TONTOU: On the Exploitability of Time-of-Neutralization to Time-of-Use Windows

Daniël Trujillo
MIT CSAIL

Mengjia Yan
MIT CSAIL

Abstract

Recently deployed Spectre v2 mitigations *neutralize* branch predictor state when switching privilege contexts or immediately prior to indirect branch execution, either through domain isolation or sanitization. These defenses assume that subsequent branch predictor behavior remains free from attacker influence until the neutralized state is used.

Unfortunately, this paper shows that this assumption does not hold on recent AMD and Intel CPUs. We find that post-neutralization (Time-of-Neutralization to Time-of-Use, TONTOU) windows can be exploited by an attacker to re-poison the predictor. Specifically, within the post-neutralization window, the attacker can re-direct control-flow of the victim to a training gadget that updates the predictor.

To re-direct control-flow, we introduce INTERRUPT INJECTION, a primitive that exploits post-neutralization windows by leveraging the fact that interrupts can occur at nearly any point in time. Using this primitive, we demonstrate that an attacker can trigger mispredictions during kernel execution on recent AMD and Intel CPUs. To prove its practicality, we build an end-to-end exploit using INTERRUPT INJECTION that leaks arbitrary kernel memory on AMD Zen 2 at a rate of 5.47 bytes/s, despite the latest neutralization techniques.

1 Introduction

Speculative execution attacks, such as Spectre v2 [18], have been shown to be a serious threat to modern systems. Spectre v2 attacks exploit mispredictions of indirect branches to leak secrets from cryptographic libraries and the kernel. These attacks widely affect processors from the major vendors, including Intel and AMD. Spectre v2 attacks train branch predictor units, such as the Branch Target Buffer (BTB), Branch History Buffer (BHB), or Return Stack Buffer (RSB), causing a victim branch to consume poisoned predictor state. The victim branch speculatively jumps to an attacker-chosen location, typically containing a so-called disclosure gadget that leaks secrets in registers or memory via microarchitectural side channels [11, 25, 39]. Many Spectre variants have been demonstrated [8, 31, 35–38], and various mitigations have been deployed in response [3, 12, 13, 26, 32].

This paper focuses on studying one class of mitigations, which relies on *neutralization*. The key idea of neutralization is to sanitize or isolate microarchitectural states (i.e., branch predictors) either upon context switch, such as eIBRS on Intel [13], or immediately before the execution of the protected branch, such as `saferet` on AMD [12]. We call the former *entry neutralization*, and the latter *in-place neutralization*.

It is important to thoroughly understand the security limitations of neutralization-based mitigations, as they are the dominant deployed defense in today's systems, and continue to evolve. More importantly, we find these mitigations share the same underlying assumptions, which may allow attackers to break them simultaneously, and thus the stakes are high.

This paper In all the neutralization-based mitigations, there exists a "*post-neutralization window*", from the time point when the predictor state is neutralized to the time point when the state is used by the victim branch. Most mitigations heavily rely on a critical yet implicit security assumption: within this post-neutralization window, the sanitized predictor state will not be interfered with by an attacker. However, this assumption does not hold if an attacker is able to re-direct the control flow of the victim during the post-neutralization window to re-poison the predictor state. We name this class of attacks TONTOU (Time-of-Neutralization to Time-of-Use) attacks. We investigate whether TONTOU attacks are actually practical on real-world systems, as sometimes the post-neutralization window can be quite short (just a few instructions), especially in the case of in-place neutralization.

We innovate a highly controllable method to achieve the re-direction operation via *interrupts* and propose INTERRUPT INJECTION as an example of a TONTOU attack. Specifically, our attack builds upon the insight that interrupts can happen at nearly any point in time, and unprivileged user programs can schedule timer interrupts to occur during kernel execution. Therefore, we can force the kernel to be re-directed to the

interrupt handler and use this handler to poison microarchitectural states within the post-neutralization window.

Our INTERRUPT INJECTION attack breaks a widely-held assumption that interrupts are benign as they are part of the kernel code. Even if interrupts are triggered during the post-neutralization window, execution of the interrupt handler is traditionally assumed to only introduce noise, and likely infeasible to be leveraged by attackers. However, we find this assumption to be far from correct. We introduce different poisoning mechanisms using the interrupt handler to broadly target *all* types of indirect branches.

Consider indirect jumps and calls, whose prediction outcomes are determined by the BTB, BHB, and IBP. Two indirect branches need to have matching instruction addresses or histories (or hash thereof) to achieve predictor aliasing. We find that injecting the same interrupt prior to a training branch and a victim branch can re-align their path history and increase the chance of aliasing. As a result, an attacker can effectively poison a victim-consumed predictor entry.

INTERRUPT INJECTION can also cause attacker-induced mispredictions of return targets, which are retrieved from the RSB. During interrupt execution, the RSB state may be populated with return sites of the interrupt handler. However, since these targets are determined by the interrupt handler's own control flow rather than the attacker, passive RSB pollution alone is challenging to exploit. To overcome this, we combine INTERRUPT INJECTION with Inception [31], which allows an attacker to actively train the RSB during interrupt execution with an arbitrary target. When the interrupt finishes, the subsequent return will be mispredicted to an attacker-chosen disclosure gadget, resulting in data leakage.

Practical attack demonstration We evaluate the feasibility of INTERRUPT INJECTION on four different processors from Intel and AMD. Our experiments demonstrate that it is relatively reliable to trigger timer interrupts from user-space and align them with a post-neutralization window as short as only a few bytes of instructions. The probability of hitting the post-neutralization window can be improved by increasing the interrupt frequency and enlarging the post-neutralization window. We achieve the latter by evicting the instructions in the window from the cache.

As a result, INTERRUPT INJECTION can bypass both entry neutralization and in-place neutralization. For example, we show that on Intel CPUs, we can leverage the interrupt handler to mistrain the BHB and thus the IBP. This subsequently causes misprediction of indirect jumps or calls in the kernel, despite mitigations designed to prevent a poisoned BHB state from being consumed (`SW loop` and `BHI_DIS_S`). On AMD processors, we show that the interrupt handler can mistrain the RSB using Inception and cause mispredictions in the kernel on AMD Zen 2, despite the `saferet` mitigation.

To show the feasibility of our primitive in a real-world context, we target in-place neutralization for an end-to-end attack, demonstrating the practicality of INTERRUPT INJECTION even with extremely short post-neutralization windows. Our end-to-end attack leaks arbitrary kernel data on AMD Zen 2 machines, despite the latest mitigations deployed. We furthermore show that INTERRUPT INJECTION is capable of locating `/etc/shadow`, containing the hashed root password.

Contributions In summary, this paper makes the following contributions:

- The identification of a shared assumption embedded in existing neutralization-based mitigations, and the class of TONTOU attacks that exploit this assumption.
- The new attack primitive INTERRUPT INJECTION that uses interrupts to re-direct the control flow of the victim during post-neutralization windows to re-poison branch predictor states.
- A demonstration of the potential impacts of INTERRUPT INJECTION on all types of indirect branches across Intel and AMD processors.
- An end-to-end attack that leaks arbitrary kernel memory on AMD Zen 2 machines at a rate of 5.47 bytes/s with an accuracy of 91.97%, despite the latest mitigations. INTERRUPT INJECTION finds `/etc/shadow` in 18 minutes, in 5 of 10 attempts.

Responsible disclosure We disclosed this research to AMD and Intel on February 5th, 2026. Both vendors confirmed the underlying behavior. AMD informed us they plan on mitigating the attack through a kernel patch. Intel rewarded this research with a discretionary bonus under its Bug Bounty Program, but does not consider mitigation to be required, as practical exploitability "depends on many factors". Intel further stated that INTERRUPT INJECTION belongs to a category of attacks covered under existing guidance [17]. This guidance does not mention interrupts as a potential attack vector, nor do Linux's default mitigations prevent INTERRUPT INJECTION. We discuss mitigation strategies in Section 8.

Since Arm employs neutralization mitigations as well, we informed them on April 21st, 2026. Arm stated that INTERRUPT INJECTION as described in this paper falls under "passive leakage", which Arm does not "actively protect against".

2 Background

2.1 Speculative execution & branch prediction

The control-flow of a program is typically complex, involving various types of branches that update the instruction pointer in different manners. Direct unconditional branches simply change the instruction pointer to an instruction-encoded target. However, other types of branches are more dynamic.

Conditional branches either take the branch or continue execution sequentially, whereas indirect branches derive their target from a register or from memory. Determining the next instruction pointer usually involves dependencies that need to be resolved first. To avoid stalling, CPUs *predict* the destination of a branch to accommodate speculative execution of the target instructions. After dependencies resolve, the CPU determines whether the prediction was correct. If not, the processor is said to have performed *transient execution*, requiring it to discard results and re-steer to the correct instruction pointer. Since predictions are typically correct, speculative execution significantly speeds up execution of commodity processors.

To perform these predictions, the CPU is embodied with a structure called the Branch Prediction Unit (BPU). Its task is to continuously provide predictions for the address of the next instruction to be executed. When branches are resolved, feedback is provided to the BPU to allow improved predictions over time. Due to the different types of branches and their differences in efficient prediction strategies, the BPU itself consists of various components.

On Intel and AMD CPUs, the Branch Target Buffer (BTB) provides a prediction of the existence of a branch and its type, as well as a target for branches, typically for those that do not dynamically change [5, 21, 38]. The Pattern History Table (PHT) provides prediction for conditional branches by tracking their local taken or not taken histories. The Branch History Buffer (BHB) keeps track of a global history of the control-flow and is updated using hashes of recently executed branch addresses and target addresses [21]. The Indirect Branch Predictor (IBP) is used to predict the target addresses of indirect branches which behave dynamically [5, 21, 36]. The Return Stack Buffer (RSB) is a dedicated structure for predicting targets of return instructions. Upon each call, the following instruction address is pushed onto the RSB, and each return pops from the RSB to obtain a prediction target.

2.2 Spectre attacks & mitigations

Initially, it was believed that mispredictions are mainly a performance issue. However, mispredictions were shown to be a security issue as well, when Spectre was introduced in 2018 [18]. Specifically, by executing malicious code snippets, the CPU can be forced to mispredict to a *disclosure gadget*, leaking data through a side-channel such as the cache.

Spectre-v1 attacks cause conditional branch mispredictions. To mitigate Spectre v1, conditional branch targets are typically followed by an `lfence` instruction [14], halting speculation. Alternatively, registers are masked during speculation [27].

Spectre-v2 attacks mistrain indirect branches to trigger mispredictions by poisoning the BTB or IBP. SpectreRSB mistrains returns by poisoning the RSB [19]. A large number of mitigations have been deployed over the past years, and we provide a chronological view of how they evolved over time.

retpoline, RSB stuffing, jmp2ret, and saferet To mitigate Spectre v2, indirect calls and jumps were replaced by `retpoline` on Intel CPUs [32], which turns indirect calls and jumps into return instructions, sanitizing the RSB before return execution. AMD recommended `AMD-retpoline` [4], which places an `lfence` before every indirect call or jump to delay execution. However, this was shown to be insufficient [23], causing AMD to deploy regular `retpoline` as well. To mitigate SpectreRSB, RSB stuffing was deployed, which sanitizes the RSB with benign entries [15].

Retbleed showed that `retpoline` can be broken by causing RSB underflows on Intel, and BTB collisions on AMD [37]. On Intel, this was mitigated by enhancing it with call-depth tracking [40], re-stuffing the RSB upon underflow. On AMD, this was mitigated with `jmp2ret` [3], which replaces returns with a jump to a centralized return whose location is sanitized in the BTB upon kernel entry, preventing BTB collisions.

Later, Phantom showed that even non-branch instructions can mispredict due to BTB aliasing [38], which was partially mitigated by MSR bit SuppressBPOnNonBr [3]. Using Phantom, Inception pollutes the RSB from an arbitrary location, breaking `jmp2ret`. As a mitigation, AMD replaced `jmp2ret` with `saferet` [12], which furthermore executes a call before the centralized return, ensuring a benign top-of-the-RSB state.

IBRS variants and BHI prevention Newer Intel and AMD CPUs provide hardware mitigations to isolate BPU entries between user and kernel through Indirect Branch Restricted Speculation (IBRS) [13, 26]. Over time, this mitigation was enhanced, providing it automatically in the background, without needing to explicitly trigger it on context switch, through Enhanced IBRS (eIBRS) on Intel and Automatic IBRS (aIBRS) on AMD. However, Branch History Injection (BHI) showed that despite eIBRS, mispredictions can still be triggered by manipulating the BHB from user space instead [8].

While initially considered infeasible without unprivileged eBPF, InSpectre Gadget showed that BHI can be exploited natively [35], without inserting any attacker controlled code in the kernel. As a result, Intel recommends to execute a BHB-clearing sequence on kernel entry (`SW loop`) on older CPUs, and enable `BHI_DIS_S` on newer Intel CPUs, providing BHB isolation in hardware. The latest Intel CPUs provide `BHI_NO`, which is `BHI_DIS_S` that is automatically enabled [17].

Not much later, Training Solo [36] demonstrated leakage even with the `SW loop` and `BHI_NO`, by creating collisions within the kernel, without the need to manipulate the BHB from user-space. Since a number of their attacks are facilitated by classic BPF, Intel recommends clearing the BHB after execution of BPF gadgets. Intel also deployed a new instruction called Indirect Branch History Fence (IBHF), which acts as a history barrier in hardware.

Lastly, Branch Privilege Injection (BPI) discovered that the BPU is updated asynchronously on Intel CPUs [28], allowing an attacker to train kernel branch predictions despite eIBRS.

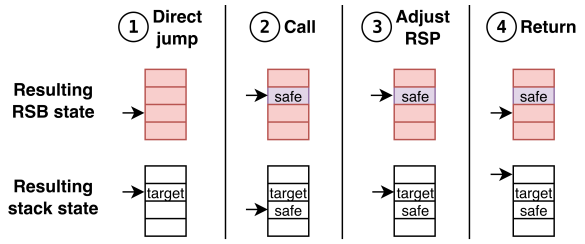


Figure 1: Overview of the instructions implementing `saferet`, which perform four steps. Every return is replaced with a direct jump to `saferet`. The call ensures a benign top-of-the-RSB state (`safe`, in purple), preventing misprediction to a disclosure gadget (in red). Before executing a return, the RSP is adjusted to architecturally ignore the pushed entry.

2.3 Phantom, Inception & `saferet`

We next expand on Phantom speculation, the Inception attack, and its mitigation (`saferet`). Phantom speculation occurs when the BPU predicts the existence of a branch at a non-branch code location [38]. Since the decoder needs time to understand whether a code location contains a branch, there will be a short window in which the processor's speculative control-flow is diverged to a BPU-provided target despite the absence of a branch, i.e., a *PhantomJMP*.

However, the speculative window of a PhantomJMP is typically short, preventing arbitrary memory leakage. Inception leverages Phantom on AMD CPUs to train the RSB during transient execution while the victim is executing [31]. If the BTB predicts a code location to contain a call instruction, the RSB is updated prematurely, before determining the existence of the call (i.e., a *PhantomCALL*). By creating a recursive PhantomCALL inside a PhantomJMP, Inception can corrupt multiple RSB entries with an arbitrary target while executing at an arbitrary victim code location. This causes subsequent return instructions to mispredict to an attacker-chosen target, providing a long transient window that allows data leakage.

Inception is mitigated using `saferet` [12], which is implemented through a sequence of instructions through which every kernel return is performed. Since the RSB state is untrusted at any time, `saferet` prevents consumption of the existing RSB state through four steps. Figure 1 depicts its operation, along with the resulting RSB and software stack state after each step. First, a return is replaced with a direct jump to `saferet` (①), at which time the RSB is untrusted. `saferet` then executes a call (②), neutralizing the RSB by pushing a safe target to the RSB. Since this also pushes the safe target on the software stack, `saferet` adjusts the software stack pointer (③). Lastly, a return is executed, guaranteeing misprediction to the safe target (④). After misprediction is corrected, execution will start at the intended target. To prevent RSB poisoning during `saferet` itself, BTB entries for the instructions implementing `saferet` are "untrained" on kernel entry.

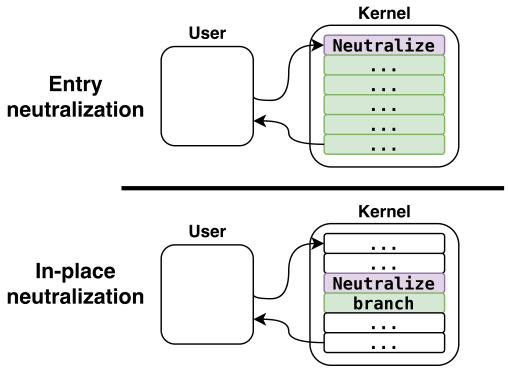


Figure 2: Two types of neutralization. Entry neutralization (top) is performed upon kernel entry, assuming no interference after entrance. In-place neutralization (bottom) is performed right before indirect branch execution, assuming no interference after neutralization and before branch execution.

3 Threat model

In this paper, we assume an attacker who can execute unprivileged but arbitrary code on the victim machine running Linux. The goal of the attacker is to leak data from the privileged kernel. We consider all deployed mitigations against Spectre v2 attacks to be enabled: aIBRS/eIBRS restricts speculative execution at targets inserted from user-space [13, 26], unprivileged eBPF is disabled [24], and user pointer sanitization is enabled [2]. Our AMD machines are susceptible to Inception, which is mitigated by `saferet` [12], sanitizing the RSB right before execution of a return. On Intel, the `SW loop` or `BHI_DIS_S` sanitizes or isolates the BHB upon kernel entry.

Our end-to-end attack assumes AMD Zen 2. We assume latest Spectre-v2 mitigations and other default system mitigations, including KASLR and `saferet`.

4 Motivation & overview

4.1 Analyzing Neutralization-based mitigations

We realize that while all mitigations seem distinct on first sight, the majority of them are essentially *neutralizing* the predictor state, to prevent a malicious predictor state from influencing speculative execution in the kernel. We define *neutralization* as the act of removing the influence of existing predictor state from a specific moment in time onwards, optionally until some end point in time. This includes both *sanitization* and *domain isolation*.

Figure 2 shows two classes of neutralization techniques: entry neutralization and in-place neutralization. In the case of entry-neutralization, a predictor is sanitized or isolated upon context switch, i.e., upon kernel entry. Examples are RSB stuffing, `jmp2ret`, `SW loop`, and automatic/enhanced IBRS.

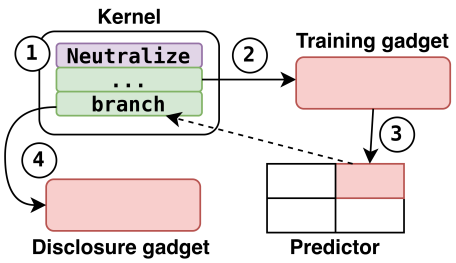


Figure 3: Steps of a TONTOU attack: ① neutralization, ② re-direction, ③ poisoning, and ④ use.

For in-place neutralization, a predictor is neutralized before every branch that is protected. Examples are `retpoline` and `saferet`, which stuff the top-of-the-RSB before each return.

For both types of neutralization, there are two critical time points involved in each execution of these mitigations: 1) the time of neutralization, when selected entries in a given microarchitectural structure (BHB or RSB) are cleared or isolated, and 2) the time of use, when the victim uses the microarchitectural state, such as when a branch is executed in the victim's address space. Importantly, there unavoidably exists a time window between these two time points, since neutralization and use are two separate events. We refer to this time window as the "*post-neutralization window*".

Despite the existence of a post-neutralization window, the following security invariant is assumed to hold: within the post-neutralization window, the neutralized microarchitectural states will not be interfered with by an attacker. That is, *the victim cannot be tricked into inserting poisoned microarchitectural states, either using speculation or using architectural control flow, within the post-neutralization window*.

Existing neutralization mitigations tentatively establish this security invariant by guaranteeing known and benign architectural control flow (pre-existing in the victim's code) and speculative control flow (avoiding interference by neutralized entries) during post-neutralization windows. As an example for in-place neutralization, take `saferet`. After neutralization of the top-of–RSB through sanitization, there is a limited number of fixed straight-line instructions before the protected branch, and thus it is easy to verify the architectural control flow. In terms of the speculative control flow, `saferet` relies on "untraining" the BTB on entry, to avoid any mispredictions during the post-neutralization window. For entry neutralization, there usually exists a potentially long code path from the entry point to the protected branch, and as a result, it is relatively difficult to analyze the architectural control flow.

4.2 TONTOU attacks

In this paper, our first contribution is to identify a class of microarchitectural attacks, which we name as Time-of-Neutralization to Time-of-Use attacks, or TONTOU attacks for short. The fundamental concept is very similar to the Time-of-Check to Time-of-Use (TOCTOU) attacks from the software security domain. Specifically, we exploit the possibility of re-poisoning the neutralized microarchitectural states within the post-neutralization window.

Figure 3 describes the high-level attack procedure.

- Step 1: Neutralization: the mitigation scheme neutralizes microarchitectural states to implement the protection, which marks the starting point of the post-neutralization window.
- Step 2: **Re-direction**: the attacker re-directs the control flow of the victim program to a training gadget at an attacker chosen point of time, either architecturally or speculatively. This re-direction happens before the protected branch is executed.
- Step 3: **Poisoning**: the attacker uses a training gadget to re-poison the target microarchitectural structure.
- Step 4: Use: the control flow is returned to the victim and the protected branch uses the poisoned microarchitectural states to speculatively jump to a disclosure gadget and leak secrets via side channels.

As described above, the attack consists of two components: re-direction and poisoning, where poisoning can reuse existing training primitives [6, 18, 29, 31, 38]. In this work, we particularly innovate in introducing a convenient, highly stable, widely applicable approach for the re-direction step, via *interrupts*. Modern systems are designed to handle interrupts at any moment, such as when a key is pressed, when a network packet arrives, or when a timer fires. Only under certain architectural and microarchitectural conditions (e.g., interrupt masking), interrupts may be delayed. Therefore, interrupts offer a convenient mechanism to re-direct control flow during kernel execution. This is the basis for our INTERRUPT INJECTION re-direction primitive.

As we discuss in related work (Section 9), several prior works can also be classified as TONTOU attacks. In the following discussion, we present technical details about the INTERRUPT INJECTION attack, achieving the re-direction step via injected interrupts.

4.3 Challenges in INTERRUPT INJECTION

There exist several challenges to making an INTERRUPT INJECTION attack work on a commodity system, assuming a user-space attacker.

Challenge (C1)

How to re-direct the control flow during kernel execution via interrupts?

Following our threat model (Section 3), we aim to attack the kernel from unprivileged users. The question is how a user process can conveniently manipulate interrupts to make this happen during kernel execution.

We find that modern operating systems such as Linux offer various ways for user space programs to install *timers*. When the timer expires, it triggers a hardware interrupt. If the attacker chooses carefully, it is feasible to align the interrupt with kernel execution. In Section 5.1, we carefully study the characteristics of user-space-installed timer interrupts and identify a reliable way to align interrupts and kernel execution with high probability.

Challenge (C2)

How to align interrupts to occur during the post-neutralization window?

To use INTERRUPT INJECTION as a re-direction step after neutralization, the injected interrupts need to align with the post-neutralization window. While this window can be long (e.g., sometimes for entry neutralization), it can also be very short, such as for in-place neutralization. For example, `saferet`'s post-neutralization window consists of 2 instructions only. How to increase the probability of this occurring?

We address these challenges by injecting interrupts frequently, slowing down the victim program by evicting targeted instructions from the cache hierarchy, and sampling multiple times to leverage post-analysis. These methods have been used in many other works (e.g., [33, 39]) to achieve different goals, such as increasing time resolution for attacks or improving attack success rates. In Section 5.2, we provide a detailed analysis of how these methods affect the success rate of an interrupt hitting the post-neutralization window.

Challenge (C3)

How to poison the microarchitectural structure entry used by the target branch from the interrupt handler?

Depending on the prediction units and microarchitecture, we consider two options to achieve poisoning after our re-direction through interrupts: *active* poisoning methods and *passive* poisoning methods. With an active poisoning method, the attacker re-uses previously proposed vulnerabilities to interfere with interrupt execution to poison a target predictor (e.g., Spectre-v1 [18], Phantom [38] or Inception [31]). In a passive poisoning method, the attacker relies on entries inserted by interrupt execution without actively interfering with its execution (e.g., updates to the BHB). In Section 5.3, we show how both passive and active methods can be used to manipulate the branch predictor state through interrupt re-direction on Intel and AMD CPUs respectively.

5 INTERRUPT INJECTION

To bypass neutralization, an attacker wishes a hardware interrupt to occur exactly during the post-neutralization window of a particular victim system call. We perform experiments to investigate feasibility on four machines with different CPUs: Zen 2 (`Ryzen 7 4700G`), Zen 4 (`EPYC 9124`), Intel Cascade Lake Refresh (`Xeon Gold 5220R`) and Intel Arrow Lake (`Ultra 9 285H`). They are running different Linux kernels between `5.15.0-168-generic` and `6.14.0-37-generic`. All default mitigations are enabled, and we do not fix the frequency or isolate cores during our experiments, to accurately reflect real-world scenarios. However, we do pin all our experiments to one core to avoid being moved to a different core by the scheduler. On Arrow Lake, we limit our experiments to its P-cores (`Lion Cove`).

To estimate the baseline number of interrupts that occur on these machines, we inspect `/proc/interrupts` before and after doing a busy loop for 10 seconds. The results confirm that we have around 100-1000 interrupts per second on all machines, which aligns with the configured kernel tick rate (`CONFIG_HZ`) used to build the kernels.

These baseline interrupts are unlikely to fire during the post-neutralization window of our victim system call. Therefore, we instead focus on *injecting* interrupts as an attacker, i.e., actively causing hardware interrupts at attacker-desired times.

5.1 High-resolution timers

To inject interrupts, we leverage the user-accessible high-frequency timer (`hrtimers`) interface. `hrtimers` was introduced to allow accurate delivery of timer signals independent of the periodic system tick, significantly reducing overhead and variable latency compared to the legacy timer wheel [10]. User-accessible `nanosleep`, `posix-timers` and `itimer` interfaces all use `hrtimers` under the hood, and in-kernel modules (e.g. multimedia) also use the subsystem.

The benefit of `hrtimers` is that they are highly controllable by an attacker, who can schedule timers with a time resolution at nano-second granularity. Second, `hrtimers` can be scheduled for a particular core, increasing interrupt predictiveness.

However, that does not mean causing interrupts during the post-neutralization window is trivial using `hrtimers`. Specifically, timers are scheduled relative to wall-clock time, whereas the execution timing of instructions within the post-neutralization window has high variability due to microarchitectural state and core execution frequency differences.

Periodic timers We opt to schedule periodic `hrtimers`, as offered by Linux, since this would let an attacker schedule a series of interrupts from a single timer, rather than arming a new timer for each one. However, through experimentation and reading source code, we find that unless the timer's expiration count is read, periodic timers do not continue to trigger

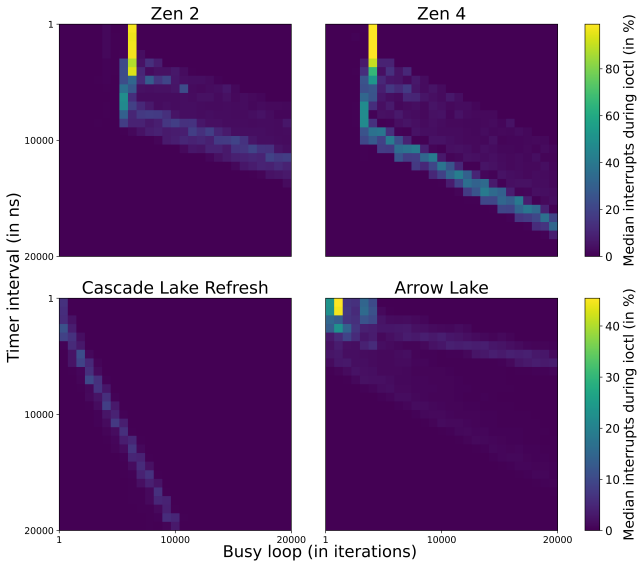


Figure 4: Success rate of injecting interrupts which occur during execution of a victim system call (`ioctl`). Median of 10 attempts, where each attempt consists of 10,000 iterations.

hardware interrupts. In particular, only when the timer's expiration count is read, the next hardware interrupt is scheduled. This is to prevent DoS attacks [22]. We therefore schedule a periodic timer, reading the expiration count repeatedly.

Periodic timers still have an important benefit over timers that fire once. An attacker wants to repeatedly trigger interrupts during kernel execution, and thus runs many iterations, each invoking the kernel. In particular, the attacker wants exactly one interrupt per iteration, occurring at a predictable moment. However, interrupt delivery is not perfectly predictable. Depending on execution speed and microarchitectural conditions, an interrupt armed in an iteration may be delayed and fire during a later iteration. A periodic timer avoids this: reading the timer's expiration count each iteration blocks in the kernel until the previous interrupt has fired. This resynchronizes an attacker's iteration loop with interrupt delivery. That is, periodic timers enable *iteration synchronization*, which ensures that one interrupt occurs per iteration.

To understand the likelihood that a timer-induced interrupt fires during a short system call, we create a kernel module that registers itself as a kernel probe on the timer interrupt handler path (`__sysvec_apic_timer_interrupt`), logging interrupts that occur during an `ioctl` system call. We set up a user-space application using `timerfd_settime` to schedule a high-frequency timer to fire every `x` nano-seconds, where $1 \leq x < 20,000$. For 10,000 iterations, after scheduling our periodic timer, we then ① read the timer expirations, ② busy-loop for `y` iterations, and ③ execute a victim `ioctl` system call to our kernel module, where $1 \leq y < 20,000$. Our kernel module returns immediately when called from user-space.

Figure 4 shows the results. We can vary the busy loop iterations and timer interval to align a large fraction of the desired interrupts with our victim system call execution, on all tested machines. Our maximum success rates across the parameters are 97% on Zen 2, 99% on Zen 4, 11% on Cascade Lake Refresh, and 45% on Arrow Lake.

Result (R1)

We can with high probability align hardware interrupts to occur during a target system call on all tested CPUs.

5.2 Interrupts during post-neutralization windows

For in-place neutralization, the post-neutralization window is typically very small (e.g., `saferet` is 2 instructions), whereas for entry neutralization, this window is typically larger (kernel entry up to vulnerable branch). That does not necessarily mean we want to trigger an interrupt anywhere during this large post-neutralization window, as depending on the predictor we wish to influence, other branches may also influence its state. Thus, to establish a desired predictor state, one may want to target a subset of the post-neutralization window that is followed by the last branch executed, up to the victim branch.

Now that we are able to trigger interrupts during kernel execution, we therefore investigate the feasibility of triggering them during a small post-neutralization window, whose size we measure in bytes. The same number of bytes can encode instructions with different latencies, that additionally depend on microarchitectural conditions. However, byte count serves as a proxy for execution time that is convenient to reason about across machines and binaries.

To perform this experiment, we call a dummy post-neutralization window in the `ioctl` handler of our kernel module. This dummy post-neutralization window consists of `x nop` instructions followed by a `ret`, where $0 \leq x \leq 63$. We change the timer interrupt kernel probe to only log interrupts that occur during the post-neutralization window. Using the results of Section 5.1, we optimize the number of interrupts that occur during the victim function.

Figure 5 shows the results (blue lines). On all tested microarchitectures, we see interrupts triggering during the post-neutralization window despite its small size.

Result (R2)

Injected hardware interrupts can be aligned with execution of a small post-neutralization window in the kernel.

We hypothesize that microarchitectural conditions such as dependencies on data or instructions that miss the cache influence the success rate, as they directly dictate how long it takes to execute our post-neutralization window.

To investigate the effect of our post-neutralization window

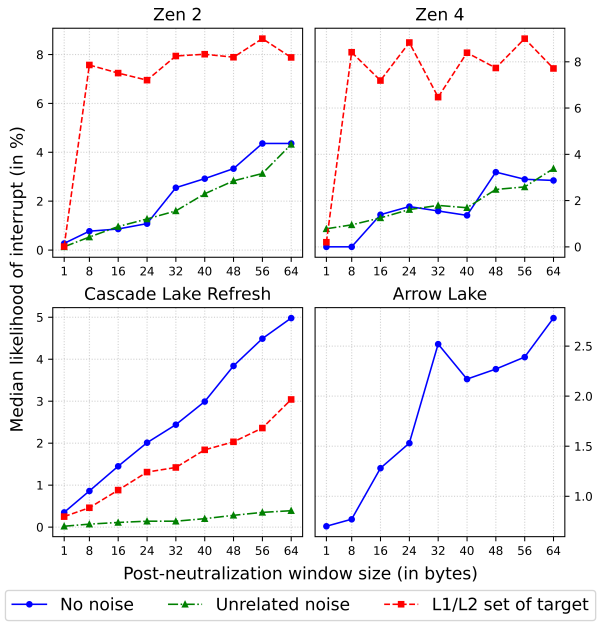


Figure 5: Probability of an interrupt occurring during the post-neutralization window, with no background noise (blue), unrelated background noise (green), and L1/L2 eviction (red). Median of 10 attempts, each consisting of 10,000 iterations.

missing the instruction cache, we set up a workload on the sibling hyperthread. We test two scenarios: 1) the workload accesses arbitrary addresses, and 2) the workload accesses the L1 and L2 cache set that the post-neutralization window falls in. For Zen 2 and Cascade Lake Refresh, we assume linear hash functions for both the L1 and L2. For Zen 4, we use previous reverse-engineering results for the L2 hash function [7]. We exclude Arrow Lake from our experiment, which does not support hyperthreading and has a per-core private L2 cache.

Our results are shown in Figure 5, with green lines (arbitrary addresses) and red lines (same L1/L2 cache set). On all three tested microarchitectures, accessing the same L1/L2 cache set as our post-neutralization window increases the success rate compared to accessing arbitrary addresses. On both AMD Zen 2 and Zen 4, the success rate is also significantly increased compared to not running a workload at all. For Cascade Lake Refresh, not running a workload yields a better hit rate than accessing the L1/L2 set that our post-neutralization window falls in. We hypothesize that this may be because hyperthreading de-stabilizes interrupt delivery. Therefore, on Cascade Lake Refresh, the simplest attacker setup (no hyperthreading) works best.

Result (R3)

By evicting the post-neutralization window from the L1/L2 cache, interrupts during the post-neutralization window increase significantly on our AMD CPUs.

The exact type of instructions we execute during the execution of the post-neutralization window also non-trivially influences the chance of an interrupt happening during its execution, since it impacts how long it takes to execute the post-neutralization window. We note that `nop` instructions are a conservative choice, since these instructions are not feeding any work to the backend. Therefore, we anticipate that the chance of an interrupt hitting the post-neutralization window would be higher when it consists of real-world instructions.

5.3 Poisoning with the interrupt handler

In Section 5.1, we introduced user-space accessible timers as a way to inject predictable hardware interrupts during kernel execution. In Section 5.2, we then showed that it is feasible to inject the interrupt such that it fires during execution of a very small post-neutralization window in the kernel (re-direct step of TONTOU). Our last goal is to leverage execution of the interrupt handler to train a particular branch predictor state (poisoning step of TONTOU). For all experiments in this subsection, we execute multiple victim system calls after injecting the interrupt. We will first focus on indirect jumps and calls in Section 5.3.1, and then continue to explore the exploitability of return instructions in Section 5.3.2.

5.3.1 Indirect jumps and calls

Indirect calls and jumps may mispredict due to branch location and/or branch history aliasing [8, 18, 35, 36]. To mitigate branch history aliasing, older Intel machines use a software loop (`SW loop`) that neutralizes the BHB state on kernel entry [17]. Newer Intel CPUs have hardware enhancements that automatically isolate user- and kernel history (i.e., `BHI_DIS_S` and `BHI_NO`). Newer AMD CPUs disable indirect jump/indirect call prediction in the kernel completely [36], while older ones replace them with return instructions (retpoline [32]). Since AMD machines do not have indirect jumps and calls left in the kernel that use speculation, we exclusively focus on our Intel machines in this subsection.

We hypothesize that interrupt execution may affect the state of the BHB. To determine if the interrupt handler changes the BHB and consequently IBP state sufficiently to cause mispredictions despite the mentioned mitigations, we perform the experiment depicted in Figure 6. We introduce the following code gadgets in the kernel: a training branch that includes an indirect jump instruction labeled as **T**, preceded by a neutralization step; a disclosure gadget (**D**) that accesses the address in register `rdi`; and a victim code snippet with a protected indirect jump labeled as **V** using the same neutralization operation. Our goal is to alias the entry used by the training branch (**T**) and the victim branch (**V**) in the IBP by synchronizing their path-history view. This synchronization is achieved by triggering the same interrupts during the post-neutralization windows before the execution of the two branches. We con-

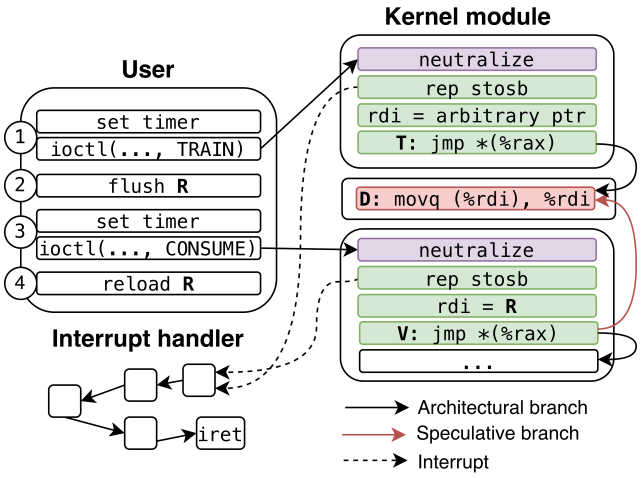


Figure 6: The interrupt handler executes during the post-neutralization window (in green) after neutralization (in purple), poisoning the BHB, which causes indirect call mispredictions to a disclosure gadget (in red).

CPU	Targeted predictor	Targeted neutralization	Success rate
Cascade Lake	IBP/BHB	`SW loop`	0.037%
Arrow Lake	IBP/BHB	`BHI_DIS_S`	0.22%
Zen 2	RSB	`saferet`	0.75%
Zen 4	RSB	`saferet`	0%

Table 1: Speculative load due to mispredictions in our kernel module because of interrupt training. Average of 100 attempts, each consisting of 100,000 iterations.

Result (R4)

Interrupts mistrain the BHB and thus the IBP, subsequently causing mispredictions in the kernel on Intel machines, despite the `SW loop` and `BHI_DIS_S` mitigations.

sider this as a passive poisoning mechanism, as we use the interrupt handler's default control flow to influence the BHB without additional attacker intervention, in contrast to the active poisoning mechanism targeting returns (Section 5.3.2).

From the attacker's perspective in user-space, there are four steps: ① setting an expiring timer and calling the kernel module to perform a training indirect jump **T** to the disclosure gadget **D**, ② flushing reload buffer **R** from the cache, which is shared with the kernel, ③ setting another timer and calling the kernel module to perform a victim indirect jump **V**, and ④ reloading **R** to determine if **V** speculatively called **D**.

The post-neutralization windows of the two jumps **T** and **V** (in green) are enlarged with a `rep stosb` instruction which copies `8192` bytes of data. If an interrupt happens during the post-neutralization window of **T** (re-direct step), the interrupt handler's execution (poisoning step) may impact the BHB state sufficiently such that the disclosure gadget **D** is inserted at a manipulated index of the IBP. If the same interrupt handler executes during the post-neutralization window of **V**, the history is affected similarly, potentially causing **D** to be served as a predicted target for **V**.

Our results are shown in Table 1. On both Cascade Lake Refresh and Arrow Lake, we can successfully cause mispredictions in our kernel module by injecting interrupts. Our timer interval and busy loop iterations are decided based on Section 5.1, and we believe a large search of these parameters can likely increase the success rate. We validate our results by overwriting the pointer **R** in memory using a `kprobe` *only* if an interrupt occurs. This prevents it from being loaded in `rdi` in step ③, and consequently prevents **D** from reading its contents. We confirm that this removes the mispredictions.

Address alignment & `BHI_DIS_S` Interestingly, although we align the lowest 20 bits of the instruction addresses of **T** and **V**, we find that this is not always necessary for mispredictions to occur. We hypothesize that this is because the initial BHB state at the time the interrupt fires differs across iterations, as the interrupt is triggered at a different point in the kernel code each iteration. The branch history generated by the interrupt handler itself may vary as well. Since the branch address is folded with history, there is in theory no need for any bit to match in the branch addresses [21].

To validate this assumption on Arrow Lake, we filter our results for only those interrupts which fire at a designated region without branches. We furthermore change Linux's return path to the kernel in `entry_64.S` to invoke the Indirect Branch History Fence (IBHF) instruction on our Arrow Lake, followed by calls to the `SW loop` to establish a deterministic and identical BHB state each time we return from an interrupt. After this change, the addresses of the training branch **T** and victim branch **V** need to align in bits 6 to 15 (i.e., 10 bits) for mispredictions to occur, agreeing with prior work [21].

The result for Arrow Lake is surprising, since previous work has shown that `BHI_DIS_S` completely disables history-based prediction in the kernel [28, 36]. Therefore, we test our experiment on Raptor Lake, an older CPU which also supports `BHI_DIS_S`. Our results prove not a single misprediction, suggesting that the implementation of the mitigation differs across generations. We hypothesize that `BHI_DIS_S` on our Arrow Lake has the same implementation as the newer `BHI_NO`, which allows history-based predictions according to prior work [36]. Our hypothesis is supported by the fact that the microarchitecture (Lion Cove) of our Arrow Lake is also found on Lunar Lake, which *does* support `BHI_NO`.

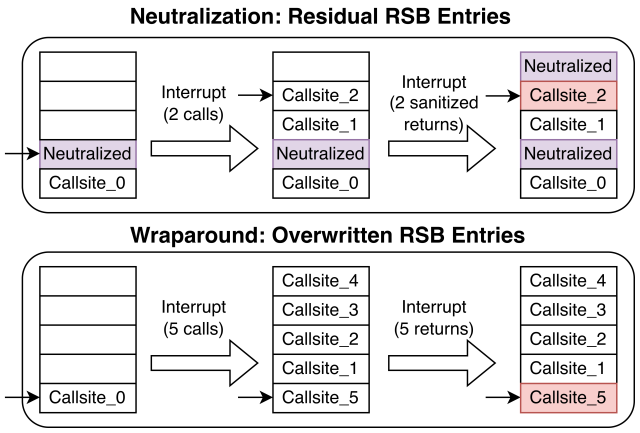


Figure 7: The interrupt handler inserts return targets in the RSB during the post-neutralization window. If in-place neutralization prevents consumption of these targets (top), or if the interrupt overwrites yet unconsumed RSB entries of the victim (bottom), misprediction of a victim return could occur.

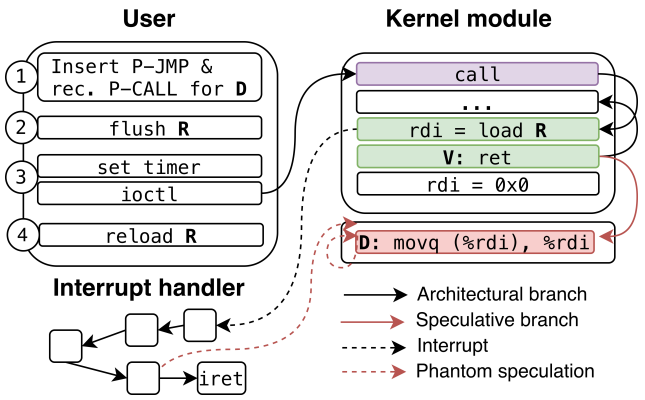


Figure 8: The interrupt handler executes in the post-neutralization window (in green), poisoning the RSB using Inception after neutralization (in purple), which causes return mispredictions to the disclosure gadget (in red).

5.3.2 Returns

We now test whether the poisoning step in INTERRUPT INJECTION can work on the RSB, and thus allowing us to control return misprediction. There exists a big difference between poisoning RSB versus BHB. For indirect jumps and calls, we use a passive poisoning method, i.e., we rely on the interrupt handler's default control-flow to manipulate the BHB, without an attacker interfering with the interrupt handler's execution.

However, in most cases such passive approach would not work to poison the RSB using the interrupt handler. RSB entries in use by the victim do not necessarily change because of an interrupt, since the number of calls and returns during the interrupt handler should be balanced. That is, upon interrupt return, the RSB entries consumed by the victim should hold identical targets compared to the case when no interrupt would have occurred. There are two exceptions to this:

1. Consider returns protected using in-place neutralization through sanitization, shown in Figure 7-top. After a call pushes an entry to the RSB, it is never popped by a return, since it consumes a newly pushed neutralized RSB entry instead. Therefore, after finishing the interrupt, the addresses following the calls made by the interrupt (e.g. `Callsite_2`) are on the RSB, and these residual RSB entries are used for speculation by the victim.

2. Second, consider entry neutralization and a circular RSB, as shown in Figure 7-bottom. If the interrupt handler makes a large number of calls, yet unconsumed RSB entries of the victim will be overwritten (`Callsite_5` overwrites `Callsite_0`). After finishing the interrupt, overwritten RSB entries (`Callsite_5`) are used for speculation by the victim.

These two scenarios are realistic. First, we have in-place neutralization on AMD Zen 1-4, namely `saferet` (case 1). Second, a victim return consuming overwritten RSB entries could happen on AMD CPUs and older Intel CPUs, which embody a circular RSB (case 2), especially on Zen 1-2, whose circular RSB only has 16 entries in dual-threaded mode.

However, in both cases, the speculative target of the victim return is chosen by the interrupt handler's execution path, not under the attacker's control. Therefore, it is unlikely to contain a disclosure gadget that can be used by the attacker.

Active poisoning using Inception As a solution, we consider a more active manner of poisoning the RSB. In particular, AMD Zen 1-4 CPUs are known to be vulnerable to RSB poisoning using Inception [31], which triggered in-place return neutralization mitigations. We therefore use Inception to arm the interrupt handler as a training gadget. Specifically, we consider an attacker which poisons the BTB such that during execution of the interrupt handler, the RSB is trained using PhantomCALLs with a desired disclosure gadget address.

Figure 8 depicts the experiment, where the post-neutralization window is shown in green. The attacker performs four steps: ① inserting a PhantomJMP and recursive PhantomCALL to be consumed by the interrupt handler, which later inserts the disclosure gadget **D** into the RSB, ② flushing a shared reload buffer **R** from the cache, ③ setting an expiring timer, and calling our kernel module which performs a call and subsequently a return **V**, ④ reloading **R** to determine whether **V** speculatively called **D**.

Since the post-neutralization window of in-place neutralization is very small, we do not add any additional `nop` instructions in the post-neutralization window. If the interrupt fires exactly during post-neutralization window (the re-direction step), the interrupt handler will execute, triggering a PhantomJMP to the disclosure gadget **D**, where recursive Phan-

tomCALLs will insert **D** in the RSB (the poisoning step). After the interrupt is over, the return in the post-neutralization window will mispredict to the disclosure gadget **D**.

Table 1 shows the results. On AMD Zen 2 we observe mispredictions, proving we successfully poisoned the RSB during interrupt execution using Inception. We again validate our result by overwriting `rdi` using a kernel probe *only* if an interrupt occurred during the post-neutralization window, which removes any mispredictions.

On AMD Zen 4, we do not observe any mispredictions. This is in line with the results reported in [31], which stated that the RSB is only partially corrupted on Zen 3-4 (i.e., top-of-RSB is unaffected). To confirm this is the issue, we execute multiple return instructions in the post-neutralization window without matching call instructions. This results in mispredictions. We note that later research showed that Inception can also poison the entire RSB on AMD Zen 3-4 if the PhantomJMP is triggered after a serializing instruction [6]. Therefore, if an attacker is able to find a serializing instruction in the interrupt handler, they may be able to successfully cause mispredictions on AMD Zen 4 as well.

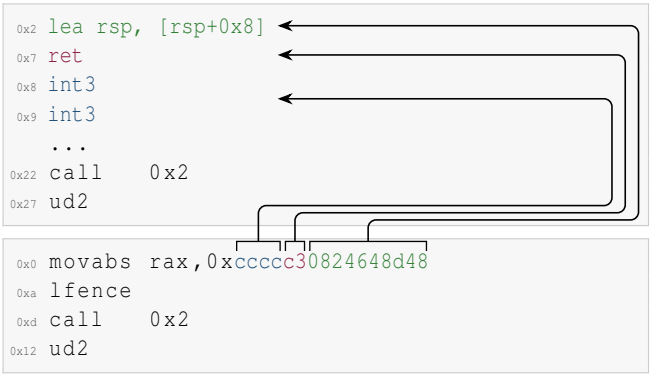
Result (R5)

Interrupts can mistrain the RSB using Inception and subsequently cause mispredictions in the kernel on AMD Zen 2, despite in-place neutralization for returns.

6 End-to-end attack

In this Section, we present our end-to-end attack which uses our insights from Section 5 to leak data on a default Linux kernel. Our techniques work on both Intel and AMD machines, as demonstrated in Section 5.3. However, leaking data from Intel CPUs has certain software requirements. First, we must find a disclosure gadget at a legitimate indirect branch target in the Linux kernel, as we only train the IBP with targets that the kernel branches into itself. Second, we must find an indirect branch during which we have sufficient register control for that particular disclosure gadget. While this complicates exploitation, these requirements can be realistically met in the Linux kernel, as shown by prior work [35, 36]. For example, in Linux v6.6-rc4, the chain of dereferences at the start of `cgroup_seqfile_show` (Listing 2 in [35]) is a disclosure gadget at a legitimate kernel target, and an indirect call in the syscall path provides sufficient attacker control over registers.

For our end-to-end attack, we focus on AMD Zen 2, where we can train the predictor with *arbitrary* targets. While this simplifies exploitation, we also note that the post-neutralization window on AMD Zen 2 is extremely small (6 bytes) since it performs in-place neutralization, making it harder to repeatedly inject interrupts during this window compared to entry neutralization on Intel CPUs.



Listing 1: `saferet` on AMD Zen 2 with two decodings of the same bytes. Every return in the kernel is executed by jumping to offset `0x22` (top), which neutralizes the RSB through a `call` before issuing a `ret`. At kernel entry, execution starts at `0x0` (bottom) to sanitize the BTB for offsets `0x2` and `0x7`.

Our AMD Zen 2 machine has 16GB of available memory, and runs Linux version `6.14.0-37-generic`.

6.1 Injecting interrupts during `saferet`

The in-place neutralization used on AMD Zen CPUs is the `saferet` mitigation, sanitizing the RSB right before executing a return. `saferet` is a sequence of bytes which can be decoded and executed from two different offsets. Listing 1-top illustrates how `saferet` neutralizes the RSB before return execution. Instead of executing a return directly, the kernel jumps to `0x22`, where a call instruction first pushes `0x27` to the RSB. The call lands at offset `0x2`, which adjusts the stack pointer using `lea` to remove `0x27` of the software stack. The `ret` at `0x7` then returns to the intended target. If the RSB was poisoned prior to neutralization, its top-of-the-stack value is overwritten with `0x27`, so the return speculates into the `ud2` at `0x27` rather than any attacker-controlled address.

The instructions implementing `saferet` must themselves be protected from Phantom speculation. An attacker which installs BTB entries for the `lea` or `ret` instructions could poison the RSB using Inception after the neutralizing call. On AMD Zen 1-2, these instructions are protected by executing their instruction bytes once from a different offset, as shown in Listing 1-bottom. Decoding these bytes from offset `0x0` yields a `movabs` instruction, whose immediate includes the bytes of `lea` and `ret`. Executing this `movabs` instruction at kernel entry removes potentially attacker-controlled BTB entries for these instructions. On AMD Zen 3-4, an address that aliases with `saferet` in the BTB is executed instead.

For our end-to-end attack, we aim to trigger an interrupt during execution of offsets `0x2` or `0x7`, which is after neutralization, but prior to the protected return instruction.

6.2 Victim system call

To leak data, we wish to poison the RSB in the post-neutralization window of `saferet`. For this, we need to identify a victim system call for which we observe interrupts during this window frequently. Furthermore, to be able to use the misprediction to leak data, we need to control the register state during the post-neutralization window.

Interrupt success As a first step, we test how frequently an interrupt happens *exactly* during execution of `saferet` (i.e., during the `lea` at offset `0x2` and `ret` at offset `0x7`). From our user-space program, we set up a timer as described in Section 5.1, and evict the instructions of `saferet` repeatedly using a sibling hyperthread from the L1 and L2 cache. For 10,000 iterations, we read the timer's expiration count, and perform different system calls. We again insert a kernel probe on the interrupt handler, and log only those interrupts coming from our process, filtering by instruction pointer. If the interrupt happened during `saferet`, we log this.

Register control From the system calls that triggered interrupts during `saferet`, we furthermore determine during which of these system calls the register state is controlled by our user space application. We thus pass easily recognizable values to the system call (e.g. `0xaaaa...`), and adjust the kernel probe logger to include the register state.

We find various system calls during which injected interrupts frequently trigger during `saferet`. While those that occur with an attacker-controlled register state are more rare, we can find several, especially when considering stack values.

Eventually, we end up picking the `write` system call, which gives us a particularly high frequency of interrupts during `saferet` execution. In particular, by calling it multiple times after reading our timer's expiration count, the chance of an interrupt during `saferet` with user-controlled registers is around 2%. Specifically, we fully control both `r13` and `r14`. The total fraction of interrupts during `saferet` is between 5% and 12%, if not considering register control.

Poisoning the RSB We install the PhantomJMP to the recursive PhantomCALLs as described in [31] such that the interrupt handler pollutes the RSB with an attacker chosen address. In particular, we insert the PhantomJMP at offset `0x123e` in the Linux kernel, which is right before the interrupt return (`iret`).

6.3 Breaking KASLR & leaking data

Like in previous work [28, 31, 36, 37], we break KASLR in multiple steps. First, we leak `_text` using a Prime+Probe attack. Second, we leak the physical address `R_phys` of our reload buffer `R` using Flush+Reload. Lastly, we find the physmap base using the leaked `R_phys`, again using a Flush+Reload attack. The gadgets for breaking KASLR are provided in Appendix A.

```
0x0 movzx  edx,BYTE PTR [r14]
0x4 mov    r12,QWORD PTR [r13+rdx*8+0x0]
```

```
0x0 movzx  ecx,WORD PTR [r14+0xb8]
0x8 mov    DWORD PTR [rbp-0x44],eax
0xb shr    r11w,0x6
0x10 sub   edx,r10d
0x13 and   r11d,0x1
0x17 mov   DWORD PTR [rbp-0x4c],r10d
0x1b movzx eax,WORD PTR [r13+rcx*1+0x6]
```

Listing 2: Gadgets to leak arbitrary data, found at offsets `0xb6c051` and `0x12e214c`. Registers `r13` and `r14` are attacker-controlled (in blue). The secret is depicted in red.

Linux places the kernel's code at one of 488 locations [20]. To leak the `_text`, we use a gadget that dereferences both registers under attacker control. We pass a guess of the kernel's `_text` to the `write` system call, and use Prime+Probe [25] to determine whether our guess was correct.

Given the leaked `_text` address, we leak a physical address of our program using a gadget that adds the direct map (physmap) base to `r13`, dereferencing the result. We then use Flush+Reload [39] to determine whether we guessed the physical address `R_phys` for our reload buffer `R` correctly. To reduce entropy, we allocate `R` as a transparent huge page.

Lastly, we derandomize the physmap base. Specifically, we pass a guess of the physmap base + `R_phys` to the system call so that it ends up in `r14`, using Flush+Reload to determine whether our guess was correct. Depending on the system configuration, the entropy of the physmap base is 15 [20].

After fully breaking KASLR, we proceed to leak arbitrary data. For this, we use the two gadgets shown in Listing 2. In the top gadget, the byte's low 3 bits fall within one cache line and are not recovered. We therefore use it to leak the upper 5 bits (3 to 7) of two consecutive bytes in memory. We then use the bottom gadget to leak the lower 3 bits of the second byte (0 to 2), which in the bottom gadget determines the cache line, allowing us to reconstruct the byte in full.

Results We evaluate our attack on our AMD Zen 2 machine, rebooting the system after each attempt. To deal with the low chance of success of our interrupt happening right after neutralization of the RSB, we parallelize all steps over the 8 available physical cores on our system. We successfully break KASLR in 10 of 10 attempts, in a median time of 548 seconds. Table 2 shows how long each step took. We then leak 4096 bytes of randomized data that we allocate in our kernel module. Our results show that we leak at an average rate of 5.47 bytes/s, with an average accuracy of 91.97%.

	Success rate	Median time (s)
_text	10/10	212.5
Physical address	10/10	131
Physmap base	10/10	90

Table 2: Evaluation of breaking KASLR using INTERRUPT INJECTION on AMD Zen 2.

6.4 Finding /etc/shadow

Now that we can leak arbitrary memory, we proceed to find /etc/shadow, which contains the root password hash. By attempting to log in using passwd, /etc/shadow will be placed in the page cache at a page-aligned address. That is, we will be searching through all 4KB aligned addresses, to determine if it starts with "root". We again parallelize the search over all 8 cores. To further speed up the search, we use a gadget (shown in Appendix A) that brings $R+g[31:0]$ into the cache, where $g[31:0]$ is the lower 32 bits of data at a guessed physical address g. This allows us to quickly test whether a physical address matches 32 - 6 (cache line offset) = 26 bits of root, instead of leaking the first bytes of each address.

Reducing the search space To reduce the search space, we leverage transparent huge pages. In particular, we allocate a large amount of memory pages and mark them with a known value. If the attacker leaks this value, they know they own this memory. Since our memory is backed by transparent huge pages, the attacker can skip searching an entire 2MB region. We find that we can reliably allocate between 5,000 and 6,000 transparent huge pages, which amounts to over 10GB of memory. At the start of the attack, we search through all memory to mark all physical addresses that are owned by us. This reduces the search space significantly.

Results We run our attack 10 times with an average execution time of 18 minutes, rebooting after each attempt. In 5 of the 10 cases, we successfully find the location of /etc/shadow, allowing us to leak its contents with the steps described in Section 6.3.

7 Discussion

In this section, we provide a brief discussion of alternative variants of INTERRUPT INJECTION and its improvements.

Improving INTERRUPT INJECTION for the BHB By using INTERRUPT INJECTION to manipulate the BHB on Intel CPUs, we observed mispredictions in our kernel module. This attack relies on an IBP index and tag match, influenced by the interrupt handler's execution path. In this paper, we do not consider the history produced by the interrupt handler to be under attacker control. However, predictor structures are known to be updated even speculatively [31, 35]. Therefore, an attacker may influence speculative behavior of the interrupt handler to have it produce a predictable BHB state. For example, conditional branches are known to be susceptible to Spectre-v1 [18], and update the BHB state [36].

Other variants of INTERRUPT INJECTION In addition to BHB-related mitigations and saferet, we believe other (older) mitigations to be also potentially vulnerable to INTERRUPT INJECTION. For example, retpoline turns all indirect jumps and calls into a return, and also neutralizes the RSB before executing this return. If an interrupt occurs during this post-neutralization window, an RSB underflow caused by the interrupt handler could in theory result in a misprediction (i.e., Figure 7-bottom). Likewise, RSB stuffing may be bypassed by an interrupt that experiences an RSB underflow. However, as discussed in Section 5.3, this requires the RSB to be circular. Furthermore, exploitation would be difficult due to the fact that the attacker has no control over the entries inserted in the RSB during the interrupt.

Triggering interrupts There are other approaches an attacker could take to force the CPU into an interrupt handler. For example, an attacker can flood the system with network packets, or generate a large amount of asynchronous I/O operations. While these methods are theoretically possible, we believe they are significantly harder to exploit in practice than timer-based injections due to the lack of control over interrupt timing, and other factors such as network jitter, disk latency, and hardware-level interrupt coalescing.

8 Mitigations

Re-neutralization A possible mitigation is to re-neutralize the potentially poisoned predictor upon interrupt return. To mitigate our attack targeting saferet, the RSB (or just the top-of-the-RSB) can be stuffed with benign entries right before the interrupt return instruction, if these instructions are guaranteed to be free from misprediction. Alternatively, the instruction pointer of the interrupted thread can be "rewound" to the sanitization step. It is also possible to disable saferet altogether and issue IBPB as entry-neutralization, but this comes at a hefty performance penalty as discussed in [31].

On Intel, re-neutralization is more challenging, since we do not rely on any particular BHB state, but rather on the interrupt producing BHB values that, combined with the branch instruction address, yield an IBP entry that was also used during training. In fact, re-neutralizing the BHB through e.g. the SW loop, as recommended by Intel for older CPUs [17], will likely increase the predictability of mispredictions rather than preventing them. Since the BHB state will be identical on each interrupt return, the same entry is selected in the IBP

regardless of prior history. On newer Intel CPUs (e.g. our Arrow Lake), the Indirect Branch History Fence (IBHF) can be called right before issuing the interrupt return instruction. According to its description, this should provide mitigation.

Preventing interrupts Another possible direction is to block the triggering of interrupt handlers during the post-neutralization window and postpone them until after the time of use. However, this raises a dilemma related to both how frequently the post-neutralization window is entered and the length of the post-neutralization window. In the case of in-place neutralization, such as `saferet`, this approach may appear plausible, as the post-neutralization window is usually short. Nevertheless, the performance penalty of disabling and re-enabling interrupts remains unclear, and they can typically only block non-maskable interrupts. This configuration needs to be applied every time execution enters and exits the post-neutralization window, which can occur at a very high frequency for in-place neutralization. In contrast, for entry neutralization, disallowing interrupts throughout the kernel execution (not just user-space timer interrupts, since other types of interrupts could in theory also be exploited as discussed in Section 7), seems to be infeasible as it would significantly hurt system responsiveness.

9 Related work

In this section, we discuss a subset of the Spectre-v2 attacks closely related to INTERRUPT INJECTION and TONTOU.

Inception [31] is a TONTOU attack, since it breaks entry neutralization by re-directing speculative control-flow using a PhantomJMP after neutralization to a recursive PhantomCALL which poisons the RSB. Inception can be mitigated by `saferet`, as every return is protected by a neutralization step immediately prior to its execution. INTERRUPT INJECTION revives Inception by achieving re-direction within the post-neutralization window and re-poisoning the neutralized RSB. This example illustrates that INTERRUPT INJECTION is flexible and can be combined with poisoning primitives.

Training Solo [36] breaks `SW loop` and `BHI_NO`, both of which provide entry neutralization for the BHB. Some of the demonstrated attacks can be classified as TONTOU attacks. In particular, to trigger mispredictions, re-direction is caused architecturally using cBPF filters which re-poison the predictor after neutralization. Other attacks demonstrated by [36] rely on lack of neutralization, and are not classified as TONTOU attacks. INTERRUPT INJECTION triggers re-direction at arbitrary points in time, allowing it to break in-place neutralization, which is unlike Training Solo, that relies on architectural re-direction by the kernel at defined moments in time.

Finally, there exist other approaches to achieve the re-direction step in TONTOU attacks other than interrupts. One example is to use exceptions, such as page faults, which are commonly used to increase the time resolution of timing side-channel attacks, for example in Microscope [30] and SGX-Step [33, 34]. An attacker can achieve relatively precise control over when an exception is triggered by selecting which page is configured to cause a page fault. However, page-table configuration is a privileged operation that user-space applications cannot perform. As a result, this approach is only suitable under stronger threat models, such as a privileged hypervisor or host OS attacking a trusted execution environment (TEE), e.g., Intel SGX [9, 16] or AMD SEV [1].

10 Conclusion

In this paper, we identify TONTOU, a class of attacks that exploits the unavoidable window between neutralization and usage of the branch predictor state. We discover a novel primitive to re-direct the architectural control flow during this post-neutralization window, INTERRUPT INJECTION, allowing us to re-poison the branch predictor state. We describe different variations, and demonstrate that mispredictions occur on both Intel and AMD CPUs, despite the latest mitigations. To prove the practicality of INTERRUPT INJECTION, we build an end-to-end exploit that leaks arbitrary kernel memory on AMD Zen 2, despite the latest neutralization techniques deployed in the Linux kernel. We anticipate that our findings will motivate further research into the attack surfaces of INTERRUPT INJECTION and TONTOU, and facilitate the development of more robust mitigations.

Acknowledgements

We thank our anonymous reviewers for their helpful feedback. We also thank Ben Gras for valuable discussions throughout this project. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-23-F-0014 in the amount of $196,173. This award was made through the Department of Defense (DoD) National Defense Science & Engineering Graduate (NDSEG) Fellowship Program. This work was also supported by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Ethical Considerations

Stakeholder analysis We identify the following groups impacted by this research:

A. CPU vendors. INTERRUPT INJECTION exploits microarchitectural behavior designed by CPU vendors. As such, this research requires CPU vendors to revisit the security guarantees of their products and to evaluate and deploy mitigations to users.

B. Linux kernel developers. Affected mitigations (`saferet`, `SW loop`, `BHI_DIS_S`) are implemented and/or initiated in kernel code. Kernel maintainers must implement and review software mitigations, and evaluate their effectiveness.

C. Cloud and virtualization providers. Cloud providers deploy affected processors in multi-tenant environments, where a co-located attacker could leak data across tenants. Hypervisors and systems relying on kernel-based isolation may similarly be vulnerable. This research requires these parties to deploy patches and audit potential impacts of TONTOU and INTERRUPT INJECTION.

D. End users. Individuals and organizations may be impacted by this research and its publication since their sensitive data is at stake, although they are to some extent affected by the underlying vulnerability regardless.

E. Security research community. The broader security research community engages with published vulnerability research by using it to identify further variants and to develop stronger mitigations.

Impact of research
Positive impacts:

1. It facilitates the development of more robust mitigations in hardware (**A**) or in software, e.g. the Linux kernel (**B**). This enables protection of post-neutralization windows in the cloud (**C**), and directly improves protection of end user data (**D**).

Negative impacts:

1. It creates a risk of information leakage during disclosure periods that enables an adversary to exploit TONTOU and INTERRUPT INJECTION, impacting most stakeholders (**A**, **B**, **C**, **D**).

Impact of publication
Positive impacts:

1. It is a motivating factor for CPU vendors (**A**) to improve the security of their products, benefiting the Linux team (**B**), cloud providers (**C**) and end users (**D**).
2. It facilitates future attack research into Spectre v2 by the security research community (**E**).
3. It enables academics to propose more informed defenses against Spectre v2 attacks (**E**).

Negative impacts:

1. It simplifies reproduction of TONTOU and INTERRUPT INJECTION attacks by an adversary (**A**, **B**, **C**, **D**).

Research justification We believe the positive impacts of this research outweigh its negative impacts. INTERRUPT INJECTION may independently be discovered by adversaries and exploited to leak sensitive data, since the vulnerability already exists in widely deployed systems. Responsible research into TONTOU and INTERRUPT INJECTION enables more robust defenses that prevent or stop such scenarios. Experiments were conducted on local systems, without accessing or affecting real-world user data or third-party systems. To further reduce risk, details of the research were shared only with a limited set of affected parties, and were otherwise kept confidential.

Decision to publish We also believe publication is warranted. Without publication, the TONTOU vulnerability class and the INTERRUPT INJECTION attacker primitive may remain unknown to the public, preventing independent verification and increasing the risk that vulnerabilities remain or re-appear. Publication also enables future academic work to expand on this research and identify new weaknesses, which again improves security for end users. Lastly, it allows academics to propose better informed defenses against Spectre v2 attacks, benefiting all stakeholders. Negative impacts are mitigated by informing CPU vendors well in advance of publication (AMD and Intel were notified in early February). Importantly, INTERRUPT INJECTION is addressable through software patches, enabling vendors to complete patches by the time of publication.

Responsible disclosure In addition to disclosing our research to CPU vendors, we contacted the Linux kernel maintainer for the stable branch on March 3rd, 2026. AMD asked our consent to share the research with downstream customers, including cloud providers, under NDA. We therefore aligned with AMD's disclosure process to avoid duplicate disclosures that could increase the risk of unintended exposure.

Open Science

In this Section, we provide an explanation of how our artifacts can be used to reproduce the results presented in this paper. We confirm our results to be reproducible with the following configurations:

1. Intel Cascade Lake Refresh (Xeon Gold 5220R), Linux version `5.15.0-168-generic`
2. Intel Arrow Lake (Ultra 9 285H), Linux version `6.14.0-37-generic`
3. AMD Zen 2 (Ryzen 7 4700G), Linux version `6.14.0-37-generic`
4. AMD Zen 4 (EPYC 9124), Linux version `6.14.0-37-generic`

Interrupt & system call alignment Run `./a.sh` to produce the results presented in Section 5.1. As an argument, pass one of the machines (`-cascade`, `-arrow`, `-zen2`, or `-zen4`). The script builds and inserts a kernel module, which registers a kernel probe on the timer interrupt path, counting each time an interrupt occurs during `ioctl`. It then performs the grid search of the different interval and busy loop iterations, calling the attacker binary, and lists the output by grepping `dmesg`. The results produce the heatmap shown in Figure 4.

Interrupt & post-neutralization window alignment Run `./b.sh` to validate our results in Section 5.2. The user should again pass the CPU under test as argument. It again inserts the kernel module, which has a short window of NOP instructions followed by a return. It translates the virtual address of this window to a physical address, outputting it to `dmesg`. The script uses this to compile a workload. The number of interrupts is again counted using a kernel probe, allowing its output to be retrieved from `dmesg` after calling the attacker binary, which simply sets the timer and performs the system calls.

Training using an interrupt Run `./c.sh` to validate our results in Section 5.3, passing the CPU under test as argument. A kernel module is installed which will ensure a shared pointer between user space and the kernel. On Intel CPUs, the attacker binary sets up the timer, and iteratively calls the kernel module to perform a training run and victim run. For AMD CPUs, the attacker binary sets up a timer, installs the PhantomJMP and recursive PhantomCALLs, and calls the kernel module. The attacker binary uses Flush+Reload to determine the number of mispredictions, outputted to standard output.

End-to-end attack To validate our results presented in Section 6, the user should first run `./attack/kaslr.sh` on an AMD Zen 2 machine with transparent huge pages enabled. This outputs the `_text` address, the physical address of its reload buffer, and the `physmap` address. The user can then run `./attack/leak.sh`, passing both the `_text` and `physmap` as arguments, and a physical address of interest to leak. The script will leak 4096 bytes to `leak.txt`. To find `/etc/shadow`, the user can run `./attack/shadow.sh`, passing both `_text` and `physmap` as arguments.

Availability The artifacts of this paper can be found at https://github.com/MATCHA-MIT/TONTOU-Interrupt-Injection.

References

[1] Advanced Micro Devices, Inc. Amd sev secure nested paging, 2022. [Online]. Available: https://www.amd.com/system/files/TechDocs/56860.pdf.

[2] AMD. The linux kernel user's and administrator's guide: Spectre side channels. [Online]. Available: https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst.

[3] AMD. Technical guidance for mitigating branch type confusion. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/technical-guidance-for-mitigating-branch-type-confusion.pdf.

[4] AMD. Software techniques for managing speculation on amd processors, 2022. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/software-techniques-for-managing-speculation.pdf.

[5] AMD. Software optimization guide for the amd zen4 microarchitecture, 2023. Accessed on 05.11.2025.

[6] Andy Nguyen, Anthony Weems, Matteo Rizzo, Alexandra Sandulescu. Novel inception/srso exploitation method, 2024. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/cpus/inception/README.md.

[7] Alexis Bagia, Vincent Quentin Ulitzsch, Daniël Trujillo, Mengyuan Li, Mengjia Yan, and Jean-Pierre Seifert. A close look at rmp entry caching and its security implications in sev-snp. In *Proceedings of the 14th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10–18, 2025.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, 2022.

[9] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[10] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux symposium*, volume 1, pages 333–346, 2006.

[11] B Gras, KAVEH Razavi, H Bos, and C Giuffrida. Tlbleed: When protecting your cpu caches is not enough. *Black Hat*, 2018.

[12] Linux Kernel Admin Guide. Speculative return stack overflow (srso), 2023. [Online]. Available: https://docs.kernel.org/admin-guide/hw-vuln/srso.html.

[13] Intel Corporation. Indirect branch restricted speculation, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[14] Intel Corporation. Speculative execution side channel mitigations, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[15] Intel Corporation. Post-barrier return stack buffer predictions / cve-2022-26373 / intel-sa-00706, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/post-barrier-return-stack-buffer-predictions.html.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2023. Volume 3D: Intel Software Guard Extensions (SGX).

[17] Intel Corporation. Branch history injection and intra-mode branch target injection / cve-2022-0001, cve-2022-0002 / intel-sa-00598, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[19] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[20] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–321. IEEE, 2020.

[21] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. Indirector:{High-Precision} branch target injection attacks exploiting the indirect branch predictor. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2137–2154, 2024.

[22] Linux. timerfd.c. [Online]. Available: https://android.googlesource.com/kernel/common/+/ecf61e4e1117/fs/timerfd.c.

[23] Alyssa Milburn, Ke Sun, and Henrique Kawakami. You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection. *arXiv preprint arXiv:2203.04277*, 2022.

[24] Alex Murray. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. [Online]. Available: https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047.

[25] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[26] Kim Phillips. [patch 0/3] x86/speculation: Support automatic ibrs, 2022. [Online]. Available: https://lore.kernel.org/lkml/20221104213651.141057-1-kim.phillips@amd.com/T/.

[27] Filip Pizlo. What spectre and meltdown mean for webkit, 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[28] Sandro Rüegge, Johannes Wikner, and Kaveh Razavi. Branch privilege injection: Compromising spectre v2 hardware mitigations by exploiting branch predictor race conditions. USENIX Security Symposium, 2025, August 2025. [Online]. Available: https://comsec.ethz.ch/wp-content/files/bprc_sec25.pdf.

[29] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: Leaking data on meltdown-resistant cpus (updated and extended version). *arXiv preprint arXiv:1905.05725*, 2019.

[30] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 318–331, 2019.

[31] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, 2023.

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

[33] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[34] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[35] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *USENIX Security*, 2024.

[36] Sander Wiebing and Cristiano Giuffrida. Training solo: On the limitations of domain isolation against spectre-v2 attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3599–3616. IEEE Computer Society, 2025.

[37] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, 2022.

[38] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 49–61, 2023.

[39] Yuval Yarom and Katrina Falkner. {FLUSH+RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[40] Peter Zijlstra and Thomas Gleixner. [patch v3 00/59] x86/retbleed: Call depth tracking mitigation, 2022. [Online]. Available: https://lkml.org/lkml/2022/9/15/427.

```
0x0 mov     rcx,QWORD PTR [r14+0x8]
0x4 mov     rax,QWORD PTR [r13+0x8]
```

```
0x0 add     r13,QWORD PTR [rip+0x188442e]
0x7 mov     rbx,QWORD PTR [r13+0x0]
```

Listing 3: Gadgets to break KASLR, found at offsets `0x14aac0f` and `0x4d7233`.

```
0x0 mov     edx,DWORD PTR [r14+0xc0]
0x7 mov     rdx,QWORD PTR [r13+rdx*1+0x8]
```

Listing 4: Gadget to find `/etc/shadow`, found at offset `0x12e2361`.

A End-to-end attack gadgets

In addition to the gadgets used for leaking arbitrary kernel memory (Listing 2), we use a number of additional gadgets to break KASLR and find `/etc/shadow`.

KASLR For finding `_text`, we pass two addresses in the guessed kernel location falling in two different cache lines in `r13` and `r14`, allowing us to detect a correct guess with Prime+Probe (Listing 3-top). We then pass a guess of the physical address of our reload buffer in `r13`, allowing us to detect a correct guess using Flush+Reload (Listing 3-bottom). Lastly, we find the physmap base by passing our leaked physical address added to a physmap guess in `r14`, allowing us to detect a correct guess using Flush+Reload (Listing 3-top).

`/etc/shadow` To find the hashed root password, we use the gadget shown in Listing 4. We pass a guess of its location in `r14`, and we pass the kernel address of our reload buffer, minus "root". If the contents at the address in `r14` are "root", this will fetch our reload buffer, which we detect with Flush+Reload. We use Listing 2-bottom to confirm the signal.