GhostRace: Exploiting and Mitigating Speculative Race Conditions

Hany Ragab, *Vrije Universiteit Amsterdam;* Andrea Mambretti and Anil Kurmus, *IBM Research Europe - Zurich;* Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity24/presentation/ragab

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

GhostRace: Exploiting and Mitigating Speculative Race Conditions

Hany Ragab[†*] hany.ragab@vu.nl
Andrea Mambretti[*] amb@zurich.ibm.com
Anil Kurmus[*] kur@zurich.ibm.com
Cristiano Giuffrida[†] giuffrida@cs.vu.nl

[†]Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

[*]IBM Research Europe
Zurich, Switzerland

Abstract

Race conditions arise when multiple threads attempt to access a shared resource without proper synchronization, often leading to vulnerabilities such as concurrent use-after-free. To mitigate their occurrence, operating systems rely on synchronization primitives such as mutexes, spinlocks, etc.

In this paper, we present GhostRace, the first security analysis of these primitives on *speculatively executed* code paths. Our key finding is that *all* the common synchronization primitives can be microarchitecturally bypassed on speculative paths, turning *all* architecturally race-free critical regions into *Speculative Race Conditions (SRCs)*. To study the severity of SRCs, we focus on Speculative Concurrent Use-After-Free (SCUAF) and uncover 1,283 potentially exploitable gadgets in the Linux kernel. Moreover, we demonstrate that SCUAF information disclosure attacks against the kernel are not only practical, but that their reliability can closely match that of traditional Spectre attacks, with our proof of concept leaking kernel memory at 12 KB/s. Crucially, we develop a new technique to create an unbounded race window, accommodating an *arbitrary* number of SCUAF invocations required by an end-to-end attack in a *single* race window. To address the new attack surface, we also propose a generic SRC mitigation to harden all the affected synchronization primitives on Linux. Our mitigation requires minimal kernel changes and incurs only ≈5% geomean performance overhead on LMBench.

"There's security, and then there's just being ridiculous."
– Linus Torvalds, on Speculative Race Conditions

1 Introduction

Since the discovery of Spectre [45], security researchers have been scrambling to locate all the exploitable snippets or *gadgets* in victim software. Particularly insidious is the first Spectre variant (exploiting conditional branch misprediction [45]), since any victim code path guarded by a source `if` statement may result in a gadget. To identify practical Spectre-v1 gadgets, previous research has focused on speculative memory safety vulnerabilities [42,45,53], use-after-free [41], and type confusion [41]. However, much less attention has been devoted to other classes of (normally architectural) software bugs, such as *concurrency bugs*.

To avoid (or at least reduce) concurrency bugs, modern operating systems allow threads to safely access shared memory by means of synchronization primitives, such as mutexes and spinlocks. In the absence of such primitives, e.g., due to a software bug, *critical regions* would not be properly guarded to enforce mutual exclusion and *race conditions* would arise. While much prior work has focused on characterizing and facilitating the architectural exploitation of race conditions [49], very little is known about their prevalence on transiently executed code paths. To shed light on the matter, in this paper we ask the following research questions:

"*How do synchronization primitives behave during speculative execution? And what are the security implications for modern operating systems?*"

To answer these questions, we analyze the implementation of common synchronization primitives in the Linux kernel. Our key finding is that *all* the common (write-side) primitives (i) lack explicit serialization and (ii) guard the critical region with a conditional branch. As a result, in an adversarial speculative execution environment, i.e., with a Spectre attacker mistraining the conditional branch, these primitives essentially behave like a *no-op*. The security implications are significant, as an attacker can speculatively execute *all* the critical regions in victim software with *no* synchronization.

Building on this finding, we present GhostRace, the first systematic analysis of Speculative Race Conditions (SRCs), a new class of speculative execution vulnerabilities affecting all common synchronization primitives. SRCs are pervasive, as an attacker can turn arbitrary (architecturally) race-free code into race conditions exploitable on a speculative path—

[*]This work was partially done at IBM Research, Zurich, Switzerland

in fact, one originating from the synchronization primitives' conditional branch itself. While the effects of SRCs are not visible at the architectural level (e.g., no crashes or deadlocks), due to the transient nature of speculative execution, a Spectre attacker can still observe their microarchitectural effects via side channels. As result, any SRC breaking security invariants can ultimately lead to Spectre gadgets disclosing victim data to the attacker. To investigate the practical security impact of SRCs, we focus on Speculative Concurrent Use-After-Free (SCUAF), a subclass of speculative race conditions which concerns all critical regions which, once speculatively executed, can expose Use-After-Free (UAF) vulnerabilities.

To investigate the resulting attack surface, we first present new techniques to exploit SCUAF in practice. SCUAF exploitation carries with it all the exploitation challenges of Spectre as well as those of architectural race conditions. The latter alone is far from trivial, as reliable exploitation relies on *controlling* and *stretching* the race window to fit the exploit [49]. Nonetheless, architectural exploits are typically *one-shot*, i.e., with a single iteration of a (e.g., UAF) primitive and thus a single (successful) race window. Spectre exploits, in turn, typically require thousands or millions of iterations to scan memory looking for the secret to leak [27]. As such, with existing race window-stretching techniques [49], we would need to win the race an overwhelming number of times, hindering practical exploitation. To address this challenge, we propose a new (architectural or speculative) UAF race window-massaging technique to (i) surgically interrupt the victim thread at the "*right time*" and (ii) create an *unbounded* window. Our technique builds on and extends existing timer interrupt-based techniques [81] to not only make SCUAF attacks realistic, but accommodate a full end-to-end speculative information disclosure attack in a *single* race window.

Second, to investigate the extent of the problem, we present a SCUAF gadget scanner and apply it to the Linux kernel to find 1,283 potentially vulnerable gadgets. By manually reaching one of our identified (device-specific) gadgets, we implement a Proof of Concept (PoC) which triggers a concurrent UAF on a speculative path to hijack the control flow to disclosure gadget in the kernel, allowing an unprivileged attacker to leak arbitrary kernel memory at the rate of 12 KB/s.

Finally, we present a mitigation to serialize the execution of all the vulnerable synchronization primitives. Our mitigation has general applicability and can completely close the attack surface of not only SCUAF but SRC in general. Moreover, it requires minimal kernel changes and incurs low performance overhead (≈5% geomean on LMBench).

Contributions. We make the following contributions:

1. We present a new exploitation technique to precisely interrupt any (kernel) thread and create an architecturally unbounded UAF exploitation window (Section 4.1).
2. We present Speculative Race Conditions (SRCs), a new class of speculative execution vulnerabilities affecting all common synchronization primitives (Section 4.2).
3. We study the security implications of SRCs on architecturally race-free critical regions in the Linux kernel, demonstrating a Proof of Concept exploiting a Speculative Concurrent UAF (SCUAF) and leaking arbitrary kernel memory at a rate of 12 KB/s (Section 4.3).
4. We propose a gadget scanner to find 1,283 potentially vulnerable SCUAF gadgets in the kernel (Section 5).
5. We propose a generic mitigation to harden synchronization primitives against SRC, with a ≈5% geomean performance overhead on LMBench (Section 6).

The PoC code, the gadget scanner, and additional information are publicly available at https://www.vusec.net/projects/ghostrace.

2 Background

2.1 Transient Execution

```
if (x < array1_size){
    y = array2[ array1[x] * 0x1000 ];
}
```

Figure 1: An example of Spectre bounds check bypass. The conditional branch (in red) with the attacker-controlled `x` speculatively bypassing the comparison with `array1_size`, the first speculative load reading a secret byte at address `(array1 + x)` (in blue), and the second speculative load referencing the CPU cache with a secret-dependent address (in orange).

Since 2018, after the discovery of Spectre [45] and Meltdown [50], transient execution attacks have become an intensively studied area of research. Whenever a modern CPU implements speculative optimizations (e.g., branch prediction), it speculatively executes a sequence of instructions. The two possible outcome for these instructions are that either they are committed and made visible to the architectural level or they are squashed due to mispeculation (e.g., misprediction)—leading to transient execution. When the instructions are squashed, the CPU rollbacks the state. Despite the rollback, some microarchitectural side effects are left and can be observed through one of the many side channels available (e.g., data cache [21,29,76,77], branch target buffer [52], port contention [12], etc.) to leak sensitive information.

Spectre-PHT, also known as Spectre-v1, is the first known attack of this kind, targeting the pattern history table and exploiting a code pattern such as the one shown in Figure 1. As shown in the figure, the code checks for x to be in-bound

before performing a double array access. For exploitation purposes, the attacker can ensure *x* is out-of-bound and *array1_size* is not present in the cache. In this scenario, instead of waiting for *array1_size* to be loaded from main memory to perform the comparison, the CPU speculates and starts to transiently execute the instructions beyond the comparison. If the comparison has been executed several times before with *x* in-bound, the CPU is prone to speculate that *x* is once again in-bound, hence transiently performing the out-of-bound access of *array1*. When the not cached *array2* is accessed using the byte retrieved from the out-of-bound access of *array1*, the specific accessed location is loaded into the cache. The attacker can complete the 1 byte leak by testing which location of *array2* can be accessed faster than the others. Its position within the buffer reveals the secret byte value.

Notably, Spectre-PHT remains unmitigated in hardware. Software developers remain responsible to harden potentially vulnerable branches with mitigations (e.g., fencing to prevent speculation), but the extent to which all the "*right*" branches have been adequately hardened in large high-value codebases such as the Linux kernel remains an open question.

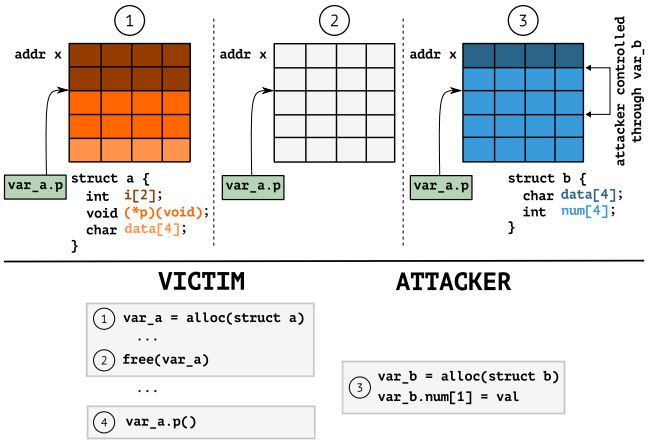

Figure 2: Object reallocation in a UAF attack. First, the victim allocates a heap object which is subsequently freed (steps *1* and *2*). Then, the attacker forces the allocation of an object (*var_b*) reusing memory slot of the victim's object, taking control of the data referenced by the dangling pointer (step *3*). Finally, the dangling pointer is dereferenced, ultimately allowing the attacker to hijack control flow (step *4*).

2.2 Concurrency Bugs

Concurrency bugs are a category of bugs which affect multithreaded programs and occur due to the absence or the incorrect use of synchronization primitives. Due to their nondeterministic behavior, concurrency bugs are one of the most elusive and difficult to triage classes of bugs. Under certain conditions, concurrency bugs can also lead to memory error vulnerabilities. In modern operating systems such as the Linux kernel, one of the most common memory error vulnerability caused by concurrency bugs is Use-After-Free (UAF).

In a UAF attack, the first step is generally to free a memory object. This operation invalidates all the pointers to that object, which become *dangling*. The second step generally involves forcing the allocator to reuse the memory slot of the free object for the allocation of a new object. This step reinitialize the previously freed memory slot. The final step of the attack is generally to force the victim to use one of the dangling pointers, which now points to the newly allocated object. A read from or write to such pointer to controlled data can be used to exploit the bug in a variety of ways. An example illustrated in Figure 2 is for instance to mount a control-flow hijacking attack via a dangling function pointer.

When this attack is performed in concurrency settings, and the free step and the use step are executed by distinct threads sharing the underlying object. Such concurrent use-after-free vulnerability is harder to exploit than the single-threaded UAF case, since exploitation depends on thread interleaving and the availability of a sufficient *race window* [49]. While the community has invested significant effort in investigating traditional concurrency bugs and concurrent UAF—e.g., studies demonstrating that more than 40% of the UAF vulnerabilities patched in Linux kernel drivers are concurrent UAF [9]—their microarchitectural properties have largely been neglected. In this paper, we study such properties and their security implications for the first time, uncovering a new class of speculative execution vulnerabilities in the process.

3 Definitions and Threat Model

3.1 Definitions

A traditional *data race* entails two threads accessing the same memory location, with one thread performing a write and no synchronization primitive protecting the shared accesses. The data race is referred to as a *race condition* when it impacts the correctness of the program. We define that a *Speculative Race Condition* (SRC) occurs when two threads access the same memory location, with one thread performing an architectural write operation and another a transient access, with an impact on the correctness of the speculatively executed program. Intuitively, the synchronization primitive, such as exclusive locking, can be bypassed by one of the two threads due to speculative execution. Due to their security impact, we specifically focus on *Speculative Concurrent Use-After-Free* (SCUAF), the SRC equivalent of traditional concurrent use-after-free [49]. Intuitively, because concurrent use-after-frees are often escalated to control-flow hijacking [49], SCUAFs are also likely to be escalated to speculatively control-flow hijacking, a powerful speculative execution primitive [45].

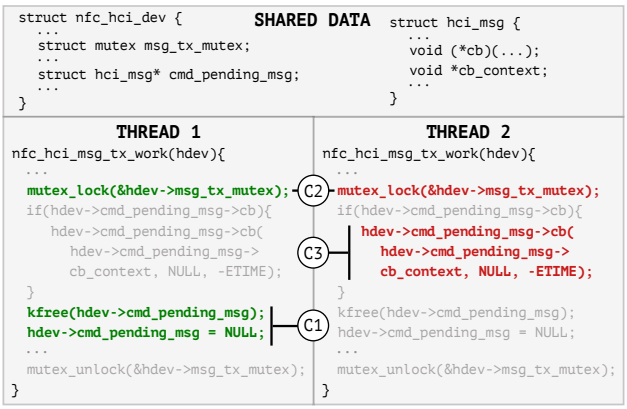

Figure 3: The NFC gadget (`net/nfc/hci/core.c:78`) found by our scanner and the three main challenges to mount an end-to-end GhostRace attack.

3.2 Threat Model

We consider a typical cross-domain Spectre threat model, with a local unprivileged attacker able to issue system calls to the victim kernel. The attacker seeks to leak arbitrary kernel data by exploiting a speculative race condition in an otherwise architecturally-race-free critical region in the kernel. We assume state-of-the-art mitigations against transient execution attacks are all enabled and other classes of (e.g., software) bugs are out of scope—for instance, subject to orthogonal mitigations. Hereafter, without loss of generality, we specifically focus on the Linux kernel running on Intel x86-64.

4 GhostRace Attacks

The goal of a GhostRace attack is to disclose arbitrary kernel data by exploiting a speculative race condition in an otherwise architecturally race-free critical region. To illustrate the workings of an attack, we use one of the gadgets found in Linux kernel v5.15.83 by our gadget scanner (`net/nfc/hci/core.c:78`) as a running example. Figure 3 depicts two threads both executing the gadget, at the core of the `nfc_hci_msg_tx_work` function. Such function serves as part of the implementation of the Host Controller Interface (HCI) layer of the Near Field Communication (NFC) driver core of the Linux kernel and processes pending messages to the NFC device. As we do not have the required NFC hardware to natively execute this function, we added a system call to reach this code path during our analysis. The function contains a critical region (i.e., our gadget) operating on the `nfc_hci_dev hdev` device and performing the following operations. *First*, it locks the `msg_tx_mutex` mutex to gain exclusive access to the device and enter the critical region. *Second*, it checks whether the pending `hci_msg hdev->cmd_pending_msg` command message to process has a callback set. If so, the callback is invoked via the `hdev->cmd_pending_msg->cb` function pointer. *Third*, it frees the memory backing the command message and sets the now-dangling `hdev->cmd_pending_msg` pointer to `NULL`. *Lastly*, it exits the critical region by releasing the mutex.

Since the critical region can be concurrently accessed by different user processes/threads sharing the NFC device, it is crucial for the mutex to guard the region, enforcing mutually exclusive access to the device and ruling out any race conditions. Indeed, absent the mutex, the code in question would be vulnerable to a concurrent Use-After-Free (UAF) vulnerability: as *Thread 1* executes *Free* code (in bold green), between the `kfree` of the `hdev->cmd_pending_msg` pointer and the `NULL` update of the pointer, *Thread 2* may execute the *Use* code (in bold red) and invoke the `hdev->cmd_pending_msg->cb` callback of the pending message which was just freed. An attacker able to control memory reuse can then trigger the *Use* with a controlled callback value and escalate the vulnerability to control-flow hijacking. Luckily, thanks to mutexes and other synchronization primitives offered by modern operating systems such as the Linux kernel (spin locks, RW locks, etc.), architectural exploitation of such race-free code is infeasible.

Unfortunately, as we will show, the same does not apply in the speculative domain, where architecturally race-free execution can still be (microarchitecturally) subject to SRCs. Specifically, in our GhostRace attack, we exploit the NFC gadget to craft a SCUAF primitive and ultimately disclose data, with *Thread 1* architecturally executing its critical region, i.e., architectural *Free*, and *Thread 2* concurrently being speculatively executed i.e., speculative *Use*. Moreover, we need to ensure that *Thread 1* is interrupted immediately after the *Free*, with a sufficiently large *race window* (or UAF exploitation window) for practical exploitation. Finally, we need to effect our SCUAF primitive several times to mount end-to-end speculative information disclosure attacks.

In other words, to mount GhostRace attacks, we need to address the following challenges highlighted in Figure 3:

- (C1) Create a large, ideally unbounded, architectural UAF exploitation window between `kfree` and the `NULL` `hdev->cmd_pending_msg` pointer update to accommodate as many SCUAF primitive invocations as possible.
- (C2) Turn our architecturally race-free gadget into a speculative race condition, crafting a SCUAF primitive speculatively dereferencing the (dangling) `hdev->cmd_pending_msg->cb` function pointer.
- (C3) Use the building blocks above to mount end-to-end information disclosure attacks against the kernel.

4.1 Creating an Unbounded UAF Window

To address (C1), we need a strategy to interrupt an arbitrary thread in the Linux kernel for a large and ideally unbounded

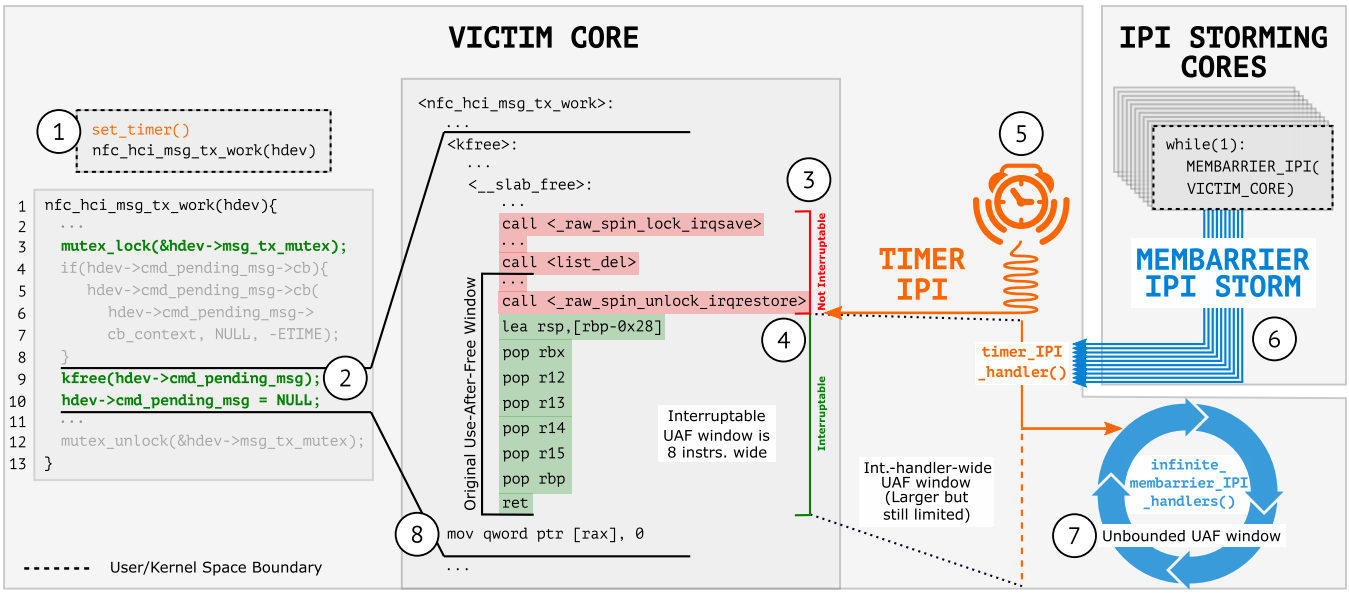

Figure 4: From eight instructions-wide to **unbounded** architectural Use-After-Free exploitation window. Steps 1 and 6 run in user mode, issuing syscalls to trigger the relevant kernel code. The other steps run in kernel mode.

period of time. We can then use this strategy to create an architectural unbounded UAF window in the *victim Thread 1* for the *attacker Thread 2* to exploit. Our case study is particularly challenging as the "original" UAF exploitation window is very small: the attacker must be able to use the `hdev->cmd_pending_msg` pointer (at line 5 in *Thread 2*) after the `hci_msg` memory object is freed (line 9 in *Thread 1*) and before the pointer is nullified (line 10 in *Thread 1*). In other words, the time between `kfree` freeing the object and the `NULL` update of the `hdev->cmd_pending_msg` pointer is the only span *Thread 2* could exploit to craft a UAF primitive.

A Tiny Window. To precisely quantify the original UAF exploitation window, one can inspect the implementation of `kfree`. As shown in Figure 4, under the hood, the default (`__slab_free`) implementation frees the object (`list_del`) and then releases an interrupt-safe spinlock (`_raw_spin_unlock_irqrestore`) immediately before returning control to the caller. Since such spinlock runs with interrupts disabled (i.e., the CPU cannot be interrupted), the original (interruptible) UAF exploitation window is as tiny as *eight instructions*—accounting for the time between spinlock release and the `NULL` pointer update at line 10. To stretch such a tiny window, we can build on existing interrupt-driven techniques [49, 81]. Nonetheless, this is challenging as such techniques were not designed to produce race windows that would reliably accommodate several UAF invocations. Moreover, other techniques relying on a high-priority user thread to preempt the victim kernel execution [34] are not applicable to stock kernels, which run with preemption off by default (i.e., `CONFIG_PREEMPT` unset).

From Tiny to Unbounded. To address these issues, we propose a new strategy based on a combination of techniques. First, drawing from the *timerfd*-based technique proposed in [81], we rely on *high-precision hardware timers* [26] to interrupt the victim thread at the right time and slightly amplify the original UAF window. Note that, in our setting, the original *timerfd*-based technique [81] becomes more effective, since we can exploit the interrupt-disabled behavior of `kfree` to more precisely interrupt the victim thread at the right time. Second, we rely on user interfaces to trigger an (interprocessor) interrupt (*IPI*) *storm* to (less precisely) interrupt the victim thread in the *amplified window* and stretch such window indefinitely. This is possible since the victim CPU is stuck handling IPIs until the attacker so wishes. Figure 4 illustrates the steps of our strategy.

As shown in the figure, the attack starts with ① the attacker scheduling a high-precision hardware timer [26] on a victim core. The attacker calibrates the timer to expire at some point in the future, at nanosecond resolution. Next, ② the attacker starts a victim (i.e., *Free*) thread on the same core, which issues a system call to reach the target gadget and thus the victim `kfree` invocation. Next, ③ `kfree` acquires the interrupt-safe spinlock and completes the freeing of the victim memory object within uninterruptible execution. Next, ④ `kfree` releases the spinlock and resumes interruptible execution. At this point, as long as the timer already expired during uninterruptible execution, ⑤ the victim gets immediately interrupted at the start of the interruptible UAF window. Indeed, when the timer expires, the hardware raises an interrupt, but its actual delivery gets delayed until interrupts are enabled again (i.e., upon spinlock release). In other words,

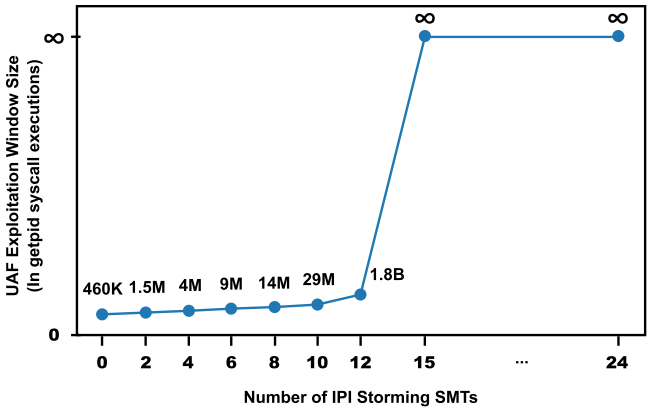

Figure 5: Size of the UAF exploitation window vs. number of IPI storming cores targeting the victim core. The size of the exploitation window is measured in number of `getpid` syscalls (a standard benchmark to evaluate generic round trips to the kernel [11]) that attackers can run before the victim core handles all incoming `membarrier` IPIs and updates the dangling `hdev->cmd_pending_msg` pointer to `NULL`. The experiment is performed on a commodity client Intel 12th-generation i9-12900K CPU, which has 16 cores and 24 Simultaneous Multithreads (SMTs). We observe that only 15 SMTs are sufficient to obtain an unbounded UAF exploitation window. We also observe that the location of the IPI storming cores matters [23, 57], as the physically closer a storming core is to the victim core, the higher the IPI throughput due to the lower latency on the interconnect. This explains the big increase from 10 to 12 and from 12 to 15 SMTs as the storming cores added in both experiments were physically the closest to the victim core among all available cores.

even with imprecise timer calibration or jitter, the attacker has significant chances to interrupt the victim at precisely the right time. It is, in fact, sufficient to cause the timer to expire *any time* within the core (uninterruptible) execution of `kfree`, rather than in the original tiny eight-instruction UAF window. Put differently, the interrupt-disabled behavior of `kfree` ultimately *helps* rather than hinders the attack.

When the timer interrupt gets delivered, kernel execution switches from the victim thread to the timer interrupt handler. The latter normally is short-lived, but one can amplify the window by registering several timer observers, e.g., via `timerfd` support. Still, while this strategy can interrupt the victim thread at the right time and amplify the original UAF window, the latter is still insufficient to accommodate many SCUAF primitive invocations. However, the amplified window *is* sufficient for the attacker to interrupt the timer interrupt handler with a more jittery Inter-Processor Interrupt (IPI) sent by another core. Building on this intuition, ⑥ the attacker schedules on the remaining cores *storming* threads that constantly send IPIs to the victim core. For this purpose, the `membarrier` system call

```
1 void mutex_lock(struct mutex *lock){
2   ...
3   if (!__mutex_trylock_fast(lock))
4     if (atomic_long_try_cmpxchg_acquire(&lock, ...))
5       return true;
6   ...
7 }

  Call Stack:
  atomic_long_try_cmpxchg_acquire(&lock, ...)
   └arch_atomic_long_try_cmpxchg_acquire(&lock, , ...)
     └arch_atomic_try_cmpxchg_acquire(&lock, , ...)
       └arch_atomic_try_cmpxchg(&lock, , ...)
         └arch_try_cmpxchg((&lock, , ...)
           └__raw_try_cmpxchg(ptr, ...)({
               asm volatile(
                    "lock cmpxchgq %2, %1"
                    : "=a" (ret), "+m" (*ptr)
                    : "r" (new), "0" (old)
                    : "memory"
                    );
           })
```

Figure 6: Top part: The core implementation of the `mutex_lock` synchronization primitive, with the conditional branch which can be abused to craft SRCs in red. Bottom part: The branch ultimately checks the outcome of the `lock cmpxchgq` instruction which does not serialize the execution.

`MEMBARRIER_CMD_PRIVATE_EXPEDITED_RSEQ` IPI is ideal, since, unlike other IPIs used by previous work [49], its delivery can be triggered via a low-latency system call targeting a single (victim) core. The resulting IPI storm not only causes the timer interrupt handler to be interrupted, but completely overwhelms the victim core. Figure 5 relates the size of the UAF exploitation window to an increasing number of storming cores on our test platform, with 15 SMTs being sufficient to overwhelm the victim. Indeed, ⑦ the victim core is forced to constantly handle an indefinite number of `MEMBARRIER` IPIs, effectively creating an architectural unbounded UAF exploitation window to mount an arbitrarily long end-to-end attack. Finally, once the attack completes, the attacker terminates the storming threads, ⑧ victim thread execution resumes, and only then the dangling `hdev->cmd_pending_msg` pointer is updated to `NULL`. Note that, between steps ⑦ and ⑧, the attacker may execute a speculative execution attack as many times as they wish, given the unbounded window. This means the steps ① to ⑧ here for creating the unbounded exploitation window only need to succeed once for the attacker to be able to leak as many bytes as desired.

4.2 Crafting Speculative Race Conditions

To address **C2** and craft speculative race conditions, we turn to the implementation of common synchronization primitives (e.g., *mutex*, *spinlock*, etc.). At the architectural level, these primitives guarantee mutual exclusion of critical regions and this is no different for our gadget. However, they offer no

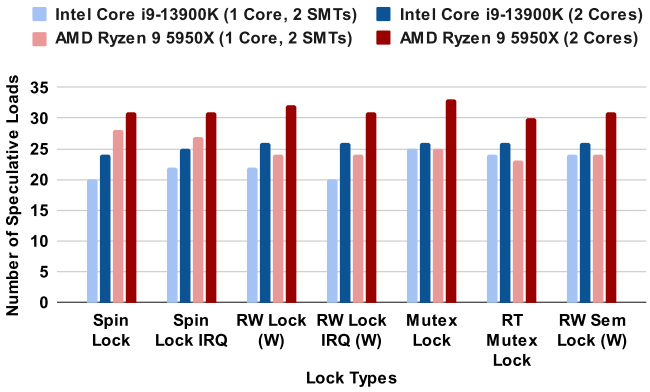

Figure 7: Transient window size for different write-side synchronization mechanisms, i.e., number of speculative loads that leave an observable microarchitectural trace.

microarchitectural guarantees and their behavior is subject to their implementation. To understand their behavior, we investigated the implementation of common synchronization primitives in the Linux kernel. The core implementation of the `mutex_lock` primitive (Figure 6 top) serves to illustrate.

As shown in the figure, line 4 includes a conditional branch that ultimately checks the outcome of the `lock cmpxchgq` instruction. Such instruction atomically compares the current value of the mutex `ptr` with its old one `old`, and, if identical, it means that the mutex can be locked—setting the mutex to the new value `new` and granting access to the guarded critical region. Likewise, if the comparison fails, it means that the mutex has been locked by another thread, therefore the code falls through after failing to acquire the mutex.

Although the comparison is done atomically, the instruction (as we experimentally verified) does not serialize the execution stream. As a result, we can mistrain the conditional branch at line 4 (e.g., simply by acquiring the mutex multiple times) to be taken and consistently trick speculative execution into acquiring a mutex and entering the guarded critical region. Since this is the case regardless of the current (architectural) state of the mutex, we can speculatively acquire a mutex already held by another thread. In other words, the mutex becomes a no-op on the speculative path, leading to a *speculative race condition* and opening the door to arbitrary concurrency vulnerabilities at the microarchitectural level.

Generalizing, our analysis shows *all* the other common write-side synchronization primitives in the Linux kernel are ultimately implemented through a conditional branch and are therefore vulnerable to speculative race conditions. To experimentally confirm this intuition, we tested all such synchronization primitives under speculative execution after mistraining the vulnerable branch. In all cases, we confirmed transient execution of the guarded critical region despite another victim thread already architecturally executing in the region. To determine the transient window size, we measured the maximum number of speculative load instructions we could speculatively execute inside the critical region.

Figure 7 presents our results for two microarchitectures (Intel Core i9-13900K and AMD Ryzen 9 5950X) and two configurations: (i) attacker and victim thread co-located on the same core; (ii) attacker and victim thread running on different cores. As shown in the figure, the transient window size is significant across settings (20+ loads). Moreover, the window is usually larger when the two threads are running across cores, evidencing that the cache coherency protocol plays a crucial role in propagating the lock architectural state across cores before speculation aborts. Finally, our results show some variations across microarchitectures. For instance, the AMD processor has a longer speculation window than the Intel one, matching trends from prior work [61]. Overall, our results show an attacker can speculatively bypass all the common (write-side) synchronization primitives and craft speculative race conditions, turning every (architecturally race-free) critical region into a potential generic Spectre gadget.

To conclude, we note that not all the vulnerable synchronization primitives we analyzed are equally exploitable. For instance, uninterruptible primitives (e.g., irq-safe spinlocks) are not amenable to the interrupt-driven techniques we detail in the next section, preventing the attacker from stretching the race window. Moreover, our analysis focuses on the kernel, excluding primitives that normally apply only to user execution (e.g., Intel TSX-based primitives, also uninterruptible without aborting the underlying memory transactions).

4.3 Exploiting Speculative Race Conditions

To address (C3), we need to mount end-to-end information disclosure attacks. To this end, armed with knowledge of speculative race conditions (SRCs), we can now bypass synchronization primitives (e.g., mutex) on a speculative path and turn safe architectural *uses* into a speculative concurrent *use-after-free* (SCUAF). Next, armed with control over memory reuse, we can escalate our SCUAF primitive to first speculative control-flow hijacking and then speculative information disclosure of some target kernel data. Finally, armed with an unbounded UAF window, we can repeatedly effect our primitives, disclose arbitrary kernel data, and mount end-to-end attacks leaking some target secret in kernel memory. Figure 8 details the steps of an attack based on our NFC gadget.

Initialization. To kickstart the attack, ① the attacker starts executing on a given (attacker) core and triggers the allocation of `hdev` and `hdev->cmd_pending_msg`. Next, ② the attacker mistrains the victim mutex's conditional branch by executing the gadget and acquiring the mutex architecturally many times. Next, ③ the attacker starts the victim thread and the storming threads on the corresponding cores.

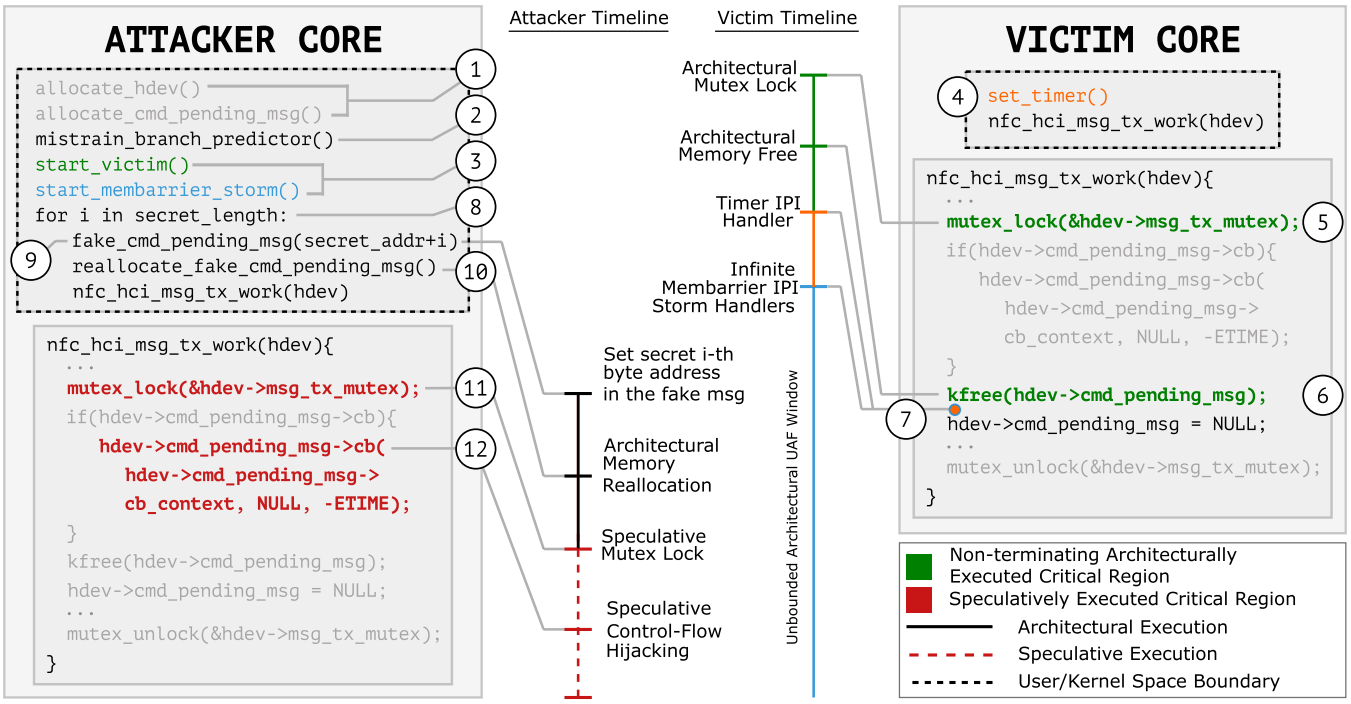

Figure 8: Speculative information disclosure attack exploiting a speculative race condition. Steps 1-4 and 8-10 run in user mode, issuing syscalls to trigger the relevant kernel code. The other steps run in kernel mode. The `nfc_hci_msg_tx_work` gadget code is shown only to explain how the speculative race condition is created.

Unbounded UAF Window. To create an unbounded UAF window, ④ the attacker schedules a high-precision timer on the victim core. Next, ⑤ the attacker causes the victim thread to trigger the execution of the NFC gadget, architecturally locking the mutex and entering the guarded critical region. Next, ⑥ the NFC gadget calls `kfree` on the `hdev->cmd_pending_msg` pointer. Next, ⑦ the high-precision timer expires, interrupting the victim thread when the `hdev->cmd_pending_msg` pointer is still dangling. Shortly after, the attacker signals the storming threads to target the victim core with a storm of `MEMBARRIER` IPIs, preventing the victim thread to resume execution until the attacker signals again the storming threads to terminate.

Speculative Control-Flow Hijacking. ⑧ For each kernel address to leak in order to disclose the target secret, the attacker first crafts a speculative control-flow hijacking (SCFH) primitive. To this end, the attacker needs to allocate a controlled object reusing the memory slot `kfreed` earlier via slab massaging. With SLUB (default slab implementation), the attacker can exploit same-CPU, same-size-class slab cache collision techniques [48] to achieve memory reuse. Specifically, ⑨ the attacker first creates a malicious `struct msgbuf` message, casts it to `struct hci_msg`, and sets: (i) the `hci_msg.cb` callback to the speculative SCFH *target*, (ii) the `hci_msg.cb_context` first callback argument to the SCFH *argument*. Next, ⑩ the attacker calls the `msgsnd` system call with the malicious `msgbuf` message, which ultimately gets allocated in the same slot as the freed `hdev->cmd_pending_msg`. Next, ⑪ the attacker triggers the execution of the gadget. Due to the mistrained branch, the CPU speculatively enters the critical region despite the mutex being architecturally held by the victim thread. Next, ⑫ the gadget speculatively dereferences the dangling pointer, hijacking control flow to the attacker-controlled callback target.

Speculative Information Disclosure. To craft a speculative information disclosure primitive, the attacker needs to first break KALSR using existing techniques [2]. Next, the attacker needs to trigger the SCFH primitive with the callback target set to a kernel disclosure gadget using the controlled first callback argument (`rdi`) as input. For this purpose, we use the `vp_del_vqs` Spectre gadget in Figure 9, also exploited in older kernel versions in prior work [27]. As shown in the figure, the controlled memory referenced by `rdi` is referenced by the gadget to compute the secret address (`r12`+0x28) and the base address of an array (`rax`)—which we can use as the reload buffer of a classic FLUSH+RELOAD Spectre covert channel [45]. The gadget reads the secret in the `rdx` register, then used to index the reload buffer with stride `8` at line 22. To handle the small stride and the high secret entropy, we can use sliding techniques, as done in prior work [27, 73]. Ultimately,

```
1  ffffffff817a1a20 <vp_del_vqs>:
2        ...            // Function Prologue
3  ffffffff817a1a39:    mov rbx, rdi
4  ffffffff817a1a3c:    sub rsp, 0x8
5  ffffffff817a1a40:    mov r12, QWORD PTR [rdi + 0x310]
6  ffffffff817a1a47:    mov QWORD PTR [rbp - 0x30], rax
7  ffffffff817a1a4b:    mov r13, QWORD PTR [r12]
8  ffffffff817a1a4f:    cmp r12, rax
9  ffffffff817a1a52:    jne ffffffff817a1a5c <vp_del_vqs + 0x3c>
10 ffffffff817a1a54:    jmp ffffffff817a1b43 <vp_del_vqs + 0x123>
11 ffffffff817a1a59:    mov r13, rax
12 ffffffff817a1a5c:    movzx r14d, BYTE PTR [rbx + 0x3e0]
13 ffffffff817a1a63:
14 ffffffff817a1a64:    cmp r14b, 0x1
15 ffffffff817a1a68:    ja ffffffff81d05847 <vp_del_vqs.cold>
16 ffffffff817a1a6e:    mov edx, DWORD PTR [r12 + 0x28]
17 ffffffff817a1a73:    and r14d, 0x1
18 ffffffff817a1a77:    lea rsi, [rdx * 8 + 0x0]
19 ffffffff817a1a7e:
20 ffffffff817a1a7f:    je ffffffff817a1ac8 <vp_del_vqs + 0xa8>
21 ffffffff817a1a81:    mov rax, QWORD PTR [rbx + 0x3b8]
22 ffffffff817a1a88:    mov rax, QWORD PTR [rax + rdx * 8]
```

Figure 9: A double-load gadget in the `vp_del_vqs` function with an attacker-controlled `rdi` argument in the Linux kernel. Highlighted in blue are the instructions loading the secret address and in orange the instructions loading the secret byte into reload buffer with a cache stride of `0x8`.The blue and orange instructions are semantically equivalent to their corresponding ones in the Spectre-v1 of Figure 1, with the conditional branch being the red one in Figure 6.

```
@free_script@
expression LOCK;
type TARGET_FUNC_RET_TYPE;
identifier TARGET_FUNC;
type OUTERMOST_STRUCT_TYPE;
OUTER_STRUCT_TYPE *OUTER_STRUCT_PTR;
identifier FREE_STRUCT_PTR;
identifier LOCK_FUNC ~= "_lock|_trylock";
identifier UNLOCK_FUNC ~= "_unlock";
identifier FREE_FUNC ~= "kfree|...";
@@

TARGET_FUNC_RET_TYPE TARGET_FUNC(...){
  ...
  LOCK_FUNC(LOCK)
  ...
  FREE_FUNC(OUTER_STRUCT_PTR->
                        FREE_STRUCT_PTR)
  ...
  OUTER_STRUCT_PTR->FREE_STRUCT_PTR
                                = NULL
  ...
  UNLOCK_FUNC(LOCK)
  ...
}

@type_script depends on free_script@
type FREE_STRUCT_TYPE;
type free_script.OUTER_STRUCT_TYPE;
identifier free_script.FREE_STRUCT_PTR;
@@

OUTER_STRUCT_TYPE {
  ...
  FREE_STRUCT_TYPE FREE_STRUCT_PTR;
  ...
};
```

```
@use_script@
expression LOCK;
type TARGET_FUNC_RET_TYPE;
identifier TARGET_FUNC;
type OUTERMOST_STRUCT_TYPE;
OUTER_STRUCT_TYPE *OUTER_STRUCT_PTR;
identifier USE_STRUCT_PTR;
identifier FPTR;
identifier LOCK_FUNC ~= "_lock|_trylock";
identifier UNLOCK_FUNC ~= "_unlock";
@@

TARGET_FUNC_RET_TYPE TARGET_FUNC(...){
  ...
  LOCK_FUNC(LOCK)
  ...
  OUTER_STRUCT_PTR->
              USE_STRUCT_PTR->FPTR(...)
  ...
  UNLOCK_FUNC(LOCK)
  ...
}

@type_script depends on use_script@
type USE_STRUCT_TYPE;
type use_script.OUTER_STRUCT_TYPE;
identifier use_script.USE_STRUCT_PTR;
@@

OUTER_STRUCT_TYPE {
  ...
  USE_STRUCT_TYPE USE_STRUCT_PTR;
  ...
};
```

Figure 10: Simplified Cocci scripts (left *Free* and right *Use*) scanning for SCUAF gadgets in the Linux kernel.

this translates to 3 gadget repetitions needed to leak one byte from a known prefix [73]. Finally, to break the entropy of the kernel (direct map) address referencing the (user) reload buffer, we can repeatedly probe for known secret data, guess the kernel reload buffer address, and check for a cache signal in the (user) reload buffer until successful.

4.4 Proof-of-Concept Exploit

We implemented a Proof-of-Concept exploit (PoC) based on the end-to-end attack workflow described earlier. To exploit our NFC gadget (not easily reachable on commodity platforms), we simulated an attacker-controlled service interacting with the NFC function via a dedicated system call. We evaluated our PoC an Intel 12th-generation i9-12900K Alder Lake processor. We ran our PoC on Linux kernel v5.15.83, using the default kernel configuration while enabling all transient execution mitigations. Using only 15 SMTs (out of 24 available on the i9-12900K CPU), we experimentally confirmed our PoC exploit can reliably create an unbounded UAF exploitation window after a single attempt. We also observed an average of 13 attempts to achieve successful memory reuse. This confirms that, despite the entropy of slab massaging and that of having to interrupt the victim thread at precisely the right time, our proposed techniques are successful in making exploitation reliable and almost deterministic. Note that *some* nondeterminism is acceptable and has marginal impact on the attack. Indeed, our PoC can detect unsuccessful memory reuse (i.e., lack of signal) and simply terminate the UAF exploitation window and start over. Once memory reuse is successful, we keep the window open and effect as many (fully reliable) speculative information disclosure iterations as needed to complete the attack. When instructing our PoC to leak 10 MB of kernel memory (after breaking the kernel reload buffer address entropy in milliseconds), we observed a leakage rate of 12 KB/s with an error rate of 1% on average (of 100 repetitions). To put things in perspective, on our idle system running Ubuntu with 8 GB of RAM, this translates to an end-to-end attack time of around 35 seconds (all in a single race window) to leak the root password hash from Linux kernel memory—a common target in previous user-to-kernel Spectre exploits [27, 66, 73].

5 Gadget Scanner

The Linux kernel, as a linchpin of open-source software, is susceptible to security vulnerabilities, among which UAFs represent one of the most frequently recurring class of vulnerabilities [48]. Concurrent programming exacerbates the detection and resolution of UAF vulnerabilities due to the intricacies of shared resource management [9].

Existing solutions focus on gadget scanning for the kernel [8, 10, 31, 41, 63, 64, 72, 73], however, none of them target SCUAF gadgets. Therefore, we systematically explore the attack surface for speculative concurrent UAF gadgets in Linux kernel, specifically focusing on (speculative) control-flow hijacking primitives based on UAF. To this end, we rely on the

Coccinelle static code-pattern matching engine [58, 59] to automate the detection of SCUAF vulnerabilities, which arise from the intricate interplay of concurrent UAF patterns and the effect of speculation-unsafe synchronization primitives in critical regions. We immediately note that, since we focus on SCUAF for control-flow hijacking, our SRC analysis is not exhaustive. Moreover, the accuracy of the results is subject to the precision of static pattern matching. Nonetheless, our analysis helps estimate the extent of the problem.

5.1 SCUAF Coccinelle Scripts

Coccinelle is an advanced program matching and transformation engine. Its effectiveness lies in the utilization of the Semantic Patch Language (SmPL) for expressing semantic patches. SmPL enables developers to define intricate code patterns and transformations in a human-readable manner. It is widely used by Linux kernel developers to identify undesirable code patterns across the kernel code base, including misuse of APIs or vulnerabilities such as UAFs.

Coccinelle facilitates the automated identification of UAF vulnerabilities, as an attacker can easily articulate complex patterns indicative of SCUAF vulnerabilities using Coccinelle's pattern definition capabilities.

Figure 10 presents the Cocci scripts used to detect hundreds of SCUAF gadgets, including the `nfc_hci_msg_tx_work` gadget discussed in Section 4. On the left is the script scanning for the *Free* part of the SCUAF vulnerability. Specifically, the script scans for functions with synchronization primitives guarding critical regions and the latter containing calls to (slab) free functions (e.g., kfree) over a nested data structure pointer followed by the `NULL` update of the pointer. On the right of the Figure 10 is the *Use* Cocci script. The script scans for functions with synchronization primitives guarding critical regions with a nested function pointer-based call. Highlighted in orange is the synchronizing pointer in shared memory (i.e., mutex, lock, etc.). The latter has to match between the *Free* and *Use* for the architecturally exclusive execution of the corresponding critical regions to be guaranteed (i.e., architecturally race-free invariant). In green is the pointer to the data structure being freed and in red the pointer to the function pointer being used. In blue is the matching data type of the freed structure and the one containing the function pointer.

Our gadget scanner can find different *variants* of potential SCUAF gadgets, namely, as illustrated in Figure 11:

Ⓐ *Free*: Guarded *free*

Ⓑ *Free*: Guarded *free* + *list_del*

Ⓒ *Free*: Guarded *free* + *NULL*

Ⓓ *Free*: Guarded *free* + pointer update

Ⓔ *Use*: Guarded pointer dereference + function pointer call

Figure 11: Simplified Cocci scripts used to scan the Linux kernel for different variants of guarded *Free* gadgets (left) and guarded *Use* (right). Any combination of these *Free* and *Use* variants results into a potential SCUAF vulnerability.

Ⓕ *Use*: Guarded function pointer *copy* + function pointer copy call

In the case of the `nfc_hci_msg_tx_work` gadget, both the *Free* and *Use* happen to be in the same function (hence the reason we used this gadget as our compact running example), but generally this is not the case as shared resources are more likely to be used in different contexts and functions. We also scan for different levels of nested pointers both for *Free* and *Use*. Our results show that the deepest (i.e., most nested) function pointer in the Linux kernel is 3 chained pointer dereferences away e.g., `ptr1->ptr2->ptr3->fptr(...)`, which our scanner can identify. It can also automatically extract the type of the data structure for each pointer in the chain and match it with any *Free* gadget operating on the same data type, resulting in a potential SCUAF gadget.

Table 1: Gadgets per lock type.

Sync Primitives	Gadgets Found
Mutex	887
Spin Lock	301
Spin Lock IRQ	95
Total	**1283**

Table 2: Gadgets variants vs. nesting levels.

Gadget Variants	Nesting Levels 0	1	2	3
Ⓐ	757	71	1	0
Ⓑ	80	8	0	0
Ⓒ	78	39	0	0
Ⓓ	87	42	0	0
Ⓔ	1387	565	71	2
Ⓕ	9	1	0	0

5.2 Evaluation

We ran our gadget scanner on the source code of Linux kernel v5.15.83. Table 1 presents our results. As shown in the table, our scanner found a total of unique 1,283 gadgets, with 69% (887) of the gadget pairs guarded by *Mutex*, 23% (301) guarded by *Spin Lock*, and 7% (95) guarded by *Spin Lock IRQ*. Almost 78% of the identified gadgets are in device drivers and 15% is located in kernel code serving (device) specific implementations of kernel services (e.g., the NFC HCI core in Figure 3). Table 2 reports the distribution of the nesting levels of the different gadgets variants for both *Free* and *Use* variants. As shown in the table, < 2 nesting levels are the most common and so are variants Ⓐ and Ⓔ. Overall, our static analysis confirms these gadgets are prevalent in modern operating system code bases such as the Linux kernel.

Limitations. Generally, false positives and negatives are possible because we rely on types to match objects that are freed before their uses. False positives occur when, despite having the same type, the objects referenced in the code are different. False negatives occur when a reference to an object may be cast to one of a different type and our analysis fails to account for it. Moreover, despite its widespread use, Coccinelle does exhibit one significant limitation in the exploration of SCUAF gadgets in the Linux kernel. Specifically, Coccinelle provides no inter-procedural analysis capabilities. Inter-procedural analysis involves examining the interactions and dependencies between different functions or procedures within a program. In the case of the Linux kernel, which is characterized by a multitude of interconnected functions, Coccinelle's inability to perform inter-procedural analysis may result in incomplete vulnerability detection in general and for SCUAF in particular. Indeed, SCUAF vulnerabilities often span multiple functions and complex execution paths, thus the lack of inter-procedural analysis may result in false negatives. Moreover, additional imprecision may originate from static pattern matching's inability to capture arbitrarily complex patterns, potentially leading to additional false negatives.

Finally, concerning false positives, our static analysis-based approach cannot guarantee that the identified gadgets can be realistically reachable in a practical attack. Indeed, reachability verification is a hard problem for SCUAF and concurrent UAF in general, as many concurrent UAF vulnerabilities plague device-specific code—as also reported in previous work [9]—and reachability is subject to device availability. For instance, we could not easily reach the `nfc_hci_msg_tx_work` gadget we used in our attack. Since the gadget is in an NFC driver, we contacted the maintainers to ask which devices use this code but never received an answer. We tried 3 NFC devices and development platforms, but none of them used the gadget code. Therefore, we implemented a system call extension making use of the exact same code as the NFC driver, to simulate it being exercised as realistically as possible. Nonetheless, while precise gadget and reachability analysis are crucial for gadget-specific mitigations (e.g., hardening seemingly dangerous *Use* patterns against attacks), we found these properties to be less important for SCUAF since a generic and efficient gadget-agnostic mitigation not just for SCUAFs but for SRCs overall is at reach. We present such mitigation in the next section.

6 Mitigation

To mitigate the SRC class of vulnerabilities, we implement and evaluate the simplest, most robust, and generic one: introducing a serializing instruction in every affected synchronization primitive before it grants access to the guarded critical region, thus terminating the speculative path. This provides a baseline to evaluate any future mitigations, and, as mentioned, mitigates not only the SCUAF vulnerabilities presented in the paper, but all other potential SRC vulnerabilities.

6.1 Mitigation via Serialization

Our approach to mitigate Speculative Race Conditions (SRCs) is to place a serializing instruction such as `lfence` after the `lock cmpxchg` instruction (i.e., bottom instruction in Figure 6) in each of the affected synchronization primitives. We have implemented such a mitigation in just a few lines of code by patching the `arch/x86/include/asm/cmpxchg.h` Linux kernel source file. Specifically, our patch adds a `lfence` instruction in both the `__raw_cmpxchg` and `__raw_try_cmpxchg` assembly macros, which are used to implement all (write-side) synchronization primitives.

Table 3: LMBench performance overhead of our mitigation.

LMBench Test	Mean Performance Overhead	95% Confidence Interval
null call	0.00%	±0.00%
null I/O	0.00%	±0.00%
stat	0.32%	±0.37%
open clos	11.02%	±0.46%
slct TCP	0.19%	±0.16%
sig inst	11.11%	±1.42%
sig hndl	1.38%	±1.05%
fork proc	10.82%	±1.17%
exec proc	7.53%	±0.56%
sh proc	5.93%	±0.21%
2p/0K ctxsw	6.91%	±0.37%
2p/16K ctxsw	6.10%	±0.24%
2p/64K ctxsw	7.00%	±0.47%
8p/16K ctxsw	5.76%	±0.40%
8p/64K ctxsw	5.38%	±0.73%
16p/16K ctxsw	5.83%	±0.64%
16p/64K ctxsw	3.93%	±1.45%
2p/0K ctxsw	6.91%	±0.37%
Latency - Pipe	9.73%	±0.13%
Latency - AF UNIX	7.37%	±1.18%
UDP	7.98%	±0.49%
RPC/UDP	8.54%	±0.63%
Latency - TCP	14.29%	±0.37%
RPC/TCP	8.30%	±0.33%
0K File Create	8.64%	±0.28%
0K File Delete	12.35%	±0.13%
10K File Create	7.18%	±0.27%
10K File Delete	11.37%	±0.18%
Mmap Latency	9.67%	±0.24%
Prot Fault	4.68%	±2.62%
Page Fault	7.90%	±0.32%
100fd selct	-0.23%	±1.40%
Bandwidth - Pipe	0.00%	±0.00%
Bandwidth - AF UNIX	-9.27%	±0.95%
Bandwidth - TCP	6.67%	±1.27%
File reread	3.40%	±0.40%
Mmap reread	0.20%	±0.39%
Bcopy (libc)	0.30%	±0.20%
Bcopy (hand)	-0.32%	±0.32%
Mem read	0.00%	±0.00%
Mem write	-0.02%	±0.19%
Overall Geomean	**5.13%**	

6.2 Evaluation

To evaluate the performance impact of our mitigation on the Linux kernel, we ran two benchmarks, i.e., the standard LMBench [54] since: (i) it is a system call benchmark commonly used by the security and Linux community to evaluate Linux kernel performance; (ii) it allows one to compare mitigation overheads against other solutions; (iii) it includes parallel benchmarks that stress synchronization primitives and thus uncover the overhead of our mitigation unlike other benchmarks we tried. To further evaluate our mitigation overhead, we also included our own microbenchmark to stress-test the synchronization primitives.

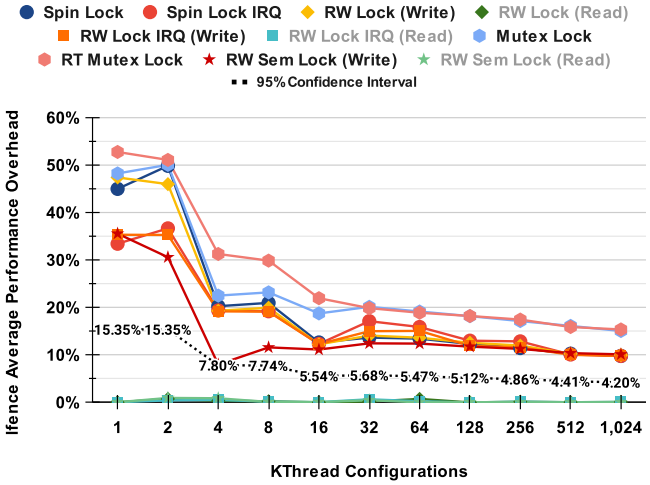

Figure 12: Average performance overhead of our mitigation across different kernel thread configurations and synchronization primitives. The branchless read-side synchronization primitives (in grey) experience no performance overhead.

LMBench. Table 3 presents the results of the LMBench benchmark for over 30 independent runs on our test platform (Section 4). In the default Linux kernel configuration, we measured an overall geomean performance overhead of 5.13%. We believe such a performance overhead is well within the range of a practical mitigation, especially when compared to the 95% LMBench geomean overhead of the default transient attack mitigations enabled on Ubuntu [32].

Microbenchmark. We designed a stress-test microbenchmark which consists of a kernel module measuring the time it takes to simultaneously run N kthreads equally distributed over all CPU cores, with each kthread acquiring and releasing the same synchronization primitive for one million iterations.

Figure 12 presents the average (over 30 independent runs on our test platform) performance overhead caused by our mitigation, serializing the execution of each synchronization primitive across the different kthreads configurations.

As shown in the figure, read-side synchronization primitives (in grey), which contain no conditional branch, are unaffected by the mitigation and thus experience no performance overhead. On the write side, we observe the highest overhead in a synthetic single-kthread configuration, ranging between 33% for *Spin Lock IRQ* to 53% for *RT Mutex Lock*. The overhead decreases to 10% for 1,024 kthreads. This trend suggests that, with an increasing number of kthreads contending the same synchronization "lock", the performance overhead of our mitigation is increasingly masked by lock contention overhead. Overall, our results confirm the overhead of our mitigation is isolated in synchronization-heavy scenarios and even then may be aggressively masked by lock contention.

7 Related Work

7.1 UAF Detection

Most of the work on detecting UAF is focused on sequential bugs. Solutions like UAFChecker [78] and Hua et al. [75] apply static analysis to find problematic code sequences, while others [55, 70] rely on fuzzing. Another technique to detect UAF bugs is to observe memory accesses at runtime [14, 46, 82]. To detect concurrency UAF vulnerabilities, UFO [36] uses execution traces and applies model checking to infer thread causality. Alternatively, DCUAF [9] relies and extends lockset analysis to verify whenever a concurrency UAF is present. ConVul [15] is also based on execution traces and can detect concurrency UAF by identifying exchangeable events. Finally, DDRace [80] specifically targets UAFs in Linux drivers by implementing new heuristics and metrics to simplify the work of the directed fuzzer in targeted locations. Given that they do not analyze transient execution, none of these solutions can find SCUAFs.

7.2 Fuzzing

Fuzzing is a popular technique to find vulnerabilities in software. By feeding crafted input to a program, a fuzzer looks for instances that make the program crash. In this section, we briefly survey existing work on directed fuzzing and then specifically focus on fuzzing to find race conditions. None of the existing work described in this section can find SCUAFs.

Directed Fuzzing. Directed fuzzing techniques aim to reach specific targets within the code [13, 18, 22, 47, 79]. For instance, to reach its targets, AFLGo [13] attempt to minimize the average distance of the basic blocks found in execution traces that link an input to the fuzzing targets. Work like FuzzGuard [82] and Beacon [35] instead apply techniques like deep learning and static analysis to filter inputs that cannot reach the fuzzing targets. Related to UAF fuzzing, CAFL [47] uses a constraint-distance metric that is able to prioritize the inputs towards the goal. The constraints are expressed as a combination of a target site and data conditions which can be used to find UAFs. However, CAFL is only focused on sequential UAFs.

Fuzzing for Concurrency Bugs. Fuzzing techniques have also been extended to concurrent programs to find specific vulnerabilities such as data race bugs. In this area, solutions like RAZZER [38], ConAFL [51], RaceFuzzer [62], and others [28, 39, 44, 67] statically identify potential race situations and then dynamically test the interleavings using generic fuzzing. The program is executed with run-time instrumentation or in a virtualized environment where the race is checked. DDRace [80] works in a similar fashion but employs directed fuzzing instead of traditional fuzzing, reducing the input space and improving performance. Given that data race bugs present themselves in different settings, Conzzer [40], Muzz [17], and KRace [74] design ad-hoc coverage metrics that are thread- and context-aware.

7.3 Gadget Scanning

The ultimate goal of gadget scanning is to find code patterns of interest in a target program, either for offensive or defensive purposes. In the context of transient execution attacks, the majority of scanners, with the exception of [41, 43], are designed to detect bounds check bypass patterns, like the one in Figure 1. Two types of scanners exist, that is based on *static* or *dynamic* analyzers, described in the following. None of the existing work described in this section can find SCUAFs.

Static Analyzers. Existing static gadget analyzers use a plethora of different techniques to scan either source or binary code. For instance, Smatch and Respectre [16, 37] rely on pattern matching against the program's source code while others [19] operate on binary code. oo7 [69] uses static taint analysis while Spectector [30] and KLEESpectre [68] use symbolic execution to detect valid Spectre gadgets. Gadget scanners based on static analysis techniques often lead to a high number of false positives, while those based on symbolic execution tend to suffer from path explosion issues, hindering their scalability.

Dynamic Analyzers. Dynamic analysis techniques like fuzzing and dynamic taint analysis (DTA) can also be used to detect vulnerable patterns. SpecFuzz [56] relies on fuzzing and sanitizers to detect bounds check bypass violations. SpecTaint [60] relies on DTA to link the memory accesses and the leakage points. Finally, Kasper [41] is also based on DTA and sanitizers but its the detection capabilities go beyond the simple bounds check bypass gadget case. Dynamic analysis techniques for Spectre gadgets, like static analysis, also suffer from false positives. Moreover, they also suffer from false negatives due to lack of coverage. Indeed, the analysis is limited to the code executed during the analysis. Reaching high coverage is particularly challenging for large code bases such as the Linux kernel.

7.4 Intel SGX

The techniques we use to control the race window are comparable to those used in various SGX controlled-channel attacks. AsyncShock [71] shows how controlled-channel attacks can be used to exploit concurrency bugs in SGX enclaves. Similar to the use of the non-interruptible *kfree* in our work, SGX-Step [65] uses a coarse-grained APIC timer interrupt to reliably interrupt enclave execution at instruction granularity. AEX-Notify [20] proposes a defense against such attacks.

8 Discussion

SRC Beyond SCUAF. In this paper, we mainly analyze SCUAFs, because architectural UAF vulnerabilities represent one of the most frequent class of memory error vulnerabilities [5–7], thus it is very likely that SCUAF gadgets have the recurrence. Nevertheless, other classes of exploitable SRCs may exist. For example, an SRC where one thread may update a shared, lock-protected index, into an array architecturally, and another thread, speculatively bypassing the corresponding lock and writing attacker-controlled data at that index (See Appendix). Such a speculative buffer overflow type of SRC is likely exploitable, however we have not found kernel coding patterns likely to lead to such a condition in practice in the Linux kernel. More generally, we deem that any SRC may lead to leaking secret data as soon as the speculative thread is susceptible to a speculative control-flow hijack, through any traditional concurrent memory corruption pattern. Nonetheless, finding such gadgets in practice requires significant work as we show in Section 5, and we leave to future work to develop novel approaches to find such gadgets, for example by extending a speculative vulnerability fuzzer [41, 56] to a concurrent fuzzing setting. Note that the mitigation we propose in Section 6 prevents all potential attacks in the generalized SRC class, not merely SCUAFs.

SRC with Speculative Writing (*Free*) Thread. We recall that, by definition, a traditional race condition requires at least one writing thread. Our definition of SRCs in Section 3.1 is worded such that the architectural thread is the writing (*Free*) thread, and the speculative thread is the reading (*Use*) thread. This is because, on existing microarchitectures, writes are only visible to other threads once they are committed, therefore the writing thread cannot bypass the synchronization mechanism. However, CPU architectures where writes may become visible to other threads during speculation, via Store-To-Load (STL) forwarding [33] across simultaneously executing microarchitectural threads, have been discussed in the literature [24, 25]. Such a microarchitecture, if implemented in practice, would unlock additional gadgets.

Beyond x86 and Linux. While we have explicitly focused on x86 and Linux in the paper, SRCs affect other hardware and software targets as well. On the hardware front, we have verified that all the major hardware vendors are affected by SRCs since, regardless of the particular compare-and-exchange instruction implementation, the conditional branch that follows is subject to branch (mis)prediction. In other words, all the microarchitectures affected by Spectre-v1 are also affected by SRCs. On the software front, any target relying on conditional branches to determine whether to enter critical regions—a common design pattern that extends well beyond Linux—is vulnerable to SRCs. In summary, any OS, hypervisor, etc. implementing synchronization primitives through conditional branches and running on any microarchitecture (e.g., x86, ARM, RISC-V, etc.) which allows conditional branches to be speculatively executed without any serializing instruction on that path, is vulnerable to SRCs.

9 Conclusion

In this paper, we presented Speculative Race Conditions (SRCs), a new class of speculative execution vulnerabilities. SRCs stem from all the common synchronization primitives using a conditional branch as a building block (Section 4.2). Such an implementation allows attackers to mistrain the branch prediction unit and speculatively enter a critical region already concurrently accessed by another thread. This enables attackers to bypass these synchronization primitives (Section 4.2) in *all* critical regions, reintroducing many (otherwise architecturally-mitigated) security issues such as Use-After-Free (UAF) and control-flow hijacking.

To study the security impact of SRCs, we focused on Speculative Concurrent Use-After-Frees (Section 4.3), a subclass of SRCs which speculatively exploits a concurrent UAF vulnerability. We demonstrated the practicality of SCUAF attacks by developing a Proof of Concept (PoC) which allows an attacker to speculatively disclose arbitrary kernel memory at a leakage rate of 12 KB/s (Section 4.4). To mount an end-to-end speculative disclosure attack with a tiny UAF exploitation window, we presented a novel race window-massaging technique which allows an attacker to (i) precisely interrupt any kernel thread at any point during its execution and (ii) create an unbounded architectural UAF window (Section 4.1). While our technique is also applicable to the (already challenging) exploitation of architectural race conditions, it is particularly powerful in the context of SRCs, allowing the attacker to mount end-to-end attacks within a *single* race window. Furthermore, to explore the SCUAF attack surface, we developed a gadget scanner which identified 1,283 potentially vulnerable SCUAF gadgets in the Linux kernel (Section 5). Finally, we proposed and evaluated a new mitigation which tackles the root cause of SRCs by placing a serializing instruction at the vulnerable conditional branch, thus terminating the speculative path. Our proposed mitigation incurs a ≈5% geomean performance overhead on LMBench (Section 6).

10 Disclosure

We disclosed Speculative Race Conditions to the major hardware vendors (Intel, AMD, ARM, IBM) and the Linux kernel in late 2023. Hardware vendors have further notified other affected software (OS / hypervisors) vendors and all parties have acknowledged the reported issue (CVE-2024-2193 [1]). Specifically, AMD responded with an explicit impact statement (i.e., "existing [Spectre-v1] mitigations apply"), pointing to the attacks relying on conditional branch mis-speculation,

like Spectre-v1. The Linux kernel developers have no immediate plans to implement serialization of synchronization primitives due to performance concerns. However, they confirmed the IPI storming issue (CVE-2024-26602 [3]) and implemented an IPI rate limiting feature to address the CPU saturation issue by adding a synchronization mutex on the path of `sys_membarrier` and avoiding its concurrent execution on multiple cores [4]. Unfortunately, as our experiments show (Figure 5), hindering IPI storming primitives (i.e., 0 storming cores) is insufficient to completely close the attack surface.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback, Andrew Cooper for his early comments on the paper, Julia Lawall for the Coccinelle clarifications, and Alessandro Sorniotti for the early discussions about the project. This work was partially supported by Intel Corporation through the "Allocamelus" project, by the Dutch Research Council (NWO) through project "INTERSECT", and by the European Union's Horizon Europe program under grant agreement No. 101120962 ("Rescale").

References

[1] GhostRace - CVE-2024-2193. https://www.cve.org/CVERecord?id=CVE-2024-2193.

[2] Kernel address space layout derandomization (kasld). https://github.com/bcoles/kasld.

[3] Membarrier IPI Storming - CVE-2024-26602. https://lore.kernel.org/lkml/2024022614-unhappily-python-2cd0@gregkh/.

[4] sched/membarrier: reduce the ability to hammer on sys_membarrier. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=944d5fe50f3f03daacfea16300e656a1691c4a23.

[5] Cwe-416: Use after free, 2006.

[6] Analysis and exploitation of pegasus kernel vulnerabilities (cve-2016-4655 / cve-2016-4656), 2016.

[7] Mac os x privilege escalation via use-after-free: Cve-2016-1828, 2016.

[8] Intel research on disclosure gadgets at indirect branch targets in the linux* kernel, 2022.

[9] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *USENIX ATC*, 2019.

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

[11] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged {CPU} features. In *OSDI*, 2012.

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

[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *CCS*, 2017.

[14] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[15] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *ESEC/FSE*, 2019.

[16] Dan Carpenter. Smatch check for Spectre stuff, 2018.

[17] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *USENIX Security*, 2020.

[18] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS*, 2018.

[19] Nick Clifton. Spectre variant 1 scanning tool, 2018.

[20] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *USENIX Security*, 2023.

[21] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[22] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *ICSE*, 2022.

[23] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *EuroSys*, 2019.

[24] Josué Feliu, Alberto Ros, Manuel E Acacio, and Stefanos Kaxiras. Itslf: Inter-thread store-to-load forwardingin simultaneous multithreading. In *MICRO*, 2021.

[25] Josué Feliu, Alberto Ros, Manuel E Acacio, and Stefanos Kaxiras. Speculative inter-thread store-to-load forwarding in smt architectures. *Journal of Parallel and Distributed Computing*, 173:94–106, 2023.

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

[27] Enes Goktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[28] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *SOSP*, 2021.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

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

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

[32] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. Quarantine: Mitigating transient execution attacks with physical domain isolation. In *RAID*, 2023.

[33] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[34] Jann Horn. Exploiting race conditions on [ancient] Linux. In *LSSEU*, 2019.

[35] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. BEACON: Directed grey-box fuzzing with provable path pruning. In *S&P*, 2022.

[36] Jeff Huang. Ufo: Predictive concurrency use-after-free detection. In *ICSE*, 2018.

[37] Open Source Security Inc. Respectre: The state of the art in Spectre defenses, 2018.

[38] Dae Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *S&P*, 2019.

[39] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *S&P*, 2023.

[40] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *NDSS*, 2022.

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

[42] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv:1807.03757*.

[43] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. *arXiv preprint arXiv:2106.15601*, 2021.

[44] Youngjoo Ko, Bin Zhu, and Jong Kim. Fuzzing with automatically controlled interleavings to detect concurrency bugs. *Journal of Systems and Software*, 191:111379, 2022.

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

[46] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[47] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *USENIX Security*, 2021.

[48] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique. In *USENIX Security*, 2023.

[49] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: Exploiting kernel races through raising interrupts. In *USENIX Security*, 2021.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security*, 2018.

[51] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. In *ACSAC*, 2018.

[52] Andrea Mambretti, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, and Anil Kurmus. Two methods for exploiting speculative control flow hijacks. In *USENIX WOOT 19*.

[53] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. In *IEEE EuroS&P*, 2021.

[54] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[55] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities, 2020.

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

[57] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical. *arXiv preprint arXiv:2103.03443*, 2021.

[58] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *PLOS*, 2006.

[59] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Semantic patches, documenting and automating collateral evolutions in Linux device drivers. In *OLS*, 2007.

[60] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering Spectre gadgets. 2021.

[61] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*, 2021.

[62] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[63] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G Shin. SpecHammer: Combining Spectre and Rowhammer for new speculative attacks. In *S&P*, 2022.

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

[65] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgxstep: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

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

[67] Nischai Vinesh and M. Sethumadhavan. Confuzz—a concurrency fuzzer. In Ashish Kumar Luhach, Janos Arpad Kosa, Ramesh Chandra Poonia, Xiao-Zhi Gao, and Dharm Singh, editors, *ICTSCI*, 2020.

[68] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KleeSpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *TOSEM*, 29(3), 2020.

[69] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against Spectre attacks via program analysis. *IEEE TSE*, PP:1–1, 11 2019.

[70] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *ICSE*, 2020.

[71] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *ESORICS*, 2016.

[72] 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.

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

[74] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *S&P*, 2020.

[75] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE*, 2018.

[76] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

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

[78] Jiayi Ye, Chao Zhang, and Xinhui Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *CCS*, 2014.

[79] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *CCS*, 2017.

[80] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. Ddrace: Finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing.

[81] Google Project Zero. Racing against the clock – hitting a tiny kernel race window, 2023.

[82] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security*, 2020.

A Additional SRC Code Patterns

We discuss SRC code patterns other than SCUAFs that are potentially exploitable. In principle, any data race pattern that may lead to a memory error could be vulnerable, but they must also exhibit a Spectre-like transmitter pattern as shown in the examples below.

A data race may lead for example to an out-of-bounds access. We show in Listing 1 a code pattern where Thread 1, which is the architecturally executing thread, performs an update on a shared index variable. We can also assume that the offset value can be attacker-controlled. Such a pattern could occur for example when using a circular buffer. The transiently executed thread performs a speculative write, which can be out-of-bounds. Assuming that array elements are the same size as pointers, this scenario would lead to the ability to control the function pointer. Note that a speculative control flow hijack is not necessary, and any known Spectre transmitter pattern could also suffice, as shown in Listing 2. This case immediately leads to an arbitrary read primitive, without the need for additional gadgets.

Listing 1: OOB access SRC

```
//Thread 1 (architectural, interrupted):
mutex_lock (&m);
shared_idx += offset; // interrupt here
if (shared_idx > ARRAY_SIZE)
    shared_idx = 0;
mutex_unlock (&m);

//Thread 2 (transient):
mutex_lock (&m);

// spec. OOB write
array[shared_idx] = val;

// control flow hijack
fptr();
mutex_unlock (&m);
```

Listing 2: Other transmitter

```
//Thread 2 (transient):
mutex_lock (&m);

// access secret
byte = array[shared_idx];

// transmit
val = probe_array[4096*byte];
mutex_unlock (&m);
```

Other patterns are highly likely to exist. We expect future work to further study their prevalence and exploitability.