Dynamic Detection of Vulnerable DMA Race Conditions

Brian Johannesmeyer[*†]
Qualcomm Technologies, Inc.
San Diego, United States of America
brian.johannesmeyer@qualcomm.com

Raphael Isemann[*]
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
r.isemann@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
herbertb@cs.vu.nl

Abstract

The drivers of modern operating systems use Direct Memory Access (DMA) to efficiently communicate with peripheral devices. Since the memory accessed by DMA is a shared resource between driver and device, it is a possible source of race conditions. Peripheral devices are also often untrusted, so these race conditions open up a new potential attack vector against a trusted OS kernel.

In this paper, we present DMARacer, a dynamic detector called for these DMA-based race conditions in kernel code. DMARacer tracks memory accesses to DMA memory throughout the kernel's lifetime and analyses them for various indicators of race conditions. Additionally, upon detecting a race condition, DMARacer uses taint tracking to trace its impact and identify any potential vulnerabilities it may trigger, such as memory corruption or denial-of-service. We used DMARacer to search the drivers of the Linux kernel for DMA-based errors and find that DMA-based race conditions are a systemic issue in driver code. In total, DMARacer was able to detect 817 problematic memory accesses and 344 vulnerable operations in the scanned Linux kernel drivers.

CCS Concepts

• **Security and privacy → Operating systems security**; • **Software and its engineering →** *Dynamic analysis*; • **Hardware →** Communication hardware, interfaces and storage.

Keywords

Direct Memory Access, Race Condition Vulnerabilities, Dynamic Taint Analysis, Linux Kernel

ACM Reference Format:
Brian Johannesmeyer, Raphael Isemann, Cristiano Giuffrida, and Herbert Bos. 2025. Dynamic Detection of Vulnerable DMA Race Conditions. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765126

[*]Both authors contributed equally to this research.
[†]Work done while at Vrije Universiteit Amsterdam.

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765126

1 Introduction

> *"We trust the hardware pretty implicitly. There are bits and pieces of the kernel that are starting to nibble away at the "do we trust this?" portion, but for the most part, this is not how Linux was designed at all."*
>
> *— Greg Kroah-Hartman, Linux kernel maintainer (in response to our disclosure)*

Where OS developers in the past limited their security concerns to malicious user programs [43], operating system kernels today *have* to include peripheral devices in their threat model [7]. Indeed, most general-purpose computers now come equipped with IOMMUs [7]—mirroring established protection mechanisms that prevent unauthorized memory access by user programs, by means of MMUs [36], to also isolate memory for external devices. However, there is also memory that the kernel intentionally shares to communicate with either user space or peripherals—and doing so introduces the risk of race conditions.

For example, the shared memory used to transfer data from a user program to the kernel in a system call may easily become a source of race conditions. Moreover, the kernel and user/device have different privilege levels, putting them out of reach of standard synchronization approaches and verification tools [6, 16, 17, 39, 44, 48] that rely on all processes synchronizing voluntarily. Previous work extensively studied race conditions, such as double fetches, at the system call interface [11, 15, 22, 28, 29, 46, 53, 54, 58].

However, the problem also exists for drivers at the boundary between kernel and untrusted devices. As with the system call interface, the kernel and device both access shared memory to communicate. Using Direct Memory Access (DMA), a malicious device may surrepitiously modify data that the kernel assumes is valid. Recent work studied DMA-based communication between driver and device and highlighted that standard security practices (e.g., input validation and IOMMU protection) are not yet consistently adopted [8, 10, 14, 18, 20, 25, 30, 31, 33, 37, 40, 49, 50, 52, 57, 59, 61].

Despite these findings, the research community has largely ignored the issue of DMA-based race conditions. The shared memory and the different privilege levels mirror the conditions that lead to race conditions and double fetch bugs on the user-kernel interface. However, at the device-kernel interface, they may lead not just to "classic" TOCTOU-like conditions, but also other, less-known

but equally dangerous, ones. Examples include races in streaming DMA, as well as cases where attackers corrupt kernel-initialized data. The latter category is of particular concern as they are often exploitable [8, 31] and we find that such vulnerabilities are much more common than TOCTOU ones. This raises the question of how to detect such conditions in the various device drivers found in modern kernels?

In this work, we fill this gap by identifying DMA-based race conditions using insights from prior research on user-to-kernel race conditions. Unfortunately, applying these insights to the *device-to-kernel* domain poses unique challenges. For instance, few (if any) of the interactions across this boundary are standardized and interposition is hard: devices may interact with the kernel through custom driver interrupts, and the kernel may access DMA through ordinary memory operations. Thus, we cannot incorporate the methods from previous studies of unsafe DMA accesses directly and must instead adapt them specifically to DMA race conditions. In particular, unlike unsafe DMA accesses, DMA race conditions require analyzing memory accesses collectively, as well as tracking of long-lived state. For this purpose, we use dynamic taint analysis (DTA) [35] to track such complex state across complex interactions.

We present DMARacer, a dynamic DMA race condition detector for the Linux kernel. DMARacer monitors runtime invariants to determine whether DMA accesses are race-free. When DMARacer identifies a violation—a race condition—it reports the access as an *errant access*. If the errant access involves attacker-controllable data (e.g., loading from coherent DMA), we apply taint to the data to track it throughout the kernel's execution. Finally, if the tainted data reaches a security-sensitive operation (e.g., a memory write pointer), DMARacer flags it as a *vulnerable operation*.

Like other sanitizers, DMARacer instruments the kernel to detect these issues during runtime. Specifically, it (i) tracks DMA state by hooking into every DMA operation, (ii) identifies errant accesses by monitoring all memory accesses, (iii) identifies vulnerable operations through DTA policies, and (iv) covers DMA race conditions by executing a variety of device-to-kernel interactions. In combiation, it enables DMARacer to detect DMA race conditions.

By applying race condition detection to the novel domain of DMA data, DMARacer identifies 817 errant accesses and 344 vulnerable operations across the kernel—with false positive rates of 0% and 9%, respectively. Moreover, our case studies show that these race conditions are not easily eliminated: many are deeply embedded in the semantics of existing drivers and APIs.

Contributions. We make the following contributions:

- We present a new dynamic analysis approach for detecting DMA race conditions and vulnerable device-to-kernel interactions.
- We develop DMARacer, an open-source[1] DMA race condition detector for the Linux kernel.
- We evaluate DMARacer on a recent kernel to identify hundreds of errant accesses and vulnerable operations, and present case studies that highlight that the issues are difficult to mitigate.

[1] DMARacer is available at https://github.com/vusec/dmaracer.

2 Background

Race conditions. Race conditions are non-deterministic errors that occur when several processes access a shared resource without synchronization[34, 44]. A common synchronization mechanism are locks that grant exclusive access to the variable for a limited time. Many synchronization primitives require the cooperation of all involved processes to work correctly. They do not work if one of the processes is malicious and ignores the synchronization primitive.

TOCTOU bugs. A special kind of bug caused by race conditions are time-of-check to time-of-use (TOCTOU) errors. Here, a piece of code first checks whether a certain property holds for a shared resource. The rest of the code then incorrectly assumes this property still holds [41], even though there is no mechanism in place that ensures the immutability of the shared resource. An unprivileged attacker can abuse this behavior to potentially exploit the code that relies on the checked property. In a concrete attack, the attacker would set the shared resource to a 'good' state to satisfy the check, and then modify it before the first to a 'bad' state that causes unintended behavior.

Double fetch bugs. A common attack scenario involving TOCTOU errors are kernel system call handlers where they are also referred to as *double-fetch bugs* [22, 28, 46, 53, 54, 58]. In this scenario, the attacker performs a system call as an unprivileged user and the shared resource is the memory containing the system call arguments. This memory is accessible by both user and kernel while the system call is being performed. The attacker can therefore modify an argument between the time the kernel checks it for validity (1st fetch) and the actual processing of the arguments (2nd fetch). If the attacker is successful and is able to bypass a critical check such as a buffer size check, the consequences of such an attack can result in information leakage or privilege escalations [53].

Direct memory access. Direct memory access (DMA) is a hardware feature that reduces the CPU workload when communicating with peripheral hardware. In a system with DMA, a dedicated DMA controller is responsible for moving data between system memory and hardware. The CPU itself is only responsible for initiating the data transfers and can be utilized for other tasks while the data is being moved.

Kernel drivers utilize DMA by allocating a special DMA buffer that is bound to a device's registers or memory. The buffer itself is used like any other plain memory buffer in C and can be directly written to and read from without invoking special functions. The driver can decide between two kinds of DMA buffers that differ in when they synchronize their contents with the device.

(1) *Streaming* (or asynchronous) DMA buffers are used for single DMA-based data transfers. The transfer to and from the device is explicitly initialized by the driver. As the assumption is that driver and device do not access the same memory region at the same time, there are generally no strong guarantees with respect to cache coherence.
(2) *Coherent* (or synchronous) DMA buffers are implicitly synchronized between device and system memory. These buffers are set up in cache-coherent memory, therefore any memory writes

done by either device or CPU are immediately made visible to the other. Coherent buffers are used when the driver and the device require concurrent write access.

DMA errors. The two kinds of DMA buffers impose each a very different challenge for a driver that uses them. For streaming DMA buffers, the main challenge is the correct timing of the synchronization request. All data that needs to be transferred must have been written to the buffer before it is synchronized with the device.

Coherent buffers do not impose this requirement, but instead come with different security implications. Because their contents can be concurrently accessed by an untrusted device and the kernel, they can be a potential source of TOCTOU bugs. The possible attack scenario using these bugs is very similar to the TOCTOU attacks on kernel system call handlers (see the previous section). Instead of the user switching out system call arguments, a device could change the contents of a DMA buffer after the driver performed the necessary sanity checks on it.

3 Threat Model

We adopt the threat model of previous work [10], which considers a local attacker who controls a peripheral device (e.g., a USB keyboard) and its workload. The kernel and its drivers are trusted and not compromised. The attacker's goal is to cause a denial of service (e.g., trigger a kernel panic) or achieve privilege escalation (e.g., corrupt kernel memory). The ability to control the peripheral device allows the attacker to observe all DMA accesses performed by the kernel driver and control the values of the accessed DMA memory. E.g., the attacker can observe that a certain part of the DMA memory is accessed twice within a short time span and change the memory contents so that two different values are read by each access. The system memory is protected by an idealized IOMMU that can protect each individual byte of memory against unintentional writes from an external device. This means the system is invulnerable against accidental exposure of memory caused by too coarse-grained IOMMU protection capabilities [31].

4 Defining DMA Race Conditions

In this section, we will define three possible race conditions that can arise out of improper DMA usage.

4.1 Coherent DMA-based Race Conditions

Coherent DMA is simultaneously accessible to both the kernel and a malicious device, similar to how userspace data is simultaneously accessible to both the kernel and a malicious program. Hence, we can gain insights from userspace-based race conditions when defining coherent DMA-based race conditions.

TOCTOU bugs. Previous work on user-to-kernel TOCTOU bugs [22] holds that the kernel should not load the same user data twice within a single execution context (i.e., an interrupt, system call, or new kernel thread). We apply this same rule to coherent DMA:

Invariant 1. Avoiding TOCTOU bugs.

The kernel should not read coherent DMA data more than once within an execution context.

If this invariant is violated, the kernel unsafely races against a malicious device corrupting the previously kernel-loaded data. Although drivers may sometimes busy-poll MMIO/PMIO status registers for ultra-low-latency I/O [27], polling a DMA buffer itself is uncommon—completion is normally reported via interrupts or other asynchronous mechanisms [13]. Thus, our invariant holds in practice (see Section 8).

TOITOU bugs. Recent DMA exploitation techniques [8, 31] target bugs that involve the corruption of kernel-initialized DMA data. We term such cases *TOITOU*, time-of-*initialization* to time-of-use, bugs. A TOITOU bug is defined as a race condition with the following three phases: (1) the victim *stores* valid data in a shared resource, (ii) the attacker overwrites this data after it was stored, and (iii) the victim later loads the data and assumes it is still valid, thus omitting any validity checks. The difference to a TOCTOU bug is that instead of a check being skipped, the user's attempt to satisfy a check by modifying a shared resource is disabled.

Unlike the user-to-kernel domain, where the kernel only rarely initializes data in userspace for later use (e.g., when signal handling [12]), kernel drivers will oftentimes initialize entire data structures in DMA. These drivers also expect the device to play nice and only access the parts that it is supposed to. Typically, the kernel initializes such data early on (e.g., during boot time), then uses it much later on (i.e., after boot time), all the while, expecting it to remain unchanged. Hence, we can derive a second invariant:

Invariant 2. Avoiding TOITOU bugs.

The kernel should not read coherent DMA data if it previously initialized it.

If this invariant is violated, the kernel unsafely races against a malicious device corrupting the previously kernel-initialized data.

4.2 Streaming DMA-based Race Conditions

In contrast to coherent DMA, streaming DMA is only accessible to either the kernel or the device—but not both. The kernel governs this accessibility by invoking synchronization operations that transfer access rights between the kernel and the device.

Inconsistent access bugs. Previous work on identifying unsafe DMA accesses notes that the kernel should not access a streaming DMA region when it is synchronized for device access [10]. Doing so results in an *inconsistent DMA access bug* because the data accessed by the kernel (either in the CPU cache or a bounce buffer) may not be consistent with the actual data in the (device-accessible) DMA buffer [13]. Hence, we apply this same invariant:

Invariant 3. Avoiding inconsistent access bugs.

The kernel should not access streaming DMA data if the region is synchronized with the device.

If this invariant is violated, the kernel unsafely races against the CPU cache or bounce buffer inadvertently corrupting the data. Notably, this bug is very difficult for a malicious device to exploit, so we consider such exploitation out of scope. Instead, it primarily poses a reliability issue when the kernel later uses the corrupted data. Moreover, because synchronization operations are non-blocking, the manifestation of this bug is nondeterministic: the race condition may only occur when the synchronization commits, making it difficult to consistently reproduce and diagnose.

Table 1: DMA operations hooked by DMARacer to track DMA state at runtime.

DMA Op.	Function Hooked	DMARacer Handler
Map	`dma_alloc_attrs()`	Track a *coherent* DMA buffer
	`dma_map_single_attrs()`	Track a *streaming* DMA buffer
	`__dma_map_sg_attrs()`	Track a *streaming* DMA SG list
Unmap	`dma_free_attrs()`	Untrack a *coherent* DMA buffer
	`dma_unmap_page_attrs()`	Untrack a *streaming* DMA buffer
	`dma_unmap_sg_attrs()`	Untrack a *streaming* DMA SG list
Sync	`dma_sync_single_for_cpu()`	*CPU*-sync a DMA region
	`dma_sync_single_for_device()`	*Device*-sync a DMA region

5 Overview

Having defined the various DMA race conditions, we now present the main design challenges and how we addressed them to detect these conditions. To detect DMA race conditions, DMARacer has to track all memory operations and determine which are accessing DMA memory. DMARacer must then identify which parts of the kernel are potentially affected by attacker-controlled data from these errant accesses.

Figure 1 presents an example of how DMARacer detects vulnerable code. Although this example features a hypothetical TOCTOU bug for illustrative purposes (given its similarity to well-studied double-fetch bugs), it is important to note that, as discussed in Section 8.3.1, most vulnerabilities in practice stem from TOITOU bugs instead.

Ⓐ **Tracking DMA regions.** DMA memory regions are dynamically created and then accessed like any other buffer via C pointers in the rest of the kernel code base. We therefore cannot hook a standardized access method (e.g., `copy_from_user()` on the kernel-user barrier) to detect DMA memory accesses. Instead, DMARacer hooks all relevant DMA methods and maintains its own metadata for the location and state of all DMA buffer. In Section 6.1, we detail how we hook the variety of DMA APIs within the kernel.

Ⓑ **Identifying errant accesses.** DMARacer has to determine all DMA memory operations and identify any *unsafe* DMA accesses among them. As DMA is accessed via normal store and load instructions, DMARacer uses its collected DMA metadata to determine which memory operations access DMA buffers. For errors like TOCTOU that involve several memory operations, DMARacer also has to reason about the relationship between different memory accesses. We support this post-hoc analysis of several DMA operations by additionally maintaining a list of all DMA memory accesses. In Section 6.2, we describe how our monitor checks whether DMA is accessed, and if so, whether the access violates an invariant.

Ⓒ **Identifying vulnerable operations.** Once DMARacer detects an errant DMA access, it is still unclear whether the attacker-controlled from the access is actually vulnerable. DMARacer searches for potentially exploitable code by applying taint to attacker-controlled DMA data and tracking it through the kernel's execution. This taint spreads across the kernel during execution until it reaches operations that can enable exploits if certain operands are attacker-controlled (e.g., the address of a store operation). In Section 6.3, we describe our taint policies, and the subtleties in tracking DMA data.

Ⓓ **Covering DMA race conditions.** Unlike the user-kernel boundary with syzkaller [1], there is no production-ready tooling for testing the device-kernel boundary. Given that DMARacer is a *dynamic* analysis tool, we therefore also need a way to run drivers and give them meaningful workloads that utilize DMA operations and spread them across the kernel. To overcome this challenge, we leverage the flexibility of a virtual environment to invoke device-to-kernel interactions. This allows us to easily configure the kernel to support a variety of drivers, attach multiple devices, and run diverse device-specific workloads. In Section 8.1, we explain this pipeline, and how future work could build upon our flexible tooling.

6 Detecting DMA Race Conditions

In this section, we detail each component of our approach to detecting DMA race conditions.

6.1 DMA Operation Hooks

Table 1 presents our DMA operation hooks, which allow us to track DMA state at runtime. For both coherent and streaming DMA, we begin tracking a region at a *map* (or allocate) operation, and stop tracking it at an *unmap* (or free) operation. When streaming DMA is mapped, we designate it is as device-accessible; for every *synchronization* operation thereafter, we designate it as either CPU- or device-accessible. Moreover, because the kernel may synchronize DMA partially—i.e., only a subset of the region—we track synchronization state at a per-byte granularity.

Applicability to various DMA APIs. The kernel offers a variety of APIs to map, unmap, and synchronize DMA. However, because we hook the lowest-level DMA operations, which are called by these APIs, we therefore also hook into these various APIs. For example, the various interfaces for allocating coherent DMA—e.g., the generic (`dma_alloc_coherent()`), managed (`dmam_alloc_coherent()`), pool (`dma_pool_alloc()`), and driver-specific (e.g., `hcd_buffer_alloc()`) APIs—all eventually call `dma_alloc_attrs()`. Therefore, by hooking `dma_alloc_attrs()`, we also hook such allocation interfaces.

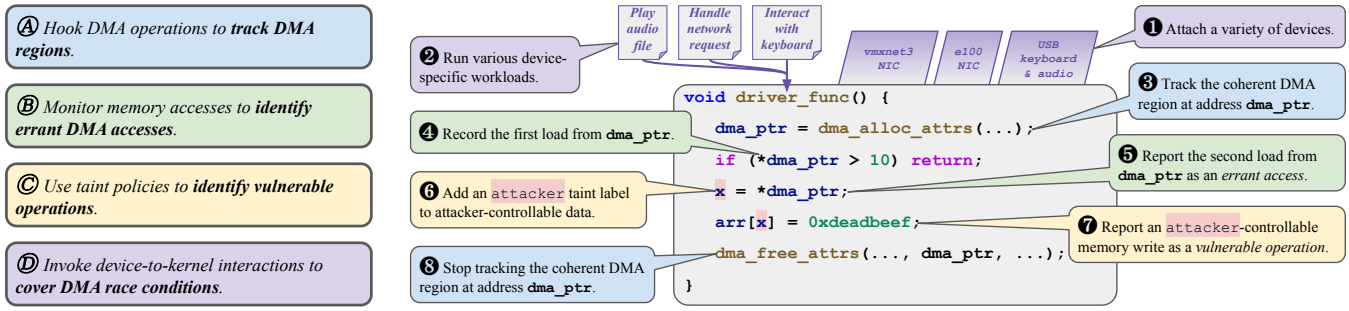
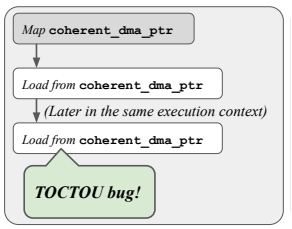
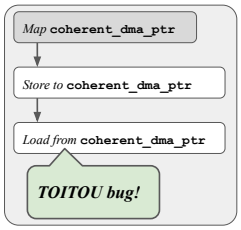


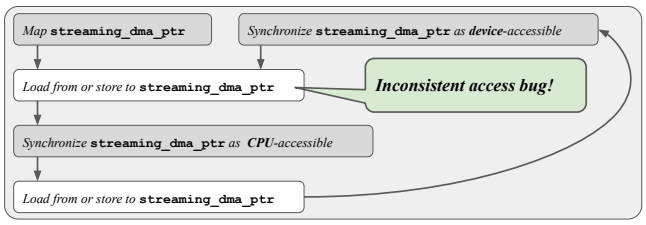
Figure 1: The components (Ⓐ–Ⓓ) used by our approach and the steps (❶–❽) they take to identify an example TOCTOU bug and a dependent attacker-controllable memory write.



(a) Invariant 1 violation.



(b) Invariant 2 violation.



(c) Invariant 3 violation.

Figure 2: Policies of the memory access monitor in determining whether an access violates an invariant.

6.2 Memory Access Monitor

To monitor every memory access, we insert callbacks to our DMARacer runtime library at memory access instructions. From our callback, we first check whether the access is to a DMA region, and if so, whether it violates an invariant, as outlined in Figure 2.

6.2.1 Coherent DMA Accesses. For loads from coherent DMA data, we check whether the data was previously accessed. Specifically, whether it was previously: (i) loaded from within the same execution context, which indicates a TOCTOU error (Invariant 1), or (ii) stored to within the lifetime of the kernel, which indicates a TOITOU error (Invariant 2).

For this purpose, we record every access into one of two structures: (i) if an access *loads* from DMA, we record it into a *local access map*, which is local to a particular execution context, and is cleared when the execution context exits; or (ii) if an access *stores* to DMA, we record it into a *global access map*, which is global to the entire kernel, and persists through the kernel's lifetime. If a load from DMA has a preceding load in the *local* access map, then we report a *TOCTOU bug* (as in Figure 2a). Otherwise, if it has a preceding store in the *global* access map, then we report a *TOITOU bug* (as in Figure 2b).

Storing kernel pointers to coherent DMA. We observe one concerning pattern in DMA usage: the kernel frequently storing pointers to DMA. Typically, this may be because it maintains a data structure (e.g., a linked list) in DMA, and expects the device to only access certain parts of the structure (e.g., the "data" of a linked list, but not the pointers).

However, because devices generally do not have access to the same address space as the kernel, the only practical reason that the kernel would store a pointer to DMA is if it will use it later. Hence, such an operation is either bad practice (at best), or the beginning of a vulnerability[2] (at worst). If it uses the pointer later, we would report it as a TOITOU bug. Therefore, if the kernel stores a pointer to DMA, we report it as an errant access, because we expect the kernel to use it later, which would then be a TOITOU bug. By reporting the initial pointer store as an errant access, we mitigate the case where our dynamic analysis' imperfect coverage fails to cover the subsequent pointer load.

6.2.2 Streaming DMA Accesses. For accesses to streaming DMA, we check whether any part of the accessed region is device-synchronized. If yes, the access violates Invariant 3 and demonstrates an invalid DMA memory access. Figure 2c outlines this process: The kernel *may* access streaming DMA data if it is immediately preceded by a CPU-synchronization operation. However, the kernel *may not* access it if it is immediately preceded by a mapping or device-synchronization operation; otherwise, we report the access as an *inconsistent access bug*.

6.3 Taint Policies

Table 2 summarizes our taint policies, which track attacker data from errant accesses to dependent vulnerable operations.

6.3.1 Taint Sources. From our memory access callbacks (Section 6.2), we add a unique taint label to the output of attacker-controllable errant accesses (i.e., TOCTOU or TOITOU bugs). Hence, as is typical of DTA systems [35], we are able to track the flow of attacker data at runtime by checking its taint label: untainted data *is*

[2]This may also be considered information leakage (e.g., breaking KASLR). However, we focus on the risk of race conditions in this work.

Table 2: Taint policies to track attacker data to dependent vulnerable operations, while avoiding taint explosion.

Taint Policy Type	Operate On	Justification
Source: Track attacker data	Output of a TOCTOU/TOITOU bug	Controllable by a malicious device
Sink: Identify vulnerable operations	Pointer of a memory write	Enables an attacker-controllable memory corruption primitive
	Condition of a loop/assertion	Enables an attacker-controllable denial-of-service primitive
Clear: Avoid taint explosion	`AND` instruction	Instruction used to access specific bits of DMA data
	Kernel hot paths	Execution may spill taint into frequently accessed global data
	Execution context entry	Cross-execution context dataflows are challenging to reproduce

not attacker data, and tainted data *is* attacker data. In our case, our taint label identifies the specific errant access (i.e., the backtrace) that loaded the attacker data.

6.3.2 Taint Sinks. We track the flow of attacker data to certain *security-sensitive operations*. Following previous work [10], we target operations leading to memory corruption and denial-of-service (DoS) vulnerabilities. Specifically, our security-sensitive operations are: (i) A tainted pointer of a store instruction, because an attacker may redirect it to corrupt memory; hence, we report it as a *vulnerable write*. (ii) A tainted loop condition, because an attacker may corrupt it to loop indefinitely, leading to a DoS or buffer overflow; hence we we report it as a *vulnerable loop*. (iii) A tainted assertion condition, because an attacker may corrupt it to fail the assertion, leading to a DoS, so we report it as a *vulnerable assert*.

We note that this is not an exhaustive list of *all possible* security-sensitive operations; rather, it is a set to demonstrate the feasibility of our approach. DMARacer can be easily extended to hook into more security-sensitive operations.

6.3.3 Taint Clears. A known challenge of DTA is avoiding taint explosion—i.e., the accidental spilling of taint into data that is not intended to be tainted. If unaddressed, we may incorrectly report certain non-vulnerable operations as vulnerable. The ideal solution for avoiding taint explosion is to perfectly model all possible dataflows in the program. However, doing so for every possible operation in a program is a difficult task [60].

Instead, we follow an approach used by other DTA systems [21, 38, 45] and clear taint at various operations to mitigate this issue. In general, our taint policies aim to be conservative to reduce false positives. That is, we clear taint unless there is a likely dataflow for a certain operation or code pattern.

Bit-level accesses. It is not uncommon for the kernel to access specific *bits* of DMA data (e.g., a 1-bit flag). In such a case, the kernel may e.g., load a one-byte word from DMA, then perform a bitwise `AND` to isolate the particular bit. Unfortunately, *bit-level* taint analysis—which could be used to accurately model such a dataflow—is known to be difficult [60].

Hence, to avoid false positives arising from such a dataflow, we instead clear its taint. Specifically, we modify the taint analysis' `AND` instruction callback—which normally propagates the taint from its inputs to its output—to instead clear the taint of its output.

Flows to frequently-accessed global objects. Certain parts of the kernel (e.g., the timer subsystem and the slab allocator) are called frequently throughout the kernel's execution and handle globally-accessible data structures. Hence, if taint passes into one of these kernel "hot paths", it soon spills into unrelated operations throughout the kernel. In principle, a heavyweight constraint solver could address this (e.g., by only propagating verifiably controllable dataflows to these hot paths). However, in practice, such approaches do not scale to the size and complexity of the Linux kernel.

Therefore, to avoid false positives arising from such dataflows, we adopt an approach from an existing kernel DTA system [38]. Specifically, we clear taint for all operations and accessed memory within various kernel hot paths[3].

Flows across execution contexts. The kernel maintains various data structures (e.g., sockets and file descriptors) that persist state from one execution context to another. If taint flows into one of these structures in one execution context (e.g., a timer interrupt), and is used in another execution context (e.g., some syscall), it is often difficult to reproduce the dataflow, as it may rely on non-deterministic (e.g., timing) constraints.

Hence, we maintain a notion of taint *validity*, which enforces same-domain dataflows using the following semantics: (i) An execution context begins with all taint labels marked as *invalid*. (ii) If a taint source (i.e., an errant access) is covered, its taint label is marked as *valid*. (iii) A taint sink only reports vulnerable operations if its taint label is *valid*. As a result, we ensure that the vulnerable operations we identify are accurate and easily-reproducible.

7 Implementation

Figure 3 presents the workflow of DMARacer, which: (i) takes the Linux kernel and our runtime library, (ii) builds them with our LLVM instrumentation, and (iii) runs the instrumented kernel as a VM in our custom QEMU hypervisor, where it identifies vulnerable DMA race conditions at runtime.

Runtime library. Our runtime library consists of: (i) the KernelDataFlowSanitizer (KDFSAN) [21] runtime library, which provides support for DTA (e.g., by creating taint labels, combining labels, etc.); and (ii) the handlers for DMARacer's various *instruction*-level and *function*-level callbacks, which collectively identify vulnerable DMA race conditions.

Specifically, the DMARacer runtime library handles the following function-level callbacks: First, for *DMA operations*, it tracks the state of DMA regions. Second, for *execution context entries and exits*

[3]The semantics are similar to those of KMSAN's `__no_kmsan_checks` function attribute; indeed, we clear taint in many of the same functions where it is applied.

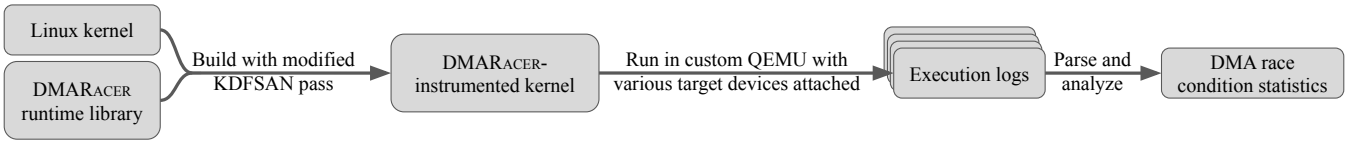


Figure 3: DMARacer's workflow: Take the Linux kernel, identify DMA race conditions, and present statistics to aid mitigation.

(e.g., `__enter_from_user_mode()`, `irq_enter_rcu()`), it: (i) clears the local access map and (ii) marks taint labels as invalid. Third, for *assertions* (e.g., `BUG_ON()`), it performs the taint sink for vulnerable asserts.

Furthermore, it handles the following instruction-level callbacks: First, for *memory accesses*, it: (i) records DMA accesses, (ii) performs the taint source for errant accesses, and (iii) performs the taint sink for vulnerable writes. Second, for *backward conditional branches*, it performs the taint sink for vulnerable loops. Third, for `AND` *operations*, it clears the output's taint.

LLVM instrumentation. We build the kernel with the KDFSAN pass, which we modified to add the above instruction-level callbacks. First, to hook memory accesses, we reuse KDFSAN's callbacks for `LOAD`, `STORE`, and `MEMTRANSFER` instructions. Second, to hook backward conditional branches, we modify KDFSAN's conditional `BranchInst` visitor to check its direction (via the `PostDominatorTree` API), and if it is backwards, to add our callback. Third, to hook `AND` operations, we modify KDFSAN's `BinaryOperator` visitor to add our callback for `AND` instructions.

8 Evaluation

In this section, we evaluate DMARacer's ability to detect DMA race conditions.

8.1 Setup

Environment. We perform our evaluation on a host machine with an AMD Ryzen 9 3950X CPU and 128GB of RAM, running Ubuntu 22.04.4 LTS (kernel v6.8). We instrument the kernel (based on Linux v6.5.8) with a modified KDFSAN pass (based on LLVM v11.0.1), and run the instrumented kernel as a guest VM in QEMU.

Device emulation with QEMU. Dynamic error detectors require that the code is executed to be analyzed. In the case of DMARacer, this means that a driver needs to allocate a DMA buffer and then violate one of the invariants. Additionally, the detection of vulnerable operations relies on tainted data from DMA to spread across the kernel.

Device driver code can only be successfully executed if their respective device is connected. Because we do not have access to the multitude of physical devices supported by the Linux Kernel, we instead decided to use the virtual device emulation feature of QEMU for our evaluation. This emulation feature allows QEMU to mimic a connected peripheral which includes the kernel-device communication via DMA. This emulation is only available for QEMU devices for which the QEMU developers implemented a backend. The backend is responsible for communicating with the running kernel in the same way as the real hardware would. Table 4 shows a summary of the 147 QEMU devices we used in our evaluation. These 147 devices include all QEMU devices except CPUs and various test devices (e.g., devices that are only used for educational purposes).

Evaluation workloads. For each emulated QEMU device we decided to evaluate, we also needed a way to exercise the DMA communication of the device and spread the data from DMA across the kernel. Unfortunately, standard kernel fuzzing techniques such as syzkaller are not suitable for this purpose for two reasons. First, they focus on general code coverage which is not directly relevant for DMARacer. For example, just covering a corner case in a system call handler will never lead to a report from DMARacer unless the code accesses DMA memory or handles tainted data from DMA. Second, some driver logic depends heavily on input from DMA devices itself (e.g. drivers for input devices), and fuzzers like syzkaller focus on the unrelated user-kernel interface.

To overcome this, we instead decided to manually create workloads that are tailored for each device. Table 4 provides a summary of each workload. In general, we mostly use shell scripts that randomly invoke shell commands specific to the device for about 5 minutes per device. For example, the workload for storage devices consists of a shell script that randomly manipulates files, file contents and folders on the mounted storage device. The workload for human interface devices was implemented by using a custom version of QEMU that produces random mouse, touchpad and keyboard presses. We again use QEMU's virtual device emulation feature for this and emit the same internal input events that QEMU creates when being used with a graphical frontend. The emulation backend then translates these generic QEMU input events into device-specific device-to-kernel communication. For the network devices, we start an HTTP server on the host machine and then send random HTTP requests from the QEMU guest.

8.2 Overall Results

Table 3 presents the race conditions found by DMARacer, categorized by: (i) the number of DMA regions affected by errant accesses, (ii) the number of errant accesses, and (iii) the number of vulnerable operations. We group DMA regions based on the *calltrace* of their mapping operation, as this reflects the number of unique *objects* a developer may need to address. This grouping is independent of any intermediate DMA mapping APIs (e.g., `dma_pool_alloc()`), which may ultimately perform a single low-level mapping operation (e.g., via `dma_alloc_attrs()`). Conversely, we group errant accesses and vulnerable operations based on the offending *instruction*, representing the number of unique *lines of code* that a developer would need to fix. It should be noted that the data flow from errant accesses can span multiple files, which means that the vulnerable operations in one row are not necessarily caused by the errant accesses in the same row (see Section 8.3.2).

Table 3: DMA race conditions found.

Kernel Source	Streaming DMA		Coherent DMA				
	Aff. Regions	Errant Accesses	Aff. Regions	Errant Accesses	Vuln. Writes	Vuln. Loops	Vuln. Asserts
block/*	-	-	-	-	32	2	1
drivers/ata/*	3	-	1	2	49	2	-
drivers/hid/hid-core.c	-	-	-	1	-	-	-
drivers/hid/usbhid/hid-core.c	-	-	-	-	1	-	-
drivers/net/ethernet/amd/pcnet32.c	2	-	3	10	-	-	-
drivers/net/ethernet/dec/tulip/*	3	-	1	8	-	-	-
drivers/net/ethernet/intel/e100.c	2	10	2	35	37	1	-
drivers/net/ethernet/intel/e1000/e1000_main.c	3	-	2	7	-	-	-
drivers/net/ethernet/intel/e1000e/netdev.c	-	-	2	8	-	-	-
drivers/net/ethernet/realtek/8139cp.c	-	-	1	10	-	-	-
drivers/net/vmxnet3/*	1	455	5	30	-	-	-
drivers/pci/msi/msi.c	-	4	-	-	-	-	-
drivers/scsi/*	-	-	-	-	46	-	1
drivers/tty/serial/serial_core.c	-	-	-	-	-	1	-
drivers/usb/core/*	28	-	3	-	-	-	-
drivers/usb/host/*	-	8	16	119	18	-	-
fs/ext4/*	-	19	-	-	-	-	-
fs/fs-writeback.c	-	-	-	-	2	-	-
fs/kernfs/dir.c	-	-	-	-	-	1	-
kernel/dma/*	-	4	-	-	16	3	5
kernel/printk/printk.c	-	-	-	-	2	-	-
kernel/sched/*	-	-	-	-	34	5	-
kernel/workqueue.c	-	-	-	-	4	-	-
lib/*	-	8	-	-	58	6	2
mm/dmapool.c	-	-	-	7	-	-	-
net/core/*	-	68	-	-	1	-	-
net/ipv4/*	-	-	-	-	9	-	1
net/ipv6/addrconf.c	-	-	-	-	1	-	-
security/keys/key.c	-	-	-	-	-	1	-
sound/core/*	-	-	1	-	2	-	-
sound/hda/hdac_stream.c	-	-	-	1	-	-	-
sound/pci/hda/hda_controller.c	-	-	-	2	-	-	-
sound/usb/pcm.c	-	-	-	1	-	-	-
Total	42	576	37	241	312	22	10

Table 4: Tested devices and workloads.

Device Kind	# Devices	Workload
Audio Card	13	Playing/Recording audio files
GPU	12	Running OpenGL/GPU test software
Mouse/Keyboard	36	Random input events
Network Adapter	61	Handling random HTTP requests
Storage Device	25	Random file system operations

Our first observation is that the sheer number of errant accesses (817) and vulnerable operations (344) highlights that DMA race conditions pose a significant threat. However, this alone does not fully convey the mitigation effort required: if errant accesses are confined to a *single* DMA region, the fix is relatively straightforward; but if they occur across multiple *distinct* DMA regions, remediation becomes more challenging. By examining the number of DMA regions affected (79), we see that the problem is indeed widespread, indicating that mitigation may be difficult. Our second observation is that the largest count of taint sinks for TOCTOU/TOITOU errors are vulnerable writes. That is, the attacker has at least partial control over the address of a store instruction. As DMARacer is a dynamic analysis tool, it cannot determine all constraints enforced for each write. However, any of these found vulnerable writes is a potential arbitrary write primitive which can corrupt kernel memory and lead to privilege escalation. In Sections 8.3 and 8.4, we draw more insights by breaking down the numbers based on the source file and type of DMA.

8.3 Coherent DMA-based Race Conditions

Our analysis of coherent DMA-based race conditions reveals two notable findings: (i) most vulnerabilities stem from drivers managing complex data structures within DMA regions, which unfortunately makes exploitation easier and mitigation more challenging; and (ii) most dataflows to vulnerable operations occur across kernel subsystems, making them difficult to track without DMARacer's DTA-based approach. To illustrate each finding, we present case studies that share a common theme: a vulnerable high-level API, which implies that any driver that uses the API may also be vulnerable.

8.3.1 TOITOU-vulnerable data structures. Our analysis reveals that although just over half (57%) of the identified errant accesses are TOITOU bugs—as opposed to TOCTOU bugs—the vast majority of the resulting vulnerabilities (99%) depend on TOITOU-loaded data. Upon manual examination, we attribute this disparity to the

fact that TOITOU bugs typically stem from the kernel managing critical data structures within DMA regions. Consequently, the kernel performs complex operations on these data structures, many of which can be exploited.

For example, it is common to find entire linked lists—with pointers and all—managed in DMA. We observe such DMA-resident lists throughout the kernel code, e.g.: the UHCI driver maintains a list of "queue headers"; the E100 driver maintains a list of "callbacks"; and the DMA pool API maintains a list of "blocks" (which we will discuss below). Whenever the kernel loads a pointer from these linked lists, it loads an attacker-controllable pointer. Thus, when the kernel dereferences it (e.g., to traverse the list), it performs an attacker-controllable memory access.

In contrast, TOCTOU-based bugs are typically more limited, as they involve simpler, less consequential data structures. For example, a common source of a TOCTOU bug is reading a status flag multiple times. If an attacker corrupts such a status flag, the impact is limited because it usually does not allow control over critical operations, such as the pointer used in a memory write.

Furthermore, we find that TOITOU bugs are more concerning than TOCTOU bugs for two main reasons. First, they are *easier to exploit*, as they involve the corruption of long-lived data. Conversely, exploiting TOCTOU bugs requires corrupting short-lived data within a narrow time window, forcing attackers to employ synchronization tricks to precisely time the corruption [2, 55, 56]. Second, they are more *difficult to mitigate*, as they affect critical data structures, and may therefore require extensive algorithmic rewrites of driver code. Conversely, TOCTOU bugs can often be resolved with minor, localized changes (e.g., combining two reads into a single read).

Case study: DMA pool API. A striking example of a TOITOU-vulnerable linked list is in the widely used DMA pool API. A DMA pool aims to reduce the overhead of coherent DMA allocations, which are resource-intensive operations. It achieves this by creating a large coherent DMA buffer—the "pool"—from which smaller buffers are allocated as needed.

Fundamentally, a DMA pool functions like a heap: it is a structure composed of linked memory "blocks", which, in this context, are DMA buffers. When a driver employs a DMA pool, it grants the device access not only to these blocks but also to the pointers linking them. Consequently, similar to traditional heap corruption vulnerabilities—where a malicious program corrupts heap metadata to e.g., hijack control flow [9, 23]—a TOITOU bug allows a malicious device to corrupt DMA pool metadata, which can trivially lead to arbitrary kernel memory corruption from any driver that uses it, as illustrated by Figure 4.

Unfortunately, because the DMA pool API is extensively used, this vulnerability is not confined to a single instance. In fact, every usage of the DMA pool API is potentially vulnerable. Each of these instances could lead to arbitrary memory corruption, highlighting the critical importance of addressing this issue.

8.3.2 Cross-subsystem dataflows. Coherent DMA-based race conditions frequently involve dataflows across different kernel subsystems, making them challenging to detect. Typically, both the mapping of a coherent DMA region and any errant accesses to it occur within the same file, usually within the driver. However, the

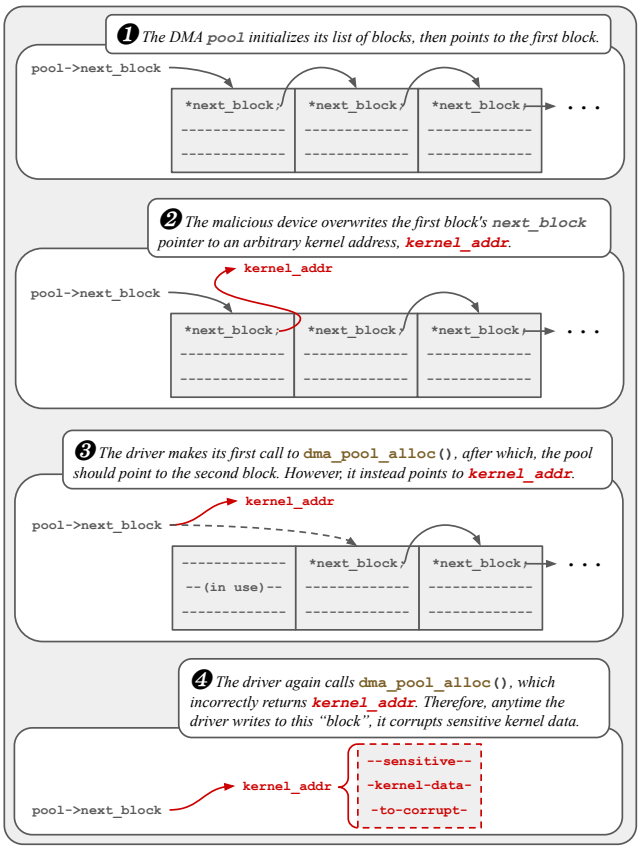


Figure 4: DMA pool exploitation: The DMA pool API initializes a linked list in coherent DMA, which a device can exploit to corrupt arbitrary kernel memory.

data loaded via DMA often propagates throughout the kernel, and any vulnerable operations that depend on this data may occur in separate files or entirely different subsystems.

This is understandable because drivers manage DMA, but the data they load can flow into other kernel components. Tracking such dataflows between disparate components is difficult for heavyweight analyses (e.g., symbolic execution) because it requires interprocedural and cross-module analysis. In contrast, DMARacer's lightweight DTA engine effectively tracks these dataflows, allowing us to identify many vulnerable operations that might otherwise be missed.

Case study: Driver-to-swiotlb dataflow. An illustrative example of a cross-subsystem dataflow occurs when a driver improperly saves the bus address of a *streaming* DMA mapping into a *coherent* DMA region—thereby making it attacker-controllable—then uses it in an unmapping operation. We observed this scenario in several drivers (e.g., the Intel E100 NIC driver, RealTek 8139C+ NIC driver), and we present one such occurrence in Figure 5.

This vulnerability is particularly challenging to detect with static analysis due to several factors. First, the three interactions with the coherent DMA region—i.e., the allocation, initialization, and usage—occur in separate syscalls and interrupts, which complicates

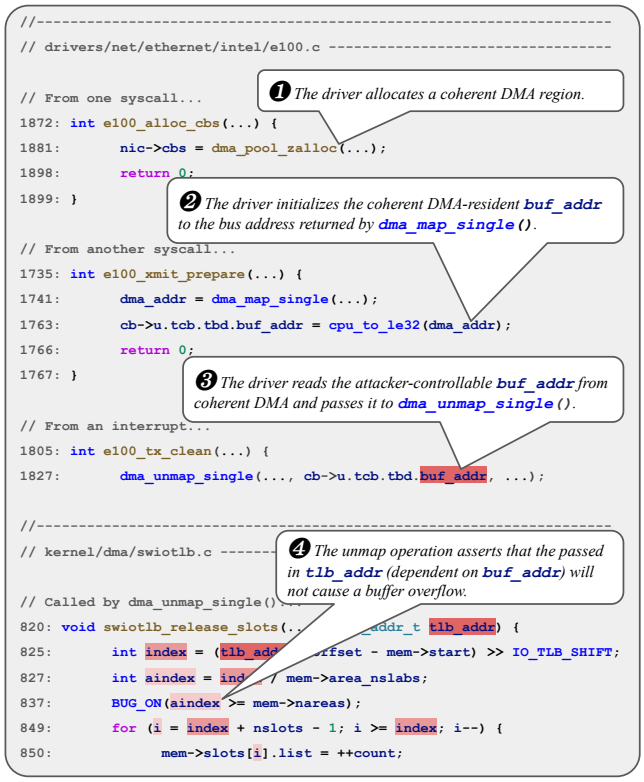


Figure 5: Driver-to-swiotlb exploitation: A driver saves a (mapped) bus address to coherent DMA, and later passes it to an unmapping operation with a vulnerable assertion, which a malicious device can exploit to cause a DoS.

control flow analysis and requires modeling asynchronous events. Second, the dataflow from the errant access (in `e100_tx_clean()`) to the vulnerable assertion (in `swiotlb_release_slots()`) crosses several subsystems: from the driver, to the mapping API, then to the swiotlb (i.e., "software I/O translation lookaside buffer") API. Even along this single control path, there are multiple intermediate indirect calls (e.g., the driver may use custom mapping operations via function pointers), which are notoriously difficult for static analysis to resolve. Consistent with this, our review of SADA's reported findings (Appendix B) suggests that the vulnerable operations it flags are typically contained within a single function, whereas the bug here spans multiple functions and subsystems.

This cross-subsystem dataflow raises a crucial question: *Which subsystem is responsible for mitigation?* On one hand, the driver should not save a bus address in an attacker-controllable DMA region and expect it to remain uncorrupted. On the other hand, the swiotlb API should handle errors more gracefully than by invoking a kernel panic[4]. Given the widespread nature of such vulnerable DMA-based race conditions, we propose that mitigation efforts should occur at both ends.

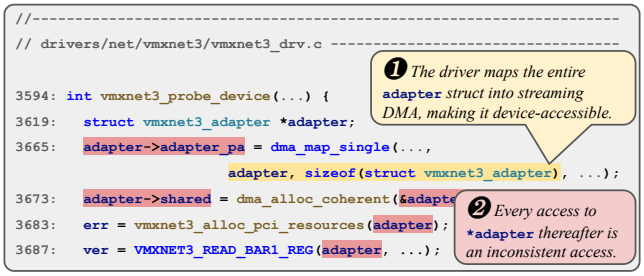


Figure 6: A misunderstanding of the rules regarding DMA synchronization leads to hundreds of inconsistent access in the VMXNET3 NIC driver.

8.4 Streaming DMA-based Race Conditions

Our analysis of streaming DMA-based errant accesses reveals two significant findings: (i) many are caused by developers seemingly misunderstanding the DMA mapping API; and (ii) many involve improper synchronization across entirely different kernel subsystems, making them difficult to detect without DMARacer's dynamic approach. Additionally, we present a case study to highlight the first finding, which also demonstrates a critical insight: a single incorrect synchronization can result in hundreds of errors.

8.4.1 Mapping API misuse. As explained in Section 6.2.2, an inconsistent access bug occurs when the accessed region is immediately preceded by either a device-synchronizing operation or a mapping operation. We found that among all errant accesses identified by DMARacer, only 10 were preceded by a device-synchronizing operation; the remaining 569 were preceded by a mapping operation and lacked any subsequent synchronization before the errant access. Evidence from prior work and commit history indicates that many of these issues stem from developers being unaware of the rules for DMA buffer synchronization—particularly, the assumption that accessing a streaming DMA buffer immediately after mapping it is safe. These findings align with SADA's analysis of inconsistent accesses [10]. Given the widespread use of the streaming DMA and the evident misunderstanding of it, we conclude that misuse of the API is a systemic problem.

Case study: VMXNET3 driver. A striking example of DMA mapping API misuse is found in the VMXNET3 driver, a high-performance NIC driver from VMware. Figure 6 illustrates how a single incorrect synchronization in this driver leads to hundreds of inconsistent access bugs.

During boot time, the driver initializes its data structures and maps the `struct vmxnet3_adapter *adapter` structure—which includes critical fields such as the transmit and receive queues—into a streaming DMA region. Immediately after the mapping operation, on the same line of code, it stores the returned value into the `adapter->adapter_pa` field, thereby committing an inconsistent access. This example highlights developers' apparent misunderstandings of DMA synchronization rules—the bug occurs on the same line as the mapping operation.

Subsequently, within the same function, the driver performs 52 additional inconsistent accesses to this object while initializing its various fields. Proper mitigation would involve deferring the

mapping operation until after initialization and adding synchronization operations when the object transitions between kernel and device access. However, such a mitigation is non-trivial, as it requires defining synchronization points in a driver not originally designed with these considerations. We are currently collaborating with the developers to address these issue, and mitigation efforts are underway.

8.4.2 Cross-subsystem synchronization. Similar to the cross-subsystem dataflows observed in coherent DMA-based race conditions, streaming DMA-based race conditions often involve cross-subsystem dependencies—though in this case, the issue revolves around synchronization rather than data propagation. In streaming DMA, the mapping of the DMA region and any errant accesses often occur in different files or subsystems.

Specifically, apart from those in the VMXNET3 and E100 drivers, all errant accesses involve DMA regions that are mapped in a driver (e.g., the ATA, E1000, and USB drivers), whereas the errant access occurs in a completely different subsystem (e.g., the filesystem or networking subsystems). This is perhaps unsurprising, as drivers are typically responsible for mapping and synchronizing DMA buffers, whereas higher layers of the stack simply access these buffers. Consequently, if a driver fails to synchronize DMA correctly, it affects accesses elsewhere in the kernel.

Because control transfers between these disparate parts of the kernel occur via indirect branches, hardware interrupts, and similar mechanisms, static analysis approaches can struggle to identify such errant accesses. In particular, SADA [10] reports that it does not analyze function pointer calls when building call graphs; consequently, it cannot construct a complete call graph and would miss the kind of cross-subsystem synchronization bugs we observe here. In contrast, DMARacer's dynamic approach effectively tracks DMA state through these control transfers, thereby identifying these synchronization issues.

8.5 Accuracy, Performance, & Coverage

8.5.1 Accuracy. We manually inspected results to estimate DMARacer's false positive (FP) rate. Unfortunately, full inspection is infeasible due to two challenges: First, some errant accesses span multiple execution contexts (e.g., across syscalls or interrupts), making complete tracing impractical. Second, due to a limited taint space (256 colors), taint labels are assigned per instruction rather than per calltrace. As a result, frequently used instructions (e.g., in `memcpy()`) may conflate independent execution paths.

Instead, we manually reviewed the 179 errant accesses and 56 vulnerable operations that we could definitively assess. We found a 0% FP rate for errant accesses and a 9% FP rate for vulnerable operations—both acceptably low. Errant accesses are precise by design, as DMARacer tracks concrete memory accesses to real DMA regions. False positives in vulnerable operations arise from not modeling dataflow constraints.

For instance, in the ALSA audio driver, the kernel reads a previously-initialized index via DMA and returns it to the PCM subsystem. While this appears attacker-controllable, the PCM layer bounds-checks the index and resets it to zero if it exceeds the buffer size. Thus, although the data is tainted, the actual access is safe—demonstrating a FP due to an unmodeled constraint. However, such cases are uncommon in practice: they require explicit checks against known-safe values, which most DMA-handling code does not perform. We further discuss false positives and false negatives in Section 9.

Table 5: Runtime overhead of DMARacer's components.

Enabled Component	Mean		Geomean	
	Overhead	Δ	Overhead	Δ
Default Kernel (Baseline)	0%		0%	
+ KDFSan (Taint Tracking)	194%	+194%	232%	+232%
+ DMA Region Tracking	194%	≈ 0%	232%	≈ 0%
+ Load Monitoring	333%	+139%	488%	+257%
+ Store Monitoring	349%	+16%	508%	+20%
+ Compare Monitoring	400%	+51%	584%	+76%
+ Reporting (DMARacer)	401%	+1%	588%	+4%

Table 6: DMA mapping (or allocation) sites.

Kernel Source	Maps	Maps Covered	Aff. Maps Covered
`drivers/ata/`	13	2	2
`drivers/net/`	589	48	23
`drivers/usb/`	58	16	11
`drivers/` (other dirs.)	932	1	-
`sound/core/`	3	1	1
`sound/` (other dirs.)	7	-	-
`/` (other dirs.)	29	-	-
Total	1631	68	37

8.5.2 Performance. We evaluated the performance of DMARacer using the OS subset of LMBench [32] and measured a mean runtime overhead of 402% across 20 iterations. Additionally, we performed an ablation study using LMBench to understand the overhead incurred by each component. Table 5 shows the incremental and total overhead of each component in LMBench. According to our measurements, the taint tracking logic and load instruction monitoring are the largest contributors to runtime overhead.

In the context of other widely used sanitizers, the overhead of DMARacer is higher than AddressSanitizer [47] (73% to 100% [3]), comparable to that of MemorySanitizer [51] (between 200% to 400% [4]) and faster than ThreadSanitizer (between 400% to 1400% [5]).

8.5.3 Coverage. The findings of this evaluation depend on the DMA-specific code we covered in the kernel. The more DMA-related code we exercise, the more DMA race conditions we can detect. In this section, we analyze how much of the kernel's DMA code we reached with our evaluation workloads.

Table 6 presents: (i) the total number of DMA mapping sites in the *entire kernel*; (ii) the number of DMA mapping sites we *covered*; and (iii) the number of DMA mapping sites we covered that are *affected* by a race condition. Note that, unlike Table 3, which groups DMA regions by mapping *calltraces*, we count mapping *callsites* here to enable direct comparison with the total number of callsites

in the kernel. Additionally, the total number of DMA mapping sites are gathered from source code analysis, so they also include sites that are potentially unreachable with the devices/drivers used in our evaluation setup.

Our first observation is that approximately half of the covered mapping sites (37 out of 68) are affected by a DMA race condition. This high proportion indicates that DMA race conditions are prevalent among the code we exercised. However, this figure represents an upper bound: some mapping sites may be used to map multiple DMA regions, and not all of these regions are necessarily affected—only at least one is.

Our second observation is that we covered only a fraction of the total mapping sites in the kernel (68 out of 1631). We attribute this mainly to the relatively low number of devices that can be emulated by QEMU (see also Section 9). Additionally, several of 147 devices we emulated with QEMU use the same driver and therefore share a mapping site. The reason for this is that they are either similar devices (e.g., the `i82557a` and `i82557b` network adapters) or follow a standard communication protocol (such as USB human interface devices).

9 Discussion & Limitations

Completeness. DMARacer may incur false negatives for two reasons. First, since KDFSAN does not track implicit data flows, DMARacer may miss vulnerabilities that rely on them. We consider this preferable to the alternative approach of approximating implicit flows, which can be a source of over-tainting and thus false-positives [24]. Second, our taint policies aimed at reducing false positives (Section 6.3.3) may occasionally result in false negatives. For instance, a vulnerability dependent on bit-level dataflows might be missed due to our policy of clearing taint for `AND` instructions.

A possible alternative to removing taint is to use bit-precise taint tracking. This removes the need to completely clear taint after bitwise operations, and reduces false-positives caused by overtainting due to DMARacer's *byte*-accurate tainting approach.

Soundness. DMARacer may also produce false positives where DMA race conditions exist, but the code explicitly defends against abuse. This is primarily because, like any DTA system, DMARacer does not fully model all dataflow constraints. For example, if a driver verifies each value it loads from DMA against an expected value, DMARacer would not capture this constraint. However, manual inspection reveals that such cases contribute to only a few false positives. Addressing this inherent limitation would require heavyweight constraint solving, which we intentionally avoided to keep our prototype scalable and practical, encouraging its real-world adoption.

Access to devices. Unlike static analyzers, DMARacer requires that each driver can be executed and perform its normal communication with its respective device. This also means that the device has to be emulated or physically connected to the system. The rise of device emulation with tools such as QEMU alleviates this issue to some degree. However, creating these drivers requires in-depth understanding of how each device works, as the kernel-device communication has to be recreated exactly in the emulation layer. It is therefore possible that some devices can never be emulated by QEMU, and therefore DMARacer is unable to find DMA errors in their respective drivers unless the physical device is available.

Comparison to Static Analysis. Compared to a static analysis approaches such as SADA [10], DMARacer can detect DMA misuse across long execution traces. DMARacer can also analyze traces that involve asynchronous completions, indirect function calls or control flow based on hardware-triggers (see Sections 8.3.1 and 8.3.2).

10 Related Work

DMA errors. Previous work investigated the security implications of communicating with peripheral devices using DMA. [31] and [8] characterize possible DMA attack scenarios in the presence of various IOMMU protections. This also includes the accidental exposure of OS-internal data to the device, which is a superset of the TOITOU bugs that are detected by DMARacer. SADA [10] proposes using static analysis to search code for various DMA bugs, which includes the TOCTOU bugs and other errant accesses detectable by DMARacer. Other work is concerned with dynamic detection of race conditions in the DMA memory controller itself [26, 42]. These race conditions are a superset of the ones detectable by DMARacer.

Double fetches. Existing work has studied double fetches and proposed detection approaches by e.g., using static analysis [28, 53, 54, 58]. Other approaches are based on dynamic analysis of memory accesses. DECAF [46] is a dynamic detection tool using modern CPU features and hardware side-channels to detect memory accesses that indicate double fetch bugs. Bochspwn [22] is a dynamic detection tool that simulates the target OS and searches the simulated memory accesses for indications of double fetches. It should be noted that the attack vector studied by previous work on double fetches is mainly concerned with the system call interface. There are no DMA-based memory accesses in this scenario and instead the memory is manipulated by a malicious thread running in user space. However, the general issue of double fetching from untrusted memory is the same in DMA-based drivers and system call handlers. The mitigations proposed for double fetches in system handlers [11, 15] are in theory also applicable to the DMA interface in drivers. However, whether their overhead is practical in the context of DMA-based device drivers is unclear.

Race conditions. The problem of detecting race conditions in concurrent programs has a significant body of research covering it. This includes approaches based on static [16, 39] and dynamic analysis [19, 44, 48] which verify the proper use of synchronization primitives within a program. However, there are no viable synchronization primitives when interacting with coherent DMA, so the applicability of these approaches to DMA data races is limited.

Kernel race detectors. Other previous work focuses on detecting data races within OS kernels [6, 17] by observing values read from memory locations via memory watchpoints. The concurrent read of different values from the same address is here used as an indication of a race condition. Because these tools do not rely on observing synchronization primitives, they can detect when peripheral devices and the kernel concurrently access memory via DMA. However, as

the information is limited to a single point in time within a thread's execution, this information is not suitable to detect TOCTOU errors.

Kernel fuzzing. Sanitizers like DMARacer provide reliable bug detection for fuzzers which drive the dynamic analysis. Previous work has proposed a wide variety of solutions for performing random testing on device driver code. This includes fuzzing approaches specific to USB [20, 37, 59], mobile devices [40, 52] or peripherals using DMA [33]. Other fuzzing methods improve the generating coverage by more accurately simulating the real interface behavior of peripheral devices [14, 18, 30, 61].

11 Conclusion

Modern OS kernels use direct memory access (DMA) to efficiently communicate with untrusted peripheral devices. However, when DMA is used incorrectly, it can lead to potentially dangerous race conditions within the kernel. In this paper, we propose a dynamic detection approach named DMARacer that detects race conditions caused by the incorrect usage of DMA buffers in kernel drivers. DMARacer tracks all DMA-based memory accesses and checks them for various kinds of DMA-specific race conditions. DMARacer can also estimate the security impact of the detected bugs. This is done by taint tracking the data from problematic memory reads and pinpointing vulnerable code that operates on this data. We applied DMARacer to the drivers in the Linux kernel and found 817 problematic memory accesses and 344 vulnerable operations. This suggests that DMA-based race conditions in driver code are a systemic issue.

Disclosure. We disclosed our general findings, the three case studies and possible mitigations, to the Linux kernel on November 13, 2024. Our DMA pool API case study was considered valid, but our proposed fix had a too large API change and performance impact to be practical. The Driver-to-swiotlb case study was acknowledged and has been addressed in recent kernel versions. The VMXNET3 case study received no response after reporting it to the developers. No CVEs were assigned for the reported issues.

Acknowledgements

We thank the anonymous reviewers for their valuable comments, Kaan Kara for helping hook the various DMA operations, and Victor Duta for helping port KDFSAN to a recent kernel. This work was funded/co-funded by the European Union (ERC, *Ghostbuster*, 101141972) and further supported by its Horizon Europe programme under grant agreement No. 101120962 ("Rescale") and by the NWO through project *InterSect*, project *Theseus*, and the Gravitation *CiCS* project grant 024.006.037. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union, the European Research Council or any of the funding agencies. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] 2015. syzkaller. https://github.com/google/syzkaller.
[2] 2016. CVE-2016-5195. https://nvd.nist.gov/vuln/detail/CVE-2016-5195.
[3] 2019. Clang 19.1.0 documentation - AddressSanitizer. https://releases.llvm.org/19.1.0/tools/clang/docs/AddressSanitizer.html.
[4] 2019. Clang 19.1.0 documentation - MemorySanitizer. https://releases.llvm.org/19.1.0/tools/clang/docs/MemorySanitizer.html#origin-tracking.
[5] 2019. Clang 19.1.0 documentation - ThreadSanitizer. https://releases.llvm.org/19.1.0/tools/clang/docs/ThreadSanitizer.html.
[6] 2020. KCSAN: Kernel Concurrency Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.
[7] Advanced Micro Devices, Inc. 2007. IOMMU Architectural Specification.
[8] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. 2021. Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU. *EuroSys* (2021).
[9] Anonymous. 2001. Once upon a free(). *Phrack* (2001).
[10] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. 2021. Static Detection of Unsafe DMA Accesses in Device Drivers. *USENIX Security* (2021).
[11] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. 2022. Midas: Systematic Kernel TOCTTOU Protection. *USENIX Security* (2022).
[12] Erik Bosman and Herbert Bos. 2014. Framing Signals—A Return to Portable Shellcode. *S&P* (2014).
[13] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers.* O'Reilly Media, Inc.
[14] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. *CCS* (2017).
[15] Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida. 2024. SafeFetch: Practical Double-Fetch Protection with Kernel-Fetch Caching. *USENIX Security* (2024).
[16] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SOSP* (2003).
[17] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. *OSDI* (2010).
[18] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. *USENIX Security* (2020).
[19] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. *PLDI* (2009).
[20] Jisoo Jang, Minsuk Kang, and Dokyung Song. 2023. ReUSB: Replay-Guided USB Driver Fuzzing. *USENIX Security* (2023).
[21] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2022. Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel. *NDSS* (2022).
[22] Mateusz Jurczyk and Gynvael Coldwind. 2013. Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns. (2013).
[23] Michel Kaempf. 2001. Vudo malloc tricks. *Phrack* (2001).
[24] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. *NDSS* (2011).
[25] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. *S&P* (2022).
[26] Michael Kistler and Daniel Brokenshire. 2011. Detecting race conditions in asynchronous DMA operations with full system simulation. *ISPASS* (2011).
[27] Damien Le Moal. 2017. I/O Latency Optimization with Polling. *Vault* (2017).
[28] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. 2018. Untrusted Hardware Causes Double-Fetch Problems in the I/O Memory. *JCST* (2018).
[29] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. 2018. DFTinker: Detecting and Fixing Double-fetch Bugs in an Automated Way. *WASA* (2018).
[30] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. 2022. PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation. *ISSTA* (2022).
[31] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2019. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. *NDSS* (2019).
[32] Larry McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. *ATC* (1996).
[33] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. *S&P* (2021).
[34] Robert HB Netzer and Barton P Miller. 1992. What Are Race Conditions? Some Issues and Formalizations. *LOPLAS* (1992).
[35] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS* (2005).
[36] Elliott I Organick. 1972. *The Multics System: An Examination of its Structure.* MIT Press.
[37] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. *USENIX Security* (2020).
[38] Alexander Potapenko. 2022. Kernel Memory Sanitizer (KMSAN). https://docs.kernel.org/dev-tools/kmsan.html.
[39] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2006. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. *PLDI* (2006).

[40] Ivan Pustogarov, Qian Wu, and David Lie. 2020. Ex-vivo dynamic analysis framework for Android device drivers. *S&P* (2020).
[41] Razvan Raducu, Ricardo J Rodríguez, and Pedro Álvarez. 2022. Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review. *IEEE Access* (2022).
[42] Selma Saidi and Ylies Falcone. 2015. Dynamic Detection and Mitigation of DMA Races in MPSoCs. *DSD* (2015).
[43] Jerome H Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *CACM* (1974).
[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *TOCS* (1997).
[45] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). *S&P* (2010).
[46] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. *AsiaCCS* (2018).
[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. *ATC* (2012).
[48] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler: Compile-time instrumentation for ThreadSanitizer. *RV* (2011).
[49] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. 2022. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. *USENIX Security* (2022).
[50] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. *NDSS* (2019).
[51] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. *CGO* (2015).
[52] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. 2018. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. *USENIX Security* (2018).
[53] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. *USENIX Security* (2017).
[54] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. 2019. DFTracker: Detecting Double-fetch Bugs by Multi-taint Parallel Tracking. *FCS* (2019).
[55] Robert NM Watson. 2007. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *WOOT* (2007).
[56] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. *ESORICS* (2016).
[57] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DevFuzz: Automatic Device Model-Guided Device Driver Fuzzing. *S&P* (2023).
[58] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. *S&P* (2018).
[59] Yiru Xu, Hao Sun, Jianzhong Liu, Yuheng Shen, and Yu Jiang. 2024. Saturn: Host-Gadget Synergistic USB Driver Fuzzing. *S&P* (2024).
[60] Babak Yadegari and Saumya Debray. 2014. Bit-Level Taint Analysis. *SCAM* (2014).
[61] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. 2022. Semantic-Informed Driver Fuzzing Without Both the Hardware Devices and the Emulators. *NDSS* (2022).

Appendix

A LMBench performance data

Table 7 shows the per-benchmark LMBench results of DMARacer compared to an unsanitized kernel.

Table 7: Runtime overhead of DMARacer.

Benchmark	Baseline Time	DMARacer		
		Time	Overhead	Increase
Simple syscall	0.25μs	0.84μs	0.59μs	+234%
Simple read	0.32μs	1.60μs	1.27μs	+392%
Simple write	0.32μs	1.31μs	0.99μs	+315%
Simple stat	0.62μs	5.58μs	4.96μs	+797%
Simple fstat	0.37μs	1.67μs	1.30μs	+353%
Simple open/close	1.21μs	10.75μs	9.54μs	+788%
Select on 10 fd's	0.41μs	1.76μs	1.35μs	+331%
Select on 100 fd's	1.36μs	13.81μs	12.45μs	+917%
Select on 250 fd's	2.90μs	33.53μs	30.63μs	+1055%
Select on 500 fd's	5.53μs	66.52μs	61.00μs	+1104%
Select on 10 tcp fd's	0.49μs	2.41μs	1.92μs	+393%
Select on 100 tcp fd's	3.72μs	40.58μs	36.85μs	+990%
Select on 250 tcp fd's	9.00μs	104.91μs	95.91μs	+1066%
Select on 500 tcp fd's	18.07μs	212.22μs	194.15μs	+1074%
Signal handler installation	0.31μs	1.02μs	0.71μs	+227%
Signal handler overhead	0.76μs	4.95μs	4.19μs	+548%
Protection fault	0.47μs	0.92μs	0.45μs	+96%
Pipe latency	2.61μs	24.31μs	21.70μs	+831%
UNIX sock stream latency	3.17μs	29.94μs	26.77μs	+843%
Process fork+exit	69.86μs	354.08μs	284.22μs	+407%
Process fork+execve	215.01μs	961.15μs	746.14μs	+347%
Process fork+/bin/sh -c	465.10μs	1919.48μs	1454.38μs	+313%
UDP latency	4.05μs	52.10μs	48.05μs	+1187%
TCP latency	5.40μs	87.20μs	81.80μs	+1514%
TCP/IP connection cost	21.18μs	238.17μs	216.99μs	+1024%

B SADA Findings

There is no publicly available code or artifact of SADA, so we could not run a direct comparison against DMARacer. Instead, we provide an analysis of the issues reported by SADA. We found these issues by searching the Linux kernel mailing list for reports and patches that are referencing the paper's authors. The list of found issues is to our knowledge complete except for privately reported and unpatched bugs.

Table 8 shows the 20 patches/reports we found. Some reports describe multiple issues affecting several DMA memory regions. We indicated if this is the case using the third table column. In total, 24 unique bugs were reported that involve a problematic DMA memory access.

The reports do not contain the actual trace that SADA analyzed. We therefore approximated the detection logic using the bug explanation in each report. Specifically, we tried to determine how often SADA finds issues involving long and complex traces.

Column 4 shows whether the vulnerable DMA operations that needed to be analyzed occurred in the same function. For a bad streaming DMA mapping, the involved code piece are the to-device mapping operation and the subsequent memory access. For a TOCTOU error, the vulnerable operations are the check and use. Unchecked DMA access only involve one read operation, so we omitted them in this column. In summary, all found issues had their vulnerable operations contained within a single function. If we consider all DMA operations including the DMA allocation call itself, then 12 of the 20 reports (16 of 24 bugs) have traces that only spanned a single function (Column 5).

The last column indicates whether a patch has been merged or the respective bug was fixed by a maintainer. In total, 10 reports containing 14 bugs were fixed.

Table 8: DMA errors detected by SADA.

Subject Line on Mailing List	Issue Class	Number of Affected Allocations	**Vulnerable** Operations in Same Function?	**All** Operations in Same Function?	Patch Merged?
rtlwifi: rtl8723ae: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
rtlwifi: rtl8192de: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
rtlwifi: rtl8192ce: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
rtlwifi: rtl8188ee: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
p54: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
atm: idt77252: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
atm: eni: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✓
net: vmxnet3: avoid accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✗
crypto: hisilicon: accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**4**	✓	✓	✓
crypto: qat: accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**2**	✓	✓	✗
net: rocker: accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✗
scsi: wd719x: accessing the data mapped to streaming DMA	Bad Streaming DMA Mapping	**1**	✓	✓	✗
media: venus: fix possible buffer overlow casued bad DMA value in venus_sfr_print()	Time-of-Check to Time-of-Use	**1**	✓	✗	✗
media: pci: ttpci: av7110: avoid compiler optimization of reading data[0] in debiirq()	Time-of-Check to Time-of-Use	**1**	✓	✗	✓
scsi: esas2r: fix possible buffer overflow caused by bad DMA value in esas2r_process_fs_ioctl()	Time-of-Check to Time-of-Use	**1**	✓	✗	✗
net: sfc: fix possible buffer overflow caused by bad DMA value in efx_siena_sriov_vfdi()	Time-of-Check to Time-of-Use	**1**	✓	✗	✗
usb: cdns3: fix possible buffer overflow caused by bad DMA value	No DMA Value Check	**1**	–	✗	✗
input: tablet: aiptek: fix possible buffer overflow caused by bad DMA value in aiptek_irq()	No DMA Value Check	**1**	–	✗	✗
usb: storage: alauda: fix possible buffer overflow casued by bad DMA value in alauda_read_map()	No DMA Value Check	**1**	–	✗	✗
net: vmxnet3: fix possible buffer overflow caused by bad DMA value in vmxnet3_get_rss()	No DMA Value Check	**1**	–	✗	✓