just-in-time (JIT) compilers generate machine code at runtime for browsers, managed runtimes, and kernel subsystems. Examples include JavaScript and WebAssembly engines, GraalVM, and the Linux BPF JIT. As part of their optimized operations, JIT engines frequently allocate, modify, and free attacker-controlled code, which induces self-modifying-code behavior but also raises the stakes for transient-execution defenses.

**SpiderMonkey.** Mozilla Firefox’s SpiderMonkey JIT engine [12] uses index masking or speculation barriers to prevent speculative out-of-bounds accesses [11]. It also reduces timer precision to hinder timing side channels [60], which, however, remain even with coarse timers [19]. Finally, Firefox uses site isolation to separate different origins into different processes on Desktop [14], while,

on mobile, isolation is incomplete—only “high-value” sites are isolated [57]. Generally, it is difficult to implement perfect site isolation and prevent all cross-origin attack opportunities [2].

**GraalVM.** GraalVM [38] is a multi-language runtime that combines ahead-of-time and dynamic (JIT) compilation. For untrusted code, it offers sandboxing levels [40], where the strictest mode limits system access and defends against both JIT spraying and transient execution attacks. To mitigate JIT spraying, it uses constant blinding and randomized entry-point offsets. To mitigate transient execution attacks, it employs the Monocle SFI technique to mask memory accesses within a sandbox [39, 48].

**BPF.** The Linux kernel JIT for BPF compiles programs at load time. Classic BPF (cBPF) is available to unprivileged users and is widely used for Linux Socket Filtering or LSF (e.g., by Snort) [32] as well as Seccomp [33] (e.g., by Docker). In contrast, eBPF is more powerful but restricted to privileged users. cBPF is intentionally simple: it supports only two 32-bit registers, forward jumps, and no register dereferences [30]. Because of these limits, it does not currently require extra Spectre JIT mitigations.

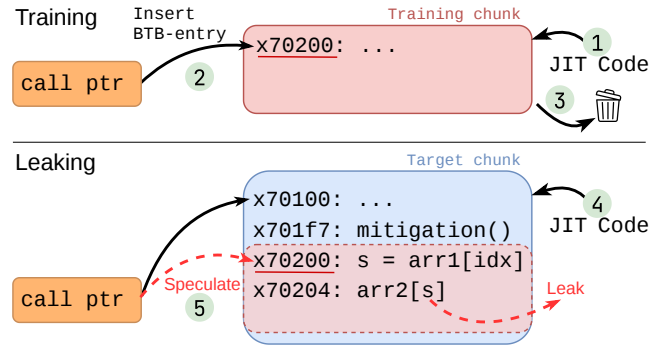
### 3 Threat Model

We consider a traditional in-domain Spectre-v2 threat model with an attacker able to run unprivileged code in a JIT engine and seeking to disclose sensitive data from the host environment. For end-to-end exploitation, we specifically focus on the Linux kernel’s cBPF JIT engine. This target translates to a cross-privilege Spectre exploitation scenario against the Linux kernel, with an unprivileged attacker able to execute arbitrary native code in user space, install arbitrary cBPF filters (e.g., via SECCOMP), repeatedly trigger kernel entry points (i.e., system calls) to execute those filters, and observe the resulting microarchitectural state changes through a covert channel to leak kernel memory. We consider a Linux kernel running on recent Intel systems with deployed mitigations enabled. Unless stated otherwise, this corresponds to the default mitigation configuration of a stock (Ubuntu) distribution (with default-on unprivileged cBPF); for our hardened setting, we additionally consider cBPF constant blinding enabled. We do not assume any architectural software vulnerability, such as memory corruption or privileged code execution. In particular, unlike prior in-kernel Spectre-v2 exploits that relied on eBPF gadget injection, we assume no access to (default-off) unprivileged eBPF.

### 4 Overview

Figure 1 presents a high-level overview of the Branch Target Reuse (BTR) attack. To exploit BTR, an attacker must target an indirect branch whose referenced code may later be rewritten. If this *victim* branch speculatively jumps to the original target after the code has been modified, the CPU transiently executes instructions from an architecturally invalid entry point.

To exploit this microarchitectural behavior, the attacker first lures the JIT engine into allocating a *training chunk* and forces the victim branch to jump to it, thereby inserting a BTB entry referencing the current entry point. Next, the attacker forces the JIT engine to free the training chunk and then allocate a *target chunk* that overlaps with the original entry point. Upon the next execution of the victim branch, if the CPU uses the stale BTB entry to



**Figure 1: Branch Target Reuse Attack.** The attacker lures the JIT engine into allocating a new code section ①, the *training chunk*, and trains the indirect branch by jumping to it ②. Next, the attacker forces a deallocation of the training chunk ③ and an allocation of the *target chunk*, reusing (partially) the same address ④. By triggering the indirect branch again, the CPU will use the now stale BTB entry and speculate to the old training-chunk entry-point ⑤, resulting in control-flow hijacking and secret data disclosure.

predict the target, transient execution triggers *execute-after-free* behavior, enabling *control-flow hijacking*. By redirecting control flow to an architecturally invalid entry point, the attacker can bypass Spectre hardening mitigations or execute misaligned instructions, ultimately disclosing secret data.

To exploit BTR in practice, several challenges must be overcome. First, the stale BTB entry must not be invalidated or replaced after the JIT engine frees the training chunk. Moreover, in order to execute misaligned instructions, the branch predictor must select the stale BTB entry for prediction. Therefore, we need to study how to reliably force the branch predictor to use stale BTB entries.

**Challenge 1:** How can the branch predictor be induced to use stale BTB entries?

Second, the attacker must force the JIT engine to free the training chunk and reallocate the target chunk at an overlapping address. To induce such *memory reuse* pattern, the attacker must implement *memory massaging* to manipulate the allocator’s behavior, which can introduce noise and potentially evict the injected BTB entry. Therefore, we need to study techniques to increase the durability of stale BTB entries in the face of system activity.

**Challenge 2:** How can stale BTB entries survive system activity?

Beyond the aforementioned microarchitectural challenges, it remains unclear whether BTR primitives can be crafted in real-world engines to construct an end-to-end attack. Specifically, the attack requires a victim indirect branch that can target dynamically generated code and is vulnerable to Spectre v2. The attacker must be also able to insert training chunks, free them, and reinsert target chunks repeatedly. Finally, the resulting transient control-flow hijack must

**Table 1: Details for the tested microarchitectures.**

| ISA     | Vendor   | Model          | $\mu$ Arch               |
|---------|----------|----------------|--------------------------|
| x86_64  | Intel    | i9-14900K      | Raptor Cove <sup>†</sup> |
|         |          | Ultra 9 285K   | Lion Cove <sup>†</sup>   |
|         | AMD      | Ryzen 9 7950X  | Zen 4                    |
| AArch64 | Broadcom | BCM2712 (Pi 5) | Cortex-A76               |
|         | Google   | Tensor G3      | Cortex-X3 <sup>†</sup>   |

<sup>†</sup> Performance core

be able to trigger a gadget that enables secret leakage. Therefore, we need to investigate whether real-world JIT engines meet these requirements, and how to craft the attack in practice.

**Challenge 3:** How can BTR primitives be used to leak secret data in real-world JIT engines?

In the following sections, we first address the microarchitectural challenges and characterize the behavior and persistence of stale BTB entries. We then assess the feasibility of our attack on three real-world JIT engines and illustrate its practical application. Finally, we demonstrate two end-to-end BTR exploits against the Linux kernel's cBPF JIT engine on recent Intel systems, even with deployed mitigations and constant blinding enabled.

## 5 Microarchitectural Primitives

In this section, we study the required microarchitectural primitives. All experiments are performed on two Intel CPUs, two ARM CPUs, and one AMD CPU. Table 1 details the evaluated platforms. While we have not performed experiments on RISC-V, we anticipate similar stale-entry behavior to the one detailed in our experiments below—e.g., as confirmed on the BOOM RISC-V core by inspecting its source code [7].

### 5.1 Using stale entries

We first study whether the branch predictor may use stale branch target entries, i.e., BTB entries whose target code has been overwritten. For this experiment, we create a single indirect branch with two targets: a target  $T$  that loads the first function argument, and a dummy (no-op) target. We train the indirect branch to jump to  $T$  and then execute the branch with the dummy target. To detect a misprediction to  $T$ , we use a classic FLUSH+RELOAD covert channel [37] on the buffer passed as the first argument. To extend the speculation window, we flush the memory location containing the dummy target before executing the branch. As expected, the results show that all the tested CPUs misspeculate to target  $T$ .

Next, we overwrite  $T$  between the training and test phases with a new function that loads the second argument instead. On ARM, we additionally execute the required barriers and invalidate the instruction cache after the overwrite. At the end of the test phase, we read performance counters to confirm the presence of the SMC event. As expected, performance counters also consistently indicate that jumping to the overwritten instruction cache line leads to a miss due to SMC. However, we observe no other relevant misses,

indicating no SMC-triggered invalidations of related microarchitectural components such as the iTLB—which Intel documents as a key component in SMC detection [24]. To confirm this intuition, we use again the FLUSH+RELOAD covert channel to still observe misprediction to the original target on all tested CPUs. This shows that, unlike instruction cache lines but similar to other related microarchitectural components, BTB entries are not invalidated when their target code is rewritten, and thus can be reused as stale entries. We also confirmed the same behavior for return prediction and (stale) RSB entries.

### 5.2 Amplifying the signal

We next study whether multiple stale BTB entries can be injected and reused for the same indirect branch. Amplifying the number of entries is beneficial since (i) it increases the probability that entries survive reuse noise, and (ii) it enables multiple mispredictions per training phase, improving the overall leakage rate.

We extend our previous experiment by executing the indirect branch under different branch histories. By controlling the history, we aim to insert multiple distinct BTB entries for the same target. To identify configurations that maximize the number of injected entries, we vary both the number of histories, the history length, and the number of taken branches.

Lastly, during our experiments, we observed that BTB insertion policies can limit the number of entries. In particular, as also evidenced in prior work, predictors may fall back to a default target if many histories map to the same target [41, 63]. To mitigate this effect, we additionally evaluate three training strategies: single-target training, mock-target training, and alternate-target training. For single-target training, we use a single training target across all histories. For mock-target training, after each single-training iteration, we re-execute the indirect branch with a fixed history and a mock target (i.e., a target touching no reload buffer), forcing a two-target behavior while still only inserting one training target. Finally, for alternate-target training, we alternate between two different training targets.

Figure 2 presents our results. We grouped the results by the count of inserted histories and training method, presenting only the optimal settings for history length and the number of taken branches. The number of successfully reused entries differs markedly across microarchitectures: for instance, Golden Cove yields approximately 500 entries, while Zen 4 and Cortex-X3 yield over 4,000 entries. Generally, the alternating-training method is the best performer, although for Lion Cove, the method choice appears irrelevant.

### 5.3 BTB-entry survival

Finally, we examine the longevity of stale BTB entries and the impact of system activity on their persistence. We replicate our previous experiment, employing the configuration that yields the maximum number of injected entries for each microarchitecture and using the alternating training method.

Between the training and reuse phases, we assess various scenarios. Initially, we incorporate a busy empty loop to investigate if stale entries become invalidated over time, with and without overwriting the target code. Additionally, we mimic system activity

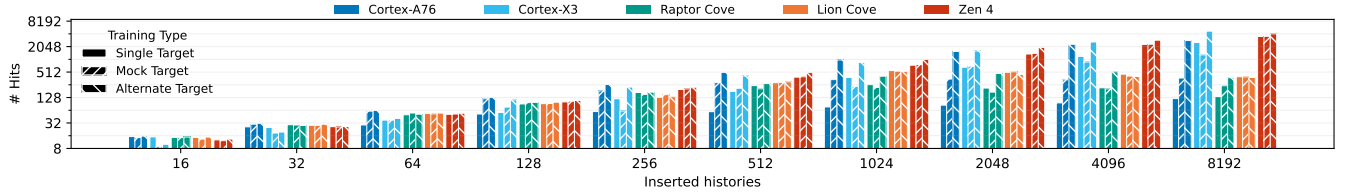


Figure 2: Number of unique BTB entries inserted and successfully reused for different machines across different training types.

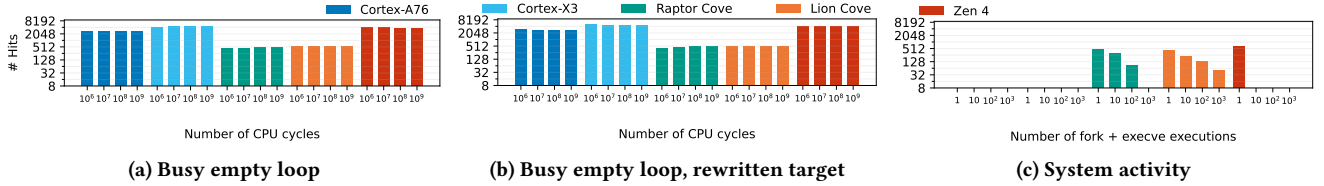


Figure 3: BTB-entry survival after different periods and activity

through repeated fork operations followed by executing the true binary using `execve`.

The results, depicted in Figures 3a and 3b, indicate that neither the busy loop nor target code overwriting influences BTB entry persistence. Conversely, system activity exerts a substantial effect. As illustrated in Figure 3c, the AMD system, which accommodates one of the largest numbers of entries, demonstrates poor persistence, with no entries surviving after 10 `execve` executions. Conversely, Lion Cove maintains 230 entries after 10 executions, 130 after 100, and even 50 surviving after 1,000 executions. On ARM systems, entries do not survive fork/execve operations.

## 6 Attack Surface Analysis

In the previous section, we presented the relevant microarchitectural primitives for BTR. This section analyzes how an attacker can abuse these primitives across three real-world JIT engines. For an engine to be vulnerable, the attacker must be able to inject a BTB entry targeting JIT-compiled code, free that code, and reuse the memory so that a Spectre gadget is present at the old entry point.

### 6.1 Linux cBPF

The Linux kernel employs JIT compilation for both eBPF and cBPF filters. While eBPF filters require elevated privileges, cBPF filters allow an unprivileged attacker to deploy them in the kernel using LSF [32] or SECCOMP [33]. Although cBPF instructions are highly constrained, cBPF still permits attackers to repeatedly add and remove programs [63]. This section explores abusing this JIT compiler to exploit stale BTB entries. For demonstration purposes, we focus on SECCOMP, which runs its filters immediately before each syscall. While AMD processors are susceptible to BTR, we omit AMD hardware from this scenario, as AutoIBRS’s current implementation disables indirect branch prediction altogether in the Linux kernel. All experiments were conducted on the stock Ubuntu 24.04 kernel, version 6.14.0-27.

**6.1.1 Victim indirect branch.** The first requirement is a vulnerable (victim) indirect branch that targets JIT-compiled cBPF programs.

```

Speculate
05 90 ff 67 10 add eax, 0x1067ff90
05 90 ff 67 10 add eax, 0x1067ff90
-----> nop; jmp [rdi + 0x10]
-----> <any gadget>

```

Figure 4: Misaligned gadget within a cBPF program

The Linux kernel maintains a linked list of attached SECCOMP filters for each task (`task->SECCOMP.filter`), each of which holds the entry point of each filter. On kernel entry, the list is traversed, and each entry point is invoked via an indirect branch. This indirect branch, located in the `__bpf_prog_run` function, serves as our victim branch. The branch is not protected by speculation barriers or other hardening mechanisms, and its target (i.e., the cBPF entry point) is loaded from the `bpf_prog` structure, making it an ideal eviction target to expand the transient execution window.

For optimization purposes, the Linux kernel employs a bitmap cache that tracks syscall numbers permitted irrespective of arguments [71], allowing the kernel to skip filter execution for syscalls that are always allowed. To ensure a BTB entry is inserted, the attacker must impose a condition on an argument for the victim syscall. Alternatively, we observed that filters are always executed for invalid syscalls, as they lack a bitmap slot.

**6.1.2 Crafting gadgets.** The next step is to craft a Spectre gadget within SECCOMP filters by combining the target chunk code with a stale BTB entry that points to the old training chunk entry point. cBPF filters support only 32-bit registers and prohibit memory dereferences, intentionally preventing direct gadget injection by construction. However, we can exploit misaligned execution: by injecting a BTB entry that points inside the filter code, transient execution begins at a misaligned offset, reinterpreting the 4-byte immediates supported by cBPF as machine instructions.

Each filter receives a pointer to `seccomp_data` as its first argument, which includes attacker-controlled syscall arguments. We construct the gadget by embedding a 3-byte jump instruction that

**Table 2: Primitive results on the Linux kernel (cBPF).**

|                         | Raptor Cove | Lion Cove  |
|-------------------------|-------------|------------|
| Time (ms) per iteration | 0.069       | 0.073      |
| Chunk reuse rate        | 38%         | 38%        |
| Survived BTB Entries    | 1 (100%)    | 1 (100%)   |
| Expected leakage rate   | 5.7 KB/sec  | 5.4 KB/sec |

targets this first argument, enabling transient execution to jump to arbitrary addresses. As shown in Figure 4, the attacker can fill the cBPF program with add instructions whose 4-byte immediates encode a NOP followed by an indirect jump. If transient execution jumps to the second or third byte of these immediates, it then executes the indirect jump to the attacker-controlled syscall argument, allowing the attacker to transiently hijack control flow to any disclosure gadget in the Linux kernel [62, 63].

This misaligned gadget construction is specific to x86; ARM’s fixed instruction size precludes this approach. Aligned jumps into the cBPF program yield no straightforward gadgets on ARM, but the dispatcher’s return path may still contain usable ones, provided the attacker wins the epilogue’s narrow window. We later discuss an example return-path gadget when examining the residual attack surface under ideal FineIBT on x86 (see Section 8 and Appendix G).

Now that we can craft a gadget with cBPF filters, we need to reliably free and reuse the training chunk. Although a filter cannot be removed once attached to a thread, we follow prior work to force removal [63]: spawn a thread to install a filter and exit the thread to remove it.

To reliably enforce reuse, we study the allocator’s behavior. When allocating a cBPF program, the allocator scans the module region for the first sufficiently large gap. Consequently, freeing a program followed by allocating another program of the same size typically reuses the same address.

In addition to reuse, we also require a discrepancy between the old and new entry points. For security reasons, Linux adds a randomized offset of illegal instructions [8] at the start of each function, causing the location of the filter entry point to vary in memory. Although the resulting randomness already makes a different entry point likely, if the old offset is smaller than the new one, the stale BTB entry may target illegal instructions.

To avoid landing in illegal instructions and increase the reuse success rate, we spawn an extra thread that inserts a padding chunk before the training chunk. After inserting the BTB entry by executing the training chunk, we remove both the training and padding chunks. We then allocate the target chunk using twice the size of the training chunk, causing it to overlap both freed regions. Hence, the stale entry point references the middle of the target chunk.

To ensure that the padding and training chunks are allocated consecutively, we first fill gaps larger than the training size but smaller than twice that size. We do this by monitoring the module-region mapping with the prefetch side channel [22], allocating training chunks until we detect a new huge-page (2 MB) mapping, which is created once no suitable gaps remain [43, 63].

**6.1.3 Evaluation.** We implemented the aforementioned steps and evaluated our reuse primitive on the Raptor Cove and Lion Cove CPUs. For ease of evaluation, we extended the victim branch’s transient execution window by inserting a kprobe that flushes the memory location storing the cBPF entry-point address—an assumption we will lift later when presenting our end-to-end cBPF exploits. We inserted a disclosure gadget via a kernel module and provided its address as the first syscall argument, so that the injected indirect branch’s transient execution jumps to it.

Constrained by Spectre v2 defenses on Raptor Cove in the kernel, we inject only a single BTB entry per iteration. While, in principle, an attacker could amplify the number of inserted entries on Lion Cove by randomizing the kernel path or entry point, we did not employ such techniques to keep the evaluation general.

We measured the time per attack iteration (a single training and leak step), the reuse rate, and the misprediction rate. Table 2 presents our results. The reuse rate is 38%, which closely aligns with the expected rate, as shown in Figure 4, where 2 out of 5 locations yield a valid gadget. We observe a similarly high misprediction rate, indicating that our single stale BTB entry is not evicted. This is because we insert only three chunks (padding, training, and target), a strategy which does not trigger excessive branch activity. One attack iteration takes on average 0.069 ms (Raptor Cove) and 0.073 ms (Lion Cove). Based on these results, we estimate the leakage rate with the formula below.

$$\text{Leakage Rate} = \frac{1}{T \cdot \frac{1}{p}} \cdot s \quad (1)$$

where  $T$  is the time per iteration,  $p$  is the success (reuse) probability, and  $s$  the number of survived BTB entries (the amplification factor). This yields an estimated leakage rate of 5.7 and 5.4 kilobytes per second for Raptor Cove and Lion Cove, respectively. Note that this rate represents a baseline leakage rate; practical factors such as using eviction-based techniques in place of kprobe may degrade performance. Moreover, the effective leakage rate varies depending on the disclosure gadget used. For example, our first end-to-end exploit (Section 7.1) uses a gadget requiring 256 attempts per byte, while our second exploit (Section 7.2) uses a more efficient gadget that eliminates this overhead. Nonetheless, these results establish a general baseline for the achievable leakage rate.

## 6.2 SpiderMonkey

In this section, we study how Firefox’s JavaScript (JS) JIT engine (SpiderMonkey) [12] can be abused to perform BTR attacks. Among the major JS engines, we focus on SpiderMonkey since, similar to WebKit [34, 35] and unlike V8 [2], it only partially deploys site isolation to counter Spectre attacks throughout the ecosystem [57]. We favored SpiderMonkey over WebKit for greater familiarity, but we expect similar primitives to be present in WebKit as well—e.g., a quick analysis revealed a vulnerable indirect branch and successful reuse patterns despite targeted use-after-free mitigations [68].

As a concrete user-space instantiation of our attack model (Section 4), we dissect SpiderMonkey’s internals and identify exploitation primitives similar to the cBPF case. For simplicity, we use Firefox 143’s standalone JS shell instead of the full browser.

**6.2.1 Victim indirect branch.** JS calls are dispatched dynamically through function references. When a call cannot be served by inline caching or inlining, SpiderMonkey uses a shared trampoline named `IonGenericCallStub`, funneling every non-inlined call through a single indirect branch. Because this branch can target different JIT-compiled programs, it is an ideal BTR victim. Moreover, the branch is not protected by speculation barriers and attacker-controlled variables are available on the stack at the call site.

Unconstrained by kernel-space mitigations, we exploit history-based prediction to insert multiple BTB entries and amplify our signal. As done in Section 5.2, we build a sequence of conditional branches before the victim branch. By randomizing taken/not-taken patterns, we force the predictor to index different BTB entries.

**6.2.2 Crafting gadgets.** Next, we need to craft a target gadget. In SpiderMonkey’s Wasm backend, we identify a gadget-construction primitive in floating-point constant handling, allowing an attacker to create a large contiguous region of attacker-controlled bytes.

Specifically, many widely deployed ISAs support PC-relative data loads, which require constants to reside within a bounded displacement range. For example, ARM64 uses a 19-bit word-scaled immediate ( $\pm 1$  MB), while x86-64 supports a 32-bit signed byte displacement ( $\pm 2$  GB).

SpiderMonkey adopts this scheme for instructions placing constants spanning multiple bytes, such as `f64.const` and `v128.const`, onto the stack. The compiler emits PC-relative loads that reference entries in a literal pool embedded in the executable code buffer. A sequence of such instructions therefore creates a contiguous region of attacker-controlled values on an executable page. When interpreted as instructions, these values can yield a disclosure gadget.

This primitive avoids the need for misaligned execution, making the attack practical even on fixed-width ISAs such as ARM64. The assembler’s 1 KB cap on the distance from a load to its pool entry forces multiple local literal pools on ARM64, effectively replicating attacker-controlled gadgets without additional effort.

**6.2.3 Forcing reuse.** SpiderMonkey manages executable memory through an internal allocator tightly coupled with the garbage collector. The allocator reserves a contiguous 2 GB virtual region and manages it with a bump-pointer *cursor* that hands out memory in 64 KB *allocation blocks*. Wasm modules and JS functions larger than 64 KB receive one or more such blocks that are not shared with any other code segment. Smaller JS JIT functions, in contrast, are packed side by side into shared blocks.

This asymmetry is directly exploitable. Small JS functions allow us to cram many jump targets into a single 64 KB block, maximizing the density of mispredictable targets during training. Wasm modules, in turn, have exclusive ownership of their blocks, providing deterministic placement when a block is later reused to host gadgets. When a contiguous set of blocks is freed, the cursor rewinds to the base of the *freed* region rather than continuing to advance.

To mount BTR attacks, we can allocate a sequence of *training chunks* back to back with JS functions and free them together. We then allocate *target chunks* containing disclosure gadgets from Wasm code, reusing the same 64 KB blocks previously held by the training chunks. By evaluating strings as code, we can generate many small JS functions repeatedly. Empirically, 200 iterations reliably trigger and complete compilation for these functions.

The garbage collector (GC) must run before the allocator can reuse freed JIT code blocks. A block is returned only once *every* function residing in it has been collected. No explicit JS API exists to force GC, but allocating two 512 MB `ArrayBuffers` reliably triggers a *stop-the-world* GC that reclaims every freed block.

**6.2.4 Evaluation.** To assess end-to-end feasibility, we first evaluate the training and reuse primitive in isolation, then test the full chain. For ease of evaluation, we patched SpiderMonkey to add a `FLUSH+RELOAD (F+R)` probe and to flush the victim branch target from the cache, enlarging the transient execution window.

**Inserting BTB entries.** First, we evaluate our amplification technique for inserting BTB entries. The experiment uses two reload targets for the victim branch (both loading the F+R buffer plus a mock target). We tested the history configurations from Section 5.2 and measured the maximum victim branch mispredictions. On Zen 4, Cortex-X3, and Cortex-A76, we measured only 55%, 18%, and 25% of the previously achieved entries, respectively, reflecting interference from JS code-path background activity.

**Establishing correct address reuse.** An assembler appends each constant to a *pending* pool, flushed when the compilation unit ends or when the next instruction would push a recorded load beyond its reach.

On x86-64 no reach-driven flush ever occurs, so the assembler accumulates every constant of a compilation batch into one pool emitted when that batch finishes. Batches are cut by accumulated bytecode length against a fixed threshold, so a function exceeding it alone is compiled in isolation and carries its own pool. Their compilation order is nondeterministic, however, so we cannot place a pool at a chosen offset and must spray training chunks blindly. We keep each chunk at 0x200 bytes, the smallest size our JS template compiles to, so that a given address range holds as many trained targets as possible. Since landing on the few bytes of the gadget is then unlikely, we fix the gadget at the start of each pool and fill the rest with encoded relative jumps branching back to it. This *trampoline* grows the effective landing target to nearly the full pool.

On ARM64 the binding constraint is the assembler rather than the ISA, capping the distance from a pending load to its entry at 0x400 bytes. A single long function therefore carries many pools, and interleaving constant loads with load-free padding sets their size and spacing. Tuning both to a 0x600-byte period, matched by training chunks of the same size, places every pool head on a trained address, so speculation reaches the gadget directly and the trampoline behind it catches any landing that misses the head.

We instantiate a Wasm target chunk spanning 128 KB of sprayed training chunks and repeat over 100 iterations. To address the gap problem of Section 6.1.2, we pre-JIT the script’s utility functions and fill the residual space of already-open shared blocks, so every later spray starts at a block boundary. The target chunk then reuses the training chunks’ memory over a stable address range, despite the allocator occasionally advancing by a full block between JS allocations. On average, 49.2% of the reclaimed addresses fall inside a target pool on x86-64 and 97.6% on ARM64. The reused range stays stable across the whole lifecycle. On x86-64, the pool sits at a fixed offset from the first training chunk, so we identify which equal-sized chunks land on a trampoline and train only those thereafter, concentrating the BTB entries on trampoline-reaching addresses.

**Table 3: Primitive results on SpiderMonkey.**

|                         | <b>Raptor Cove</b> | <b>Lion Cove</b> | <b>Zen4</b> | <b>A76</b> | <b>X3</b> |
|-------------------------|--------------------|------------------|-------------|------------|-----------|
| Time (ms) per iteration | 27.6               | 15.93            | 20.2        | 49.9       | 79.0      |
| Chunk reuse rate        | 49.2%              | 49.2%            | 49.2%       | 97.6%      | 97.6%     |
| Max. BTB entries        | 588                | 567              | 2733        | 747        | 955       |
| Survived BTB entries    | 7 (1.2%)           | 6 (1%)           | 0           | 0          | 0         |
| Expected leakage rate   | 36 B/s             | 62 B/s           | 0 B/s       | 0 B/s      | 0 B/s     |

Constant blinding, the primary mitigation against constant spraying, is rarely applied by SpiderMonkey for performance reasons, and the gadget replicas spread across pools keep the injection rate high even where it is.

**BTB entry survival.** With the required primitives in place, we combine the full iteration to test post-reuse BTB entry survival. One full iteration injects the training chunks, executes them with randomized histories, triggers GC, and then injects the target chunks. We then re-execute the victim branch using the same history while reloading the F+R buffer. Finally, we trigger GC again to reclaim the target chunks and measure BTB survival with two reload targets.

As shown in Table 3, Zen 4, A76, and X3 exhibited no BTB entry survival after reuse activity. Intel CPUs, however, retained an average of 6–7 surviving BTB entries. Using the leakage-rate formula introduced earlier, this corresponds to an estimated leakage rate of 36 and 62 bytes per second for Raptor Cove and Lion Cove, respectively. Note that to mount an end-to-end exploit, an attacker also needs techniques to lengthen the victim branch’s transient execution window and a covert channel in Firefox to bypass inaccurate timers, as demonstrated in prior work [64]. At the same time, an attacker may be able to exploit additional optimizations such as core affinity scheduling to isolate GC-induced noise on a separate core and improve the leakage rate.

### 6.3 GraalVM

As our second user-space case study, we investigate the attack surface that the Spectre-hardened GraalVM JIT engine exposes through GraalPy, its Python implementation. We conduct experiments on x86\_64 Linux: the Java host is compiled as a Native Image, while the Python code it loads via the polyglot interface runs JIT-compiled. The polyglot interface is configured with `SpawnIsolate` enabled and `UntrustedCodeMitigation` set to `Software`.

**6.3.1 Victim indirect branch.** Like SpiderMonkey, GraalVM routes dynamic calls through shared call stubs, except that they are grouped by arity: all call sites with the same number of arguments share the same stub. The stub contains our victim branch, which jumps to the JIT-compiled code. Although all parameters are passed on the stack, the call stub holds parameters in general-purpose registers while preparing the stack frame. By crafting functions with eight or more parameters, six general-purpose registers (`rbx`, `rbp`, `r10`, `r11`, `r12`, `r13`) may retain the actual parameter values, substantially relaxing the stack-only input constraint we faced in SpiderMonkey. Similar to Section 6.2, we randomize the history prior to the victim branch to inject multiple BTB entries for amplification.

**Listing 1: Crafted gadget in GraalVM.**

|                                                                                                           |                                                                                                                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> 1 [ ... ] 2 // repeated multiple times 3 sec += arg_1[ att ] 4 att += arg_1[ sec ] 5 [ ... ] </pre> | <pre> 1 // Get masked offset 2 ptr_1 = arg_1 + att 3 offset = (ptr_1 - arena)&amp;mask; 4 // - Transient entry - 5 // Load secret value 6 sec = *(arena + offset); 7 ptr_2 = arg_1 + sec; 8 // Mask secret again 9 offset = (ptr_2 - arena)&amp;mask; 10 // Transmit 11 trans = *(arena + offset); </pre> |
|-----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**(a) Written Python code****(b) Generated code**

**6.3.2 Crafting gadgets.** The next step is to craft a disclosure gadget within the generated GraalVM code. Unlike SpiderMonkey, GraalVM stores floating-point literals in a non-executable region (rather than embedding them in the code buffer), making this technique ineffective. Furthermore, as discussed in Section 2, GraalVM employs software fault isolation (SFI) to mitigate disclosure gadgets [48]. All memory accesses are masked to a sandboxed memory region (the arena), preventing accesses outside this region. However, as illustrated in Figure 1, BTR allows us to transiently jump over the masking operation and access data outside the arena.

Similar to JavaScript, Python provides `compile()` and `exec()` to evaluate code at runtime. We exploit these primitives to generate modules containing functions with controlled size and structure, which are JIT-compiled via hotspot detection and serve as training and victim chunks. Specifically, we build a simple target function that performs a sequence of array accesses (Listing 1a).

Listing 1b shows a minimized version of the generated code (the full assembly is provided in Appendix C). We observed that the compiler emits two versions of this code (64 and 72 bytes), which are alternated during compilation. For both array accesses, the compiler inserts masking operations to constrain memory accesses to the arena. However, by transiently jumping to line 6 in Listing 1b, we can skip the first masking operation and perform an out-of-bounds access. Bypassing the second masking operation (line 9) is not required, as the F+R buffer can be placed inside the arena.

To gain control over the required variables, we insert additional dummy code at the beginning of the Python function to influence register allocation. This causes the compiler to assign `r11` to the offset (line 6) and `rbp` to `arg_1` (the F+R buffer), both of which are attacker-controlled at the victim branch. With this setup, we can load an arbitrary secret and encode it into the F+R buffer.

**6.3.3 Forcing reuse.** GraalVM manages JIT code memory very differently from SpiderMonkey. First, it does not reserve a dedicated virtual-address region for JIT code; each compilation requests memory directly via `mmap()`. When reclaiming code, the underlying pages are returned to the kernel via `munmap()`. On 64-bit Linux, a `mmap()` call without an address hint scans the process’s virtual address space top-down, selecting the first sufficiently large free region. As a result, the kernel’s search state remains anchored at the highest available address and does not depend on recently freed regions. Requests are handled at page granularity, so each compiled script is allocated its own dedicated executable pages. As with cBPF,

**Table 4: Primitive results on GraalVM.**

|                         | Raptor Cove | Lion Cove |
|-------------------------|-------------|-----------|
| Time (ms) per iteration | 542.06      | 587.83    |
| Chunk reuse rate        | 97%         | 95%       |
| Max. BTB entries        | 179         | 156       |
| Survived BTB entries    | 0           | 0         |
| Expected leakage rate   | 0 B/s       | 0 B/s     |

we employ padding and training chunks to control placement, and we minimize their size to one page to reduce the JIT compilation footprint. We oversize the victim chunk by one page beyond the training span to ensure that every training address falls within the victim chunk’s mapped range.

Dynamically generated Python code on GraalVM is subject to garbage collection, as are any other managed objects, but Python exposes a native API (`gc.collect()`) that triggers a collection cycle on demand. We invoke this interface directly to drive GC during evaluation, bypassing the heuristic required for SpiderMonkey.

**6.3.4 Evaluation.** We run experiments similar to earlier sections, this time against GraalVM on Raptor Cove and Lion Cove CPUs.

**Inserting BTB entries.** For ease of evaluation, we implement the training and testing of the victim indirect branch in C and load them into the GraalPy runtime via ctypes. We then follow the same benchmarking methodology as in SpiderMonkey to measure the number of BTB entries controllable at the victim branch.

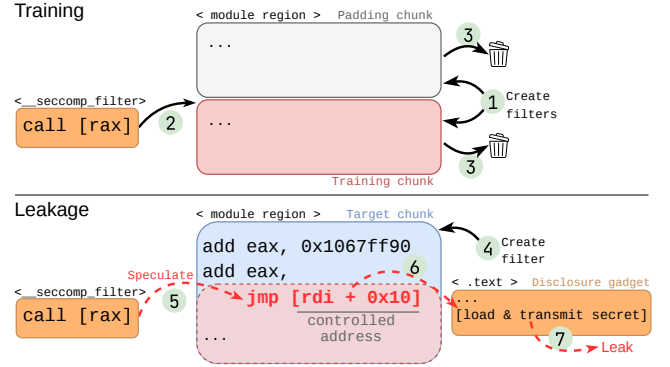
After JIT compilation stabilizes, we occasionally observe hit rates of up to 80%. However, the signal heavily fluctuates, likely due to background activity, resulting in an average hit rate of around 35%.

**Chunk reuse.** We construct training chunks of  $0x2000$  bytes and the victim chunk of  $0x4000$  bytes. When `mmap()` places both chunks starting from the same cursor position, their end addresses are always aligned, guaranteeing that every training chunk entry point falls within the victim chunk’s address range.

JIT compilation and GC are performed in the background, and the underlying memory allocations are not issued in first-come, first-served order. To enforce deterministic ordering, we insert a `time.sleep()` of 50 ms after each warm-up invocation and GC request, as determined empirically. Furthermore, we observe that allocated function memory is reclaimed after two GC cycles; we therefore insert additional GC calls to trigger reclamation and ensure sufficient free memory is available for reuse.

While training chunks occasionally fall into small pre-existing gaps at higher addresses and prevent address reuse, in the majority of iterations we achieve stable address reuse at up to 1.84 iterations per second using 4 training chunks and 4 target chunks.

**BTB Entry survival.** We next evaluate the survival rate when performing the required JIT compilation and GC activity. As shown in Table 4, we no longer observe any cache hits, indicating that no BTB entries survive under these conditions. This suggests GraalVM management noise precludes leakage—the GC activity, in particular, triggers the execution of many indirect branches. However, we do not believe the issue to be fundamental, as the current JIT behavior is inherently vulnerable without additional mitigations.



**Figure 5: BTR exploit with Linux kernel cBPF.**

The excessive noise hindering leakage may be resolved by further optimizations (e.g., core scheduling or other means to avoid GC interference) or better memory massaging.

## 7 End-to-end Exploitation

In this section, we present our end-to-end exploits using Linux kernel cBPF. We first cover exploitation on default Ubuntu, then analyze a cBPF configuration with constant blinding enabled.

### 7.1 Ubuntu kernel

Figure 5 presents a general overview of the attack. In the previous section, we discussed how an attacker can reliably force a reuse and craft a gadget in the target chunk. In this section, we will show how to use these primitives to create an end-to-end exploit and leak secret data. As done before, we use the default (precompiled) Ubuntu 24.04 kernel, version 6.14.0-27.

**Disclosure gadget.** To load and encode a secret, we can either encode a disclosure gadget inside the cBPF filter itself using the 4-byte constants, or jump to a disclosure gadget in kernel text. We choose the latter. As noted before, at the victim indirect branch the first argument (register `rdi`) is a pointer to the list of provided syscall arguments; the same pointer is also present in register `r13`. As a first step, we scan the Linux kernel text for two controlled loads so we can fetch a secret address and a `FLUSH+RELOAD` buffer address. To do so, we created a Python script that scans for two loads from `rdi` or `r13` from the controlled offset, or one load followed by two dependent loads. This yields 4,362 unique entry points.

Next, we instruct the state-of-the-art Spectre gadget scanner, InSpectre Gadget [62], to analyze the entry points for exploitable gadgets up to a depth of 5 basic blocks. InSpectre Gadget supports multiple gadget types. We enabled classic data-cache gadgets and directional-branch-predictor (DBP) gadgets [1, 10], the latter comparing a secret against a controlled value.

As shown in Figure 6, we found a total of 46 exploitable gadgets. We selected a data-cache gadget, listed in Appendix D, suitable for the `FLUSH+RELOAD` covert channel. The selected disclosure gadget requires a new training round per guessed secret byte (via the known-prefix technique [16, 58, 62]). While this reduces the leakage rate by a factor 256, we choose to demonstrate we can construct a full end-to-end exploit by using only an indirect branch in the JIT

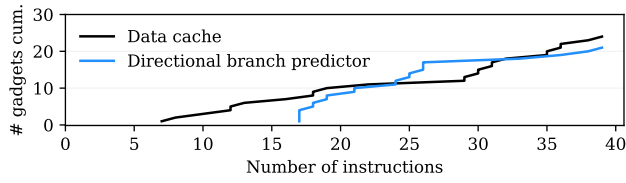


Figure 6: Found disclosure gadgets with matching control.

code. If needed, an attacker could easily gain more attacker control by jitting a code sequence (as shown in the next section), or by crafting the whole gadget in cBPF.

The gadget entry point is in the function `zlib_updatewindow` and calls `memcpy`, which transmits the secret. Although the gadget contains a conditional branch comparing an unknown 32-bit value (`esi`) with an attacker-controlled value that must be satisfied, the attacker can leak the required value byte by byte in the setup phase.

To speculatively jump to the disclosure gadget, we must provide its address as a syscall argument. This requires breaking KASLR. We use the prefetch side channel [22] to locate the kernel text start.

**Extending the transient execution window.** The next step is to extend the transient execution window of the indirect branch. Before the indirect branch resolves, the filter entry point is loaded through a chain of pointer dereferences: first a `seccomp_filter` pointer from the task struct, followed by a `bpf_prog` pointer, and subsequently the filter entry-point address. If any of these pointers are not available in the data-cache, the branch resolution will be delayed, thus increasing the transient execution window. To enforce this behavior, we construct a data-cache eviction set.

We aim to evict the `bpf_prog` struct from memory, as it is unique per filter and page-aligned, reducing interference and the number of candidate cache sets. However, since the pointer is specific to a filter instance, repeatedly removing and reinstalling the filter would invalidate the eviction set. To address this challenge, we create a dedicated victim thread in which the filter is installed once, allowing us to create and remove the other threads (padding, training, target) without affecting the eviction set.

We first construct an L2 eviction set, as we observed it improves the stability of L3 eviction on both Raptor Cove and Lion Cove. Using a 2 MB huge page, we compute the L2 cache sets and test all colors by timing the victim syscall. To avoid selecting eviction sets targeting unrelated data (e.g., kernel text), we execute a prime syscall between eviction and measurement to populate the cache along the execution path. We ensure the prime syscall is cached in the allow bitmap (Section 6.1.1), so it does not execute the filter and load the eviction target. This yields a clear timing signal of approximately 20 cycles, enabling reliable L2 set identification.

We then extend the eviction set to L3. We first repeatedly create a candidate subpool (approximately 15% of sets  $\times$  ways) until we observe a slowdown in the victim syscall, indicating eviction of the target pointer. Next, we minimize the set by removing addresses in small groups, similar to prior work [59]. We found that minimizing the set down to 2000 cache lines is fast enough to reliably evict the target cache line, while saving considerable eviction-set crafting time. We leave more optimized eviction sets to future work.

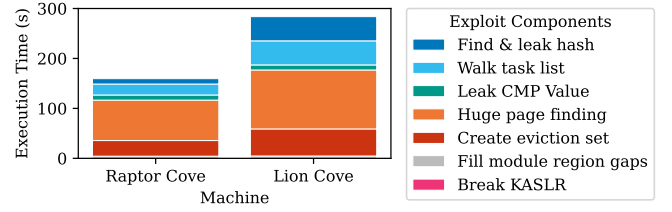


Figure 7: End-to-end exploit time breakdown.

**Finding a shared huge page.** To use FLUSH+RELOAD, we need a shared buffer between the attacker and the kernel. Drawing from prior work [62], we find such a reload buffer by allocating a user huge page and then brute-forcing its corresponding kernel address. We first find the direct map start via the prefetch side channel [22] and then probe kernel addresses sequentially in 2 MB strides.

**End-to-end exploit.** We can now construct the full end-to-end chain. We first break KASLR, fill all holes in the module region, and find an L3 eviction set for our victim filter. We then find the huge-page address and leak the value required to satisfy the comparison in our gadget. We evaluated our attack and achieved a leakage rate of 8 B/sec on average for both Raptor Cove and Lion Cove.

When we increase this rate by 256 (due to the chosen gadget), we obtain a rate only 2 times slower than the estimated leakage rate in Section 6.1, caused by walking the cache eviction set. We next show that even with just a few bytes per second of leakage, one can create an end-to-end exploit that completes within minutes.

**Leaking data from arbitrary processes.** In this section, we show how to leak arbitrary process data in a short time via pointer chasing. For demonstration purposes, we target the process memory of the `su` binary, but the attack can target any data from any process (e.g., leaking cookies from active browser processes).

By executing `su root`, `su` loads the root password hash into heap memory. To leak the hash, we start by finding the `task_struct` of `su` in memory. The Linux kernel contains a global variable `init_task`; since we broke KASLR, we know its kernel address. We find the `su task_struct` by iterating the linked task list from `init_task` and locating the target PID of the `su` process.

Next, we walk the process page tables to find the kernel addresses of the mapped memory. To avoid traversing all page tables, we first find the heap start user address by pointer-chasing into the `mm_struct` and leaking `brk_start`. We then leak the first-level page table address (pgd) and walk the page tables to find the kernel address for each user page in the heap.

For each mapped user page, we leak its contents. Because the `su` binary places the password hash at a fixed offset within a page, for each user page we need to check only one location for the root password-hash prefix ("root:"). Once we find the signature, we leak the full password hash. We evaluated the end-to-end exploit on both Raptor Cove and Lion Cove, and leak the password within 3 and 5 minutes on average, respectively. As shown in Figure 7, a large portion of the time is spent finding the huge page, which depends, among other factors, on the installed and used memory size. At the same time, one could implement techniques to search for multiple huge pages in parallel [62].

## 7.2 Bypassing constant blinding

To harden against JIT spraying attacks, the Linux kernel supports a BPF JIT compiler hardening option (`bpj_jit_harden`) which can be enabled at runtime via `sysctl` [31]. When enabled, it applies constant blinding by XORing immediate values with a random value and unmasking them before use, preventing direct injection of attacker-controlled immediates.

Although this mitigation is disabled by default, we now evaluate its effectiveness against BTR. Our previously demonstrated end-to-end exploit does not work as is, because it relies on encoding an indirect branch as an immediate value. However, prior work [44] showed that constant blinding can be bypassed by encoding attacker-controlled bytes in jump offsets. Applying this technique to cBPF, which has a constrained instruction set and a size limit, introduces additional challenges.

**Maximizing jump offsets.** The number of controlled bytes in a jump depends on the maximum jump offset. Thus, to gain sufficient control, we aim to maximize jump offsets. As cBPF programs are limited to 4,096 instructions, we select instructions that expand to as many bytes as possible after JIT compilation. Constant blinding works in our favor here, as it increases the code size. For example, a `mov` immediate normally compiles to 5 bytes, but expands to 16 bytes when blinding is enabled. We find that the `BPF_RET` instruction generates 21 bytes (two moves, one XOR, and a jump).

Using this build block, we construct a filter that begins with a jump to its end, while the body is filled with repeated 21-byte sequences. This maximizes the jump distance and yields an offset of `0x014fc5`, giving us control over two of the four offset bytes. Since the jumps are forward, keeping every offset below `0x10000` zeroes the two high-order bytes, which the misaligned stream absorbs: the crafted pair plus the first zero form one instruction, and the second zero starts an `add r/m8, r8` that consumes the next jump's `0xe9` opcode as its `ModR/M` byte, thus preserving the misalignment.

**Selecting a disclosure gadget.** With partial control over jump offsets, we can construct a gadget that redirects execution to a disclosure gadget. Initially, we targeted the same gadget used in our earlier exploit. However, since maximizing the offset requires larger, repeated instruction sequences, filter insertion becomes significantly slower—up to  $500\times$  (10 ms vs. 0.02 ms), depending on the required offset. Because the previous disclosure gadget required a new training round per guessed secret byte (via the known-prefix technique [16, 58, 62]), leaking a single byte becomes prohibitively expensive. To reduce overhead, we instead search for a more efficient gadget. Using InSpectre Gadget [62], we identify a gadget in `fuse_uring_commit_fetch` (Appendix E) that leaks a 16-byte secret, reducing the number of required training iterations.

**Encoding a gadget in offsets.** To exploit this gadget, we encode a control-flow sequence using the available jump offsets. The gadget must load attacker-controlled data and propagate control to both `rdi` and `rdx`. Due to limited control (two bytes) and the need to maintain valid misaligned execution, opcode selection is constrained. A minimized version of the resulting gadget is shown in Listing 2. The full version is available in Appendix F.

As direct access to offset 16 is not possible with a two-byte control, we first add the low byte of the syscall number (located at `rdi+0x0`) to `rax`. We then load the attacker-controlled value (first

Listing 2: Gadget crafted with jump offsets in cBPF exploit

```

1 ; < nop sled >
2 push rdi
3 pop rax
4 add al, BYTE PTR [rax+0x0] ; Add syscall nr low byte to rax
5 mov rax, QWORD PTR [rax] ; Load attacker controlled value
6 push rax
7 push rax
8 pop rdx ; Populate rdx
9 pop rdi ; Populate rdi
10 jmp QWORD PTR [rax] ; Jump to disclosure gadget

```

syscall argument) and propagate it to both `rdi` and `rdx`, before jumping to the disclosure gadget. The gadget body consists of 45 bytes. To increase the reuse rate, we inserted a 100-byte NOP sled before the gadget. The snippet has a correct misalignment for 3 out of 5 positions in the NOP sled. We evaluated the reuse rate; this consistently yields a 60% successful reuse.

**End-to-end exploit.** We integrated the constant-blinding bypass and the new gadget into our original end-to-end exploit chain, resulting in a leakage rate of 10 B/sec on both Raptor Cove and Lion Cove. Using our pointer-chasing technique, we recover the hash within 5 minutes on both CPUs.

## 8 Mitigations

When applicable, BTR can be mitigated using domain isolation mechanisms for JIT engines, such as site isolation [50]. However, implementing strict site isolation is hard [2] and its deployment throughout the ecosystem remains incomplete [34, 35, 57]. Moreover, in specialized JIT engines such as cBPF, domain isolation mechanisms are not readily applicable. In absence of such mechanisms, mitigating BTR can be approached in three main ways.

**Hindering gadgets.** A first approach is to restrict the attacker's ability to craft gadgets within JIT-compiled code [15, 29, 48]. Constant blinding is one such defense; however, as demonstrated by our cBPF exploit, this mitigation can be bypassed using implicit constants such as jump offsets. Similar considerations apply to other mitigations such as array index masking, which can easily be bypassed with control-flow hijacking. More generally, JIT engines inherently provide attackers with extensive control over generated code, making it difficult to rely solely on this strategy.

**Restricting speculation targets.** A second approach is to constrain the set of possible speculation targets for indirect branches. This can be enforced in hardware using Indirect Branch Tracking (IBT) on x86 or Branch Target Identification (BTI) on ARM, which require targets to start with an `endbr64` or `BTI` instruction. However, these defenses are not universally enabled (e.g., default-off on Ubuntu). In addition, while recent ARM architectures (v9.5+) prohibit speculation past instructions that are not BTI-instruction landing pads [4] (closing the attack surface in cBPF but not in JIT engines with literal pools such as SpiderMonkey), Intel CPUs have historically implemented IBT in a way that still allows transient execution of one or more instructions before the `endbr64` check [62].

We evaluated IBT effectiveness on the tested CPUs and found that Lion Cove is the first race-free Intel generation—with IBT allowing no transient execution before the `endbr64` check. By contrast, we confirmed that the tested Raptor Cove CPU allows one

instruction to be transiently executed [62]. We also verified that, by executing one JMP instruction in said IBT window, an attacker can inject a BTB entry that allows them to train a different indirect branch to an arbitrary target. Additionally, executing a memory-indirect jump (e.g., `jmp [rdi]`) in this window causes both the pointer and the target to be transiently accessed, enabling SLAM-style gadgets [23, 61]. Interestingly, this effect seems specific to indirect jumps; we did not observe it for indirect calls.

Even with race-free IBT, bypasses on x86 remain if attackers can embed the 4-byte `endbr64` in JIT-compiled code. With immediate values larger than 4 bytes, attackers can still place a misaligned gadget after `endbr64`; when that is impossible (e.g., in cBPF), they can jump into the middle of aligned JIT code and break invariants. For example, in cBPF, skipping the prologue redirects cBPF scratch-area stores to the caller's stack frame. As such, on an RSB-predicted return in a standard kernel configuration (with frame pointer enabled), transient execution resumes in the caller with part of the frame under attacker control. Depending on the stack layout (subject to kernel version, compiler, etc.), this allows attackers to control local pointers and execute Spectre gadgets. Appendix G shows an example where a controlled match pointer in the caller yields a dispatch gadget on the cBPF user notification path. We found no simpler exploitation avenues in cBPF, so IBT does raise the bar, but does not eliminate risk. Moreover, we were able to encode `endbr64` in JIT code only when constant blinding was disabled, suggesting race-free IBT plus constant blinding is a strong mitigation—albeit not future-proof since JIT changes (e.g., higher-entropy jump offsets, looser ISA) may re-enable BTR attacks. Finally, note that, as this method does not require misaligned execution, it is, in principle, also applicable to ARM without BTI. However, the cBPF return path on ARM is more complex than its x86 counterpart, further reducing exploitation opportunities.

**Restricting branch speculation.** A more comprehensive defense is to restrict indirect branch speculation. For the in-place BTR attacks we considered in the paper, we need to harden indirect branches that target JIT-compiled regions. However, when out-of-place mistraining is possible (e.g., with BHI [6, 62]), all indirect branches must be, in principle, considered vulnerable, as any of them may serve as a BTR entry point and consume stale BTB state.

On the software side, such branches can be guarded by retpolines [28, 56]. However, retpolines may impact fast-path performance and also yield a residual attack surface on certain microarchitectures [65]. Alternatively, recent hardware mechanisms allow disabling indirect branch prediction entirely (e.g., `IPRED_DIS`) [27], which is effective but causes significant performance degradation.

Another option is to invalidate branch predictor state when JIT code is removed, for example using Indirect Branch Prediction Barrier (IBPB) on x86 [25]. We believe this to be the most robust mitigation for BTR attacks. The performance impact may be acceptable, as long as JIT code removal is not on the fast path—as it is the case for cBPF. At the same time, its implementation is complex, as it requires careful tracking of which cores may execute the affected code and ensuring that the predictor state is flushed on all of them.

**Vendors' response.** All contacted parties acknowledged our findings (see Appendix B). The hardware vendors stated that mitigating mechanisms (e.g., IBPB) exist and that BTR mitigations

should be deployed in software. The Linux kernel and Oracle deployed mitigations. Linux kernel developers upstreamed a new mitigation for x86 that issues an IBPB on all cores when a cBPF program reuses a previously executed cBPF/eBPF region, and discourages such reuse as an optimization [20, 21]; this applies regardless of IBT, due to the return-path gadget (Appendix G). Oracle's mitigation instead hinders region reuse by randomizing JIT code-cache locations [47]. ARM indicated that the complexity of exploiting BTR in aligned cBPF execution did not warrant new kernel mitigations. Mozilla considered IBPB-based mitigations, but is currently prioritizing efforts to complete site isolation development and deployment [57]. Apple indicated similar preferences for WebKit.

## 9 Related Work

**JIT architectural attacks.** From an architectural perspective, our attack shares similarities with classic JIT Use-After-Free (UAF) attacks, which involve (i) freeing a JIT-compiled code region, (ii) reusing it with attacker-controlled code, and (iii) jumping to it via a dangling pointer to hijack control flow. This technique has been demonstrated previously on JIT code caches, where a UAF vulnerability (CVE-2018-0946) allowed injecting a gadget into the erroneously freed region, enabling arbitrary code execution [51, 70]. However, rather than dangling pointers, BTR exploits stale Branch Target Buffer (BTB) entries, eliminating the need for any software bugs.

To counter (architectural) control-flow hijacking, JIT engines have long implemented hardening measures such as constant blinding and JIT code randomization. Despite these mitigations, prior work [5] demonstrated bypasses in browser JIT engines by using short (1- and 2-byte) immediate values and leaking pointers. Moreover, others [44] demonstrated that opcodes can be encoded within jump offsets in browser JIT engines. We adapt the jump offset technique to circumvent constant blinding in cBPF, marking, to our knowledge, the first constant blinding bypass in BPF filters.

**JIT transient execution attacks.** From a microarchitectural perspective, the only previously known transient execution primitive exploiting self-modifying code (SMC) behavior in JIT engines is Speculative Code Store Bypass (SCSB) [49]. SCSB exploits the delayed synchronization between code and data views, whereas BTR desynchronizes the BTB state and code view due to insufficient BTB invalidation. Consequently, BTR offers a significantly larger attack window compared to SCSB, which prior work concluded to be too narrow for practical exploitation [49]. With BTR, we present the first practical end-to-end transient execution attack on SMC.

Beyond SMC, several generic transient execution attacks have targeted disclosure gadgets in JIT code, ranging from classic Spectre-BCB and Spectre-RSB variants [2, 13, 17, 36, 45, 46, 53, 64] to MDS [9, 52, 58], SLAP [35], and FLOP [34]. However, unlike BTR, no prior work has demonstrated a practical in-place Spectre-v2 attack in JIT engines and none has specifically targeted the high-value cBPF JIT engine in the Linux kernel.

**Spectre-v2 attacks.** Following the disclosure of Spectre [37, 42], much follow-on work has focused on the high-severity Spectre-v2 (and related) variants [18, 54, 55, 62, 63, 65–67, 72]. Spectre-v2 attacks traditionally rely on cross-domain training, which modern hardware mechanisms (e.g., eBRS, BHI\_NO) mitigate. Meanwhile,

*self-training* attacks—where both training and speculative control flow hijacking occur within the same (victim) domain—have recently been shown practical [63]. While BTR naturally falls into the in-domain category, our end-to-end cBPF exploits are also self-training, as the same cBPF indirect branch dispatcher is used for both training and exploitation. Prior (now mitigated) self-training attacks against the Linux kernel have also exploited cBPF filters, but they did so to manipulate branch history and inject training branches [63] rather than to desynchronize the BTB state and code view, as done in BTR.

## 10 Conclusion

We introduced *Branch Target Reuse* (BTR), the first practical in-place Spectre-v2 attack on JIT engines, and showed that self-modifying code can expose a long-lived transient execute-after-free primitive even when architectural code coherence is preserved. Our key observation is that stale indirect branch predictor entries survive code-cache reclamation and reuse, allowing attackers to transiently hijack control flow to newly generated code at obsolete and misaligned offsets. We evaluated BTR across three real-world JIT engines: SpiderMonkey, GraalVM, and the Linux kernel's cBPF JIT. The results demonstrate that BTR is a cross-engine issue, with exploitability characteristics subject to each engine's code-cache management and garbage-collection behavior. We found that cBPF is the most vulnerable target, due to its deterministic replay and reuse patterns, which enable reliable end-to-end exploitation. Focusing on cBPF, we developed two end-to-end exploits that recover the root password hash on Intel systems in minutes. Our results show that lightweight defenses such as constant blinding and FineBT are insufficient in isolation. Stronger mitigations like IBPB are effective, but come with higher complexity/performance costs.

## Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the “Allocamelus” project, by NWO through project “INTERSECT” and the Dutch Prize for ICT research, by the European Union's Horizon Europe programme under grant agreement No. 101120962 (“Rescale”), and by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

## References

- [1] Onur Acıçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *RSA*.
- [2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE S&P*.
- [3] ARM. 2018. Vulnerability of Speculative Processors. <https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability>.
- [4] Arm. 2026. Arm Architecture Reference Manual for A-profile Architecture. Arm DDI 0487 M.c, Part B, Chapter B2, § B2.5.5, “Restrictions on the effects of speculation from Armv9.5”. <https://developer.arm.com/documentation/ddi0487/>.
- [5] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines.. In *NDSS*.
- [6] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*.
- [7] Berkeley Architecture Research. 2026. BOOM: The Berkeley Out-of-Order Machine. <https://github.com/riscv-boom/riscv-boom>.
- [8] Daniel Borkmann. 2014. net: Bpf: Consolidate JIT binary allocator. <https://lore.kernel.org/all/1410156289-3190-2-git-send-email-dborkman@redhat.com/>.
- [9] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. 2022. Rapid prototyping for microarchitectural attacks. In *USENIX Security*.
- [10] Dmitry Evtushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ASPLOS* (2018).
- [11] Mozilla Foundation. 2026. MacroAssembler.h - Spectre Mitigations. [https://github.com/mozilla-firefox/blob/FIREFOX\\_151\\_0b3\\_RELEASE/js/src/jit/MacroAssembler.h#L4780](https://github.com/mozilla-firefox/blob/FIREFOX_151_0b3_RELEASE/js/src/jit/MacroAssembler.h#L4780).
- [12] Mozilla Foundation. 2026. SpiderMonkey. <https://spidermonkey.dev/>.
- [13] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking secrets to past instructions. In *ASHES*.
- [14] Anny Gakhokidze. 2021. Introducing Firefox's new Site Isolation Security Architecture. <https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/>.
- [15] Luis Gerhorst, Henriette Herzog, Peter Wägemann, Maximilian Ott, Rüdiger Kapitza, and Timo Hönig. 2024. VeriFence: Lightweight and Precise Spectre Defenses for Untrusted Linux Kernel Extensions. In *RAID*.
- [16] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative probing: Hacking blind in the Spectre era. In *CCS*.
- [17] Google. 2021. Leaky.page. <https://leaky.page>.
- [18] Jean-Claude Graf, Sandro Ruegge, Ali Hajiabadi, and Kaveh Razavi. 2026. Vm-scape: Exposing and exploiting incomplete branch predictor isolation in cloud environments. In *IEEE S&P*.
- [19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
- [20] Greg Kroah-Hartman. 2026. CVE-2026-64507: x86/bugs: Enable IBPB flush on BPF JIT allocation. <https://lore.kernel.org/linux-cve-announce/2026072554-CVE-2026-64507-5288@gregkh/>.
- [21] Greg Kroah-Hartman. 2026. CVE-2026-64508: bpf: Support for hardening against JIT spraying. <https://lore.kernel.org/linux-cve-announce/2026072554-CVE-2026-64508-fe26@gregkh/>.
- [22] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*.
- [23] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. 2024. Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation. In *IEEE S&P*.
- [24] Intel. 2003. Method and apparatus for self modifying code detection using a translation lookaside buffer. <https://patents.google.com/patent/US6594734B1>.
- [25] Intel. 2018. Indirect Branch Predictor Barrier. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html>.
- [26] Intel. 2018. 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>.
- [27] Intel. 2022. 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>.
- [28] Intel. 2022. 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>.
- [29] Di Jin, Alexander J Gaidis, and Vasileios P Kemerlis. 2024. {BeeBox}: Hardening {BPF} against transient execution attacks. In *USENIX Security*.
- [30] The kernel development community. [n. d.]. Classic BPF vs eBPF. [https://docs.kernel.org/bpf/classic\\_vs\\_extended.html](https://docs.kernel.org/bpf/classic_vs_extended.html).
- [31] The kernel development community. [n. d.]. Documentation for /proc/sys/net/. <https://docs.kernel.org/admin-guide/sysctl/net.html>.
- [32] The kernel development community. [n. d.]. Linux Socket Filtering aka Berkeley Packet Filter (BPF). <https://docs.kernel.org/networking/filter.html>.
- [33] The kernel development community. [n. d.]. Seccomp BPF (SECure COMputing with filters). [https://www.kernel.org/doc/html/latest/userspace-api/seccomp\\_filter.html](https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html).
- [34] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *USENIX Security*.
- [35] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *IEEE S&P*.
- [36] Jason Kim, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based timerless speculative execution attacks on Apple devices. In *CCS*.
- [37] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*.
- [38] Oracle Labs. 2018. GraalVM. <https://www.graalvm.org/>.
- [39] Oracle Labs. 2026. GraalVM Community Edition 25.0.2. [https://www.graalvm.org/release-notes/JDK\\_25/](https://www.graalvm.org/release-notes/JDK_25/).

- [40] Oracle Labs. 2026. Sandboxing. <https://www.graalvm.org/latest/security-guide/sandboxing/>.
- [41] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. 2024. Indirector: {High-Precision} branch target injection attacks exploiting the indirect branch predictor. In *USENIX Security*. 2137–2154.
- [42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: reading kernel memory from user space. In *USENIX Security*.
- [43] Song Liu. 2022. execmemalloc for BPF programs. <https://lwn.net/Articles/914128/>.
- [44] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of {JIT-ROP} Defenses. In *USENIX Security*. 139–156.
- [45] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.
- [46] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).
- [47] Matthias Neuschwandtner. 2026. [GR-77766] Randomize Runtime Code Cache Mappings with Allocator Hints. GraalVM pull request #14261, merged 19 August 2026. <https://github.com/oracle/graal/pull/14261>.
- [48] Matteo Oldani, William Blair, Shweta Shinde, and Matthias Neuschwandtner. 2025. Monocle: Transient Execution Proof Memory Views for Runtime Compiled Code. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*. 1188–1202.
- [49] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*.
- [50] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *USENIX Security*.
- [51] Google Security Research. 2018. Microsoft Edge Chakra - Cross Context Use-After-Free. <https://www.exploit-db.com/exploits/44758>.
- [52] Stephen Roettger. 2020. Escaping the Chrome Sandbox with RIDL. (2020). <https://projectzero.google/2020/02/escaping-chrome-sandbox-with-ridl.html>.
- [53] Stephen Roettger and Artur Janc. 2021. A Spectre proof-of-concept for a Spectre-proof web. (2021). <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>.
- [54] Sandro Rüegg, Johannes Wikner, and Kaveh Razavi. 2025. Branch privilege injection: Compromising spectre v2 hardware mitigations by exploiting branch predictor race conditions. In *USENIX Security*.
- [55] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *USENIX Security*.
- [56] Paul Turner. 2018. Retpoline: A software construct for preventing branch-target-injection. <https://support.google.com/faqs/answer/7625886>.
- [57] TyDraniu. 2026. Site-isolation on Android. <https://support.mozilla.org/en-US/questions/1562475#answer-1796582>.
- [58] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *IEEE S&P*.
- [59] Pepe Vila, Boris Köpf, and José Francisco Morales. 2018. Theory and practice of finding eviction sets. *arXiv preprint arXiv:1810.01497* (2018).
- [60] Luke Wagner. 2018. Mitigations landing for new class of timing attack. <https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/>.
- [61] Alan Wang, Boru Chen, Yingchen Wang, Christopher W Fletcher, Daniel Genkin, David Kohlbrenner, and Riccardo Paccagnella. 2025. Peek-a-walk: Leaking secrets via page walk side channels. In *IEEE S&P*.
- [62] Sander Wiebing, Alvis de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. 2024. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. In *USENIX Security*.
- [63] Sander Wiebing and Cristiano Giuffrida. 2025. Training solo: On the limitations of domain isolation against spectre-v2 attacks. In *IEEE S&P*.
- [64] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2022. Spring: Spectre returning in the browser with speculative load queuing and deep stacks. In *WOOT*.
- [65] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary speculative code execution with return instructions. In *USENIX Security*.
- [66] Johannes Wikner and Kaveh Razavi. 2025. Breaking the Barrier: Post-Barrier Spectre Attacks. In *IEEE S&P*.
- [67] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO*.
- [68] Haoran Xu. 2022. Understanding Garbage Collection in JavaScriptCore From Scratch. <https://webkit.org/blog/12967/understanding-gc-in-jsc-from-scratch>.
- [69] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*.
- [70] Dehui Yin. 2018. An Analysis of the Use-After-Free Bug in the Microsoft Edge Chakra Engine (CVE-2018-0946). <https://www.fortinet.com/blog/threat-research/an-analysis-of-the-use-after-free-bug-in-microsoft-edge-chakra-engine>.

- [71] YiFei Zhu. 2020. seccomp: Add bitmap cache of constant allow filter results. <https://lwn.net/Articles/834056/>.
- [72] Yuhui Zhu and Alessandro Biondi. 2025. Exploiting Inaccurate Branch History in {Side-Channel} Attacks. In *USENIX Security*.

## A Open Science

All artifacts are hosted at <https://doi.org/10.5281/zenodo.22109782>. It contains: (i) code to reproduce the microarchitectural experiments, (ii) code to evaluate the presented attack primitives, and (iii) both end-to-end exploits on the Linux kernel.

## B Ethical Considerations

We disclosed BTR to the major hardware vendors (Intel, AMD, ARM) and the Linux kernel security team in April 2025, presenting a preliminary attack surface analysis for JIT engines with a focus on cBPF, and recommending mitigations. The affected parties acknowledged the issue, placed the findings under embargo, but requested a full end-to-end exploit in order to justify the need to deploy costly mitigations such as cBPF constant blinding. In response, we developed two end-to-end cBPF exploits (including one also bypassing constant blinding) and extended our analysis to other JIT engines. We shared our updated findings with all the affected parties in May 2026, after which both the Linux kernel and Oracle deployed mitigations (see the “Vendors’ response” paragraph in Section 8). BTR has been assigned CVE-2026-64507 [20] and CVE-2026-64508 [21] by the Linux kernel.

## C GraalVM Disclosure Gadget

Listing 3 gives the full assembly of the disclosure gadget discussed in our GraalVM evaluation (see Section 6.3). BTR skips the initial masking sequence by transiently jumping to line 6.

Listing 3: Disclosure gadget crafted in GraalVM.

```

1 ; Masking sequence
2 lea r9, [rbp+rbx*4+0x8]
3 sub r9, r14
4 movabs r11, 0x7fffffff
5 and r11, r9
6 ; Gadget start (old entry point)
7 mov r9d, DWORD PTR [r14+r11*1] ; load secret value
8 mov r11d, r9d
9 add r11d, ebx
10 jo < error path >
11 add ebx, r9d ; rbx = secret value
12 test ebx, ebx
13 jl < error path >
14 nop DWORD PTR [rax+0x0]
15 cmp ecx, ebx
16 jle < error path >
17 cmp r8d, ebx
18 jbe < error path >
19 lea r9, [rbp+rbx*4+0x8] ; r9 = fr_buf (rbp) + secret (rbx)
20 sub r9, r14
21 movabs r11, 0x7fffffff
22 and r11, r9 ; r11 = secret value
23 mov r9d, DWORD PTR [r14+r11*1] ; transmit secret value

```

## D Linux cBPF Disclosure Gadget (Default)

Listing 4 lists the disclosure gadget used in the BTR end-to-end cBPF exploit under the default Ubuntu configuration (see Section 7.1).

**Listing 4: Gadget used in the BTR exploit on cBPF.**

```
zlib_updatewindow+11:
mov    r14, rdi ; rdi+0x10 points to syscall arguments
push   r13
push   r12
push   rbx
mov     r13, QWORD PTR [rdi+0x38] ; load secret address
sub     esi, DWORD PTR [rdi+0x20] ; load comparison value
mov     edx, DWORD PTR [r13+0x2c] ; load secret value
mov     r12d, esi
mov     rdi, QWORD PTR [r13+0x38] ; (irrelevant)
mov     rsi, QWORD PTR [r14+0x18] ; load base address
cmp     r12d, edx
jb     <zlib_updatewindow+91> ; base < esi - cmp?
sub     rsi, rdx
call   memcpy
...
memcpy:
xchg   ax, ax
mov     rax, rdi
mov     rcx, rdx
rep movs BYTE PTR es:[rdi], BYTE PTR ds:[rsi] ; transmit
```

## E Linux cBPF Disclosure Gadget (Blinding)

Listing 5 lists the disclosure gadget used in the BTR end-to-end cBPF exploit with constant blinding enabled (see Section 7.2).

**Listing 5: Gadget used in the BTR exploit on blinded cBPF.**

```
fuse_uring_commit_fetch+22:
mov     rax, QWORD PTR [rdi+0x8] ; load secret address
mov     rcx, QWORD PTR [rdx+0x230] ; load &base
mov     DWORD PTR [rbp-0x2c], esi ; (irrelevant)
mov     rbx, QWORD PTR [rax+0x38] ; (irrelevant)
movzx   eax, WORD PTR [rax+0x40] ; load secret 16-byte
mov     QWORD PTR [rbp-0x38], rbx ; (irrelevant)
test    rcx, rcx
je      <fuse_uring_commit_fetch+595>
cmp     rax, QWORD PTR [rcx+0x8]
jae     <fuse_uring_commit_fetch+612>
mov     rcx, QWORD PTR [rcx+0x18] ; load base address
mov     r12, QWORD PTR [rcx+rax*8] ; transmit
```

## F Linux cBPF Jump-Offset Gadget (Blinding)

Listing 6 details the cBPF gadget that bypasses constant blinding by encoding attacker-controlled bytes in jump offsets (Section 7.2), giving both (aligned and misaligned) views of the same byte sequence. Each `jmp rel32` is `0xe9` plus four offset bytes, only the first two attacker-chosen; the recurring `add cl, ch` is the second zero consuming the next jump's `0xe9`. The misaligned execution stream advances `rax` to the first syscall argument, loads the attacker-controlled value stored there, propagates it to `rdi` and `rdx` via the stack, and finally jumps indirectly to the chosen disclosure gadget.

## G Exploiting cBPF via Aligned Execution

On x86 with (race-free) IBT enabled, an attacker is confined to landing on a 4-byte immediate decoded as `endbr64`, after which execution continues aligned. Jumping into the middle of a cBPF program nonetheless exposes the speculative return path: Listing 7 shows an example RSB-return gadget, exploitable in a standard kernel configuration (with frame pointer enabled) if the compiler enforces a favorable stack layout. cBPF keeps a stack scratch area allocated by the program's prologue. Skipping that prologue via BTR redirects the program's controlled writes to the top of the

**Listing 6: cBPF gadget bypassing constant blinding.**

|                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>1 ; nop sled: 2 .rept 20 3     jmp +0x1490 4 5 6 .endr 7 ; gadget: 8 jmp +0x457 9 10 11 jmp +0x458 12 13 14 15 jmp +0x4002 16 17 18 jmp +0x8b48 19 20 21 jmp +0xc50 22 23 24 jmp +0xc50 25 26 27 jmp +0xc5a 28 29 30 jmp +0xc5f 31 32 33 34 jmp +0x20ff 35 ; 21-byte padding sequence 36 mov r10d, 0xc2c053d 37 xor r10d, 0x73d3053d 38 mov eax, r10d 39 jmp ...</pre> | <pre>1 ; nop sled: 2 .rept 20 3     nop 4     adc al, 0x0 5     add cl, ch 6 .endr 7 ; gadget: 8 push rdi 9 add al, 0x0 10 add cl, ch 11 pop rax 12 add al, 0x0 13 add cl, ch 14 ; +16: syscall_nr low byte 15 add al, BYTE PTR [rax+0x0] 16 add cl, ch 17 ; Load attacker-controlled value 18 mov rax, QWORD PTR [rax] 19 add cl, ch 20 ; Populate rdi and rdx 21 push rax 22 or al, 0x0 23 add cl, ch 24 push rax 25 or al, 0x0 26 add cl, ch 27 pop rdx 28 or al, 0x0 29 add cl, ch 30 pop rdi 31 or al, 0x0 32 add cl, ch 33 ; Jump to disclosure gadget 34 jmp QWORD PTR [rax] 35 ; 21-byte padding sequence. 36 mov r10d, 0xc2c053d 37 xor r10d, 0x73d3053d 38 mov eax, r10d 39 jmp ...</pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**(a) Decoded instructions (b) Decoded instructions during aligned execution misaligned execution**

caller's frame. Where the compiler places `match` (line 5) above `sd_local` (line 6), the attacker can thus speculatively overwrite it (line 9) and program the dispatch gadget (lines 15–18).

**Listing 7: Return-path gadget in `__seccomp_filter`.**

```
1 static int __seccomp_filter(int this_syscall,
2     const struct seccomp_data *sd, const bool recheck_after_trace)
3 {
4     u32 filter_ret, action;
5     struct seccomp_filter *match = NULL;
6     struct seccomp_data sd_local;
7     // [ ... ]
8     // executes cBPF program, may speculatively overwrite 'match'
9     filter_ret = seccomp_run_filters(sd, &match);
10    action = filter_ret & SECCOMP_RET_ACTION_FULL;
11    // [ ... ]
12    switch (action) {
13        case SECCOMP_RET_USER_NOTIF:
14            // use of 'match'
15            if (seccomp_do_user_notification(this_syscall, match, sd))
16                // May speculatively execute:
17                // (call) wake_up_poll(...), __wake_up_common(...)
18                // list_entry(match->wqh.head, ...) -> func(...);
19    }
20 }
```