Beyond Driver Isolation - Triaging Threats against Driver Isolation

Yongzhe Huang[1], Kaiming Huang[1], Matthew Ennis[1],
Vikram Narayanan[2], Anton Burtsev[4], Trent Jaeger[3], Gang Tan[1]

[1]The Pennsylvania State University - *{yzh89, kzh529, mje5606, gtan}*@psu.edu

[2] Palo Alto Networks - *{vinarayanan}*@paloaltonetworks.com

[3]UC Riverside - *{trentj}*@ucr.edu

[4]University of Utah - *{aburstev}*@utah.edu

***Abstract*—Device driver isolation aims to protect kernels from faulty/malicious drivers, yet its security guarantees are not fully understood. Compartment Interface Vulnerabilities (CIVs), known in userspace applications, also impact driver isolation, but this area is underexplored. This paper surveys existing driver isolation frameworks, systematizes CIV classifications, and evaluates them in the driver isolation context. Our analysis reveals CIV prevalence under a baseline threat model, with large drivers exhibiting over 100 CIV instances and an average of 33 across the studied drivers. Enforcing additional security properties like CFI reduces average CIVs to approximately 28. This work offers insights into driver isolation security, CIV prevalence, and guidance for future systems.**

1. Introduction

Today's Linux 6.5 kernel contains roughly 9,921 device drivers, accounting for 70% of its source code [1], nearly doubling since 2013. Developed by third parties who often lack a complete understanding of the kernel's programming and security idioms, device drivers are a primary source of defects in modern OS kernels [13]. With over 80,000 commits per year, the fast-evolving device driver codebase constitutes the largest attack surface in modern kernels, making up **16–59%** of all reported Linux kernel vulnerabilities since 2014 (Table 1).

Year	Total CVEs	Driver CVEs	Percentage
2014	130	22	**16.9%**
2015	80	15	**18.8%**
2016	216	65	**30.1%**
2017	452	267	**59.1%**
2018	180	44	**24.4%**
2019	291	154	**52.9%**
2020	129	38	**29.5%**
2021	162	42	**25.9%**
2022	310	92	**29.7%**
2023	263	75	**28.5%**

TABLE 1: Linux kernel CVEs versus those in its device drivers.

To prevent device driver flaws from impacting kernel security, *driver isolation* has been proposed, with a history dating back to the 1977 Multics report [36]. This technique runs drivers in an isolated environment, confining faults and preventing single-point failures. While easily achieved through clean-slate kernel redesigns [29], [37], [41], [66], efforts have focused on isolating drivers from monolithic kernels [18], [45], [52], [54], [62], [63]. However, many attempts have failed due to the *performance* and the *complexity* of breaking apart the kernel's monolithic, shared-memory code [20], [45], [52], [54], [63]. Recently, the balance has started to change with efficient hardware isolation mechanisms, such as memory tagging [60] and VMFunc [23], and scalable automated decomposition approaches, such as static analyses for automating kernel-driver code and data separation [21], [33].

Despite recent advances, the security benefits of driver isolation remain unclear. Isolation frameworks limit driver privileges, but restrictions vary based on threat models. For example, Nooks [63] assumes buggy but not malicious drivers, allowing isolated drivers to read kernel pages and run privileged instructions, which cannot confine attacks from malicious or compromised drivers. Understanding the *threat models* and *isolation guarantees* of different frameworks is therefore crucial.

It is also important to understand the attacks that are still possible after isolation. Malicious drivers may still compromise kernel compartments by exploiting the isolation interface, as demonstrated by Lefeuvre et al. [38] in compartmentalized user-space applications. These Compartment Interface Vulnerabilities (CIVs) occur because compartmentalized components may still share/communicate data with a domain that can be used to launch attacks. Chien et al. [14] identify a specific type of CIVs for the kernel-driver interface, showing that malicious isolated drivers can supply corrupted data to kernel memory API calls, compromising the confidentiality and integrity of kernel data. Other researchers have identified additional security issues [7], [8], [14], [38], [50], suggesting more types of CIVs may impact the kernel-driver interface. A comprehensive study of CIV

types and their impact on kernel security is needed.

CIVs pose significant threats to the kernel despite driver isolation, but the effectiveness of attacks using CIVs varies depending on the defenses applied in driver isolation. For instance, a CIV enabling buffer index corruption may succeed in frameworks lacking index validation, but can be mitigated by those implementing shared data validation [59].

Given the variety of types of CIVs, evaluating them under different threat models assuming diverse isolation protections is crucial. Previous studies, such as Conffuzz [38], estimated CIV prevalence based on a basic isolation model with limited security properties, such as memory safety and control flow integrity. Some driver isolation projects, however, have multiple kinds of protections to enforce security properties on the isolation interface. As a result, it is important to evaluate attack viability against more robust driver isolation systems with enhanced security measures.

In this study we aim to answer the following questions: (1) What are the security properties enforced by existing driver isolation frameworks? (2) What are the types and prevalence of CIVs at the kernel-driver isolation interface? (3) How effective are the existing driver isolation frameworks against the various types of CIVs? (4) How do security properties enforced by software hardening techniques, such as memory safety or CFI, reduce the threat of CIVs?

For answers, we provide a systemization of knowledge (SoK) of the attack surface and defenses at the kernel-driver interface and make the following contributions:

- We systemize the threat models and security properties of existing driver isolation frameworks.
- We summarize and augment existing CIV classifications [14], [38], [50] with general and new driver-isolation-specific CIV classes. We describe these new classes with concrete examples.
- We develop static analyses to classify and quantify CIVs for a set of representative drivers.
- We evaluate CIVs under a baseline threat model that only assumes basic driver defenses and security properties. And we discuss how enforcing additional security properties may help in mitigating CIVs.

Target interface We focus on classifying CIVs at the kernel/driver isolation interface. However, the threat of CIVs goes beyond and applies to any compartmentalized software, including isolating other kernel components (e.g., the file subsystem) and isolation in user-space programs. Our study focuses on device driver isolation for several reasons. (1) Kernel-driver interfaces are the de-facto isolation boundary by the majority of previous kernel isolation frameworks due to the high number of vulnerabilities and required privileges in drivers. (2) Device drivers normally have clear compartment interfaces to communicate with the kernel, making analysis tasks easier, as we will discuss in detail later in Section 2. (3) The driver and kernel interfaces normally pass large and complex objects, typically containing heterogeneous types of data, making observations extracted by studying device driver compartmentalization valuable for compartmentalization in other domains with less complex interface data. While we leave the study of CIVs on other interfaces to future work, we believe that the conclusions drawn in this study are generalizable to other kernel and user-space interfaces.

2. Driver Isolation Overview

This section reviews existing driver isolation frameworks for monolithic operating systems, classifying them by their core isolation techniques. Our scope excludes microkernel-based designs [22], [29], [37], [41] and systems focused on isolating general kernel modules [25], [47], [55]. We have identified twenty impactful projects in driver isolation from major OS and security venues over the past two decades. These systems are summarized in Table 2.

2.1. Sandbox architecture

All driver isolation frameworks assume a *sandbox* [38] architecture, in which the kernel is trusted and the driver is not. This is opposite to the *safebox* architecture [4], [12], [16], [25], [38], which instead assumes the isolated component is trusted and should be protected from the rest.

2.2. Driver threat models

All the driver-isolation projects we have surveyed assume two types of threat models:

1) **Buggy**: Many driver isolation frameworks [21], [39], [52], [54], [56], [63], [70] assume that a driver may contain bugs that can cause the entire kernel to malfunction or hang. This threat model primarily focuses on the availability of kernel services, rather than considering an active attack scenario. The goal of isolation in this context is to prevent a buggy driver from crashing or hanging the kernel, ensuring that other kernel services remain available.
2) **Exploitable**: Device drivers are commonly assumed to contain bugs that can be exploited by an attacker to compromise the kernel [6], [8], [10], [18], [45], [60], [67], [68]. These bugs often allow the attacker to gain powerful read or write primitives, which can be used to compromise the confidentiality or integrity of the kernel data. Isolation mechanisms aim to confine the impact of such exploits to the isolated driver, preventing the attacker from compromising the kernel.

2.3. Driver isolation boundaries

Traditionally, the device driver has two natural boundaries. The first is for *driver/kernel* interaction, and the other is for *driver/device* interaction. The kernel/driver boundary is formed by a set of kernel functions imported by the driver, including kernel and device libraries, kernel services, etc. [35], and a set of functions the driver exports to the kernel to extend the kernel's functionality via a set of function pointers and the `EXPORT_SYMBOL` macro. This natural boundary is used as the de facto isolation boundary between the kernel and a driver, with a few exceptions where

only a subset of the driver code is isolated [21], [49], [56]. The systems that choose a different isolation boundary do so to address limitations of traditional isolation, such as performance. For example, Microdrivers [21], decaf [56], and Twindrivers [49] leave performance critical driver code inside the kernel to reduce domain switching overhead, achieving better performance.

The driver/device boundary is defined by operations that interact with the device hardware, including access to memory-mapped I/O (MMIO) regions, I/O ports (on x86 architectures), Direct Memory Access (DMA), etc. [35]. By abusing the interaction with the device, a malicious driver could potentially exploit the driver/device interface to circumvent the isolation. For example, abusing DMA may allow the driver to access any kernel memory through the device. To mitigate this risk, some isolation projects [6], [52], [54] prevent devices from accessing arbitrary physical memory via DMA using the IOMMU. While studying CIVs for the driver/device interface is important, the device states that can be corrupted by the driver are highly device-dependent, e.g., each device has its own set of registers and configurations. Therefore, we leave the study as future work.

2.4. Interface data

Even under isolation, drivers and kernels still communicate by sharing data through the kernel/driver interface, typically via the arguments to and return values from invocations to interface functions or through global variables. However, not all data that could be referenced by the pointers passed across the interface needs to be shared for correct execution. For instance, when a reference to a complex structure is passed through an interface function, it may be the case that only a small subset of the structure's fields are actually needed. If a driver isolation framework imposes no restriction on the data synchronization, it may lead to the problem of *oversharing*. Various driver isolation projects propose solutions to address this issue. Projects based on object copying [6], [21], [52], [54], [56] choose to synchronize only a subset of structured object fields required for correct execution, primarily for improved performance. The required fields are normally computed via manual efforts [6], [52], or static analyses [21], [33], [56]. SFI based techniques [10], [18], [45] allow developers to specify access capabilities at fine-grained levels, e.g., a byte or field levels, to prevent access on overshared data. Despite these efforts, addressing oversharing issues remains challenging.

2.5. Driver isolation security properties

Because they assume different threat models, prior driver isolation frameworks differ in the security properties that they enforce. Below, we collect a set of security properties that are enforced by those driver isolation frameworks.

P1: Kernel data integrity This property specifies that data stored in the memory of the kernel compartment cannot be directly updated by the driver compartments.

P2: Kernel data confidentiality This property specifies that data stored in the memory of the kernel compartment cannot be directly read by the driver compartments.

P3: Interface data integrity This property ensures integrity of data exchanged through compartment interfaces. Even with P1 enforcement, kernel and drivers must share data as discussed in Section 2.4. If interface data is corrupted or violates expected invariants, the kernel becomes vulnerable. We discuss how failing to maintain this property leads to multiple types of CIVs in Section 4.

P4: Interface control flow integrity A driver's control flow must remain confined within its code, except for invocations of predefined kernel interface functions and returns to the kernel call sites of driver functions.

Properties P1-P3 focus on data flow, while P4 addresses control flow. Among these, P1 and P4 are enforced by all driver isolation frameworks, as they directly address the main goal of driver fault isolation across various threat models. They prevent drivers from corrupting kernel data directly and executing arbitrary kernel code. The remaining properties (P2 and P3) are primarily considered in frameworks that assume potentially malicious drivers. P2 prevents driver from reading sensitive information from kernel, while P3 ensures that all the shared data, even allowed to be legitimately updated by driver, cannot be corrupted , preventing the injection of malicious or erroneous values. In addition to aforementioned properties, some isolation frameworks extend security to the driver:

P5: Driver control flow integrity This property is stronger than P4, specifying that the driver's control flow must adhere to a precomputed control flow graph. It constrains the usage of kernel interfaces, mitigating potential attacks such as use-after-free vulnerabilities [10], [38], [57] that could arise from arbitrary invocation sequences of kernel-driver interface functions, even after P4 is enforced.

P6: Driver memory safety Memory safety can be introduced to drivers in multiple ways: 1) utilizing memory-safe programming languages, such as Rust [24], [53] or Java [56], or 2) extending the C type system to prevent memory errors using compile-time or run-time checks [70].

3. Driver Isolation Frameworks

In this section, we examine various driver isolation techniques and analyze the security properties they enforce. We delve into the implementation details of these isolation techniques to understand how they enable specific security properties. Table 2 provides a comprehensive overview of the frameworks that we have investigated. For each security property, we indicate whether it is enforced, and, if so, we describe the corresponding implementation mechanism.

3.1. Software-based Fault Isolation (SFI)

SFI-based techniques offer driver isolation without hardware support by establishing distinct protection domains

Framework	Venue	TM	Integrity		KData Conf.	ICFI	DCFI	MS	Isolation Technique	OS
			KData	IData						
Nooks [63]	SIGOPS EW'02	B	PT	N/A	N/A	WF	N/A	N/A	MMU	L
VM DrivReuse [39]	OSDI'04	B	GVA	N/A	GVA	Translation	N/A	N/A	VM (Xen)	L
SafeDrive [70]	OSDI'06	B	MS	N/A	MS	MS	N/A	Bounds	Language	L
XFI [18]	OSDI'06	E	Instr	N/A	N/A	WF	CFI guards	N/A	SFI	W
Microdrivers [21]	ASPLOS'07	B	PM	SC	PM	IPC	N/A	N/A	User-mode	L
TwinDrivers [49]	ASPLOS'09	B	SVM	N/A	SVM	PE	N/A	N/A	VM (Xen)	L
Decaf [56]	ATC'09	B	PM	SC	PM	IPC	N/A	N/A	User-mode+Safe Lang	L
Microdriver+ [8]	ACSAC'09	E	PM	SC	PM	IPC	N/A	N/A	User-mode+Daikon	L
BGI [10]	SOSP'09	E	Instr	Cap	Cap	WF	CFI guards	Byte-CAP	SFI	W
SUD [6]	ATC'10	E	PM	N/A	PM	IPC	N/A	N/A	User-mode	L
HUKO [68]	NDSS'11	E	MAC	N/A	MAC	PE	N/A	N/A	VM (Xen)	L/W
LXFI [45]	SOSP'11	E	CAP	N/A	CAP	CAP	CAP+SS	N/A	SFI	L
SIDE [62]	DSN'13	B	PM	N/A	PM	WF	N/A	N/A	User-mode	L
LXDs [52]	ATC'19	B	GVA	SC	GVA	IPC	N/A	N/A	VM (KVM)	L
LVDs [54]	VEE'20	B	GVA	SC	GVA	IPC	N/A	N/A	VM (Bareflank)	L
IskiOS [25]	RAID'21	E	PKU/PKK	N/A	N/A	N/A	SS	N/A	Protection keys (PKK)	L
KSplit [33]	OSDI'22	B	GVA	SC	GVA	RPC	N/A	N/A	VM (Bareflank)	L
HAKC [48]	NDSS'22	E	MTE	N/A	MTE	PAC policy	N/A	N/A	MTE + PAC	A
DriverJar [67]	DAC'23	E	TR	N/A	N/A	PE	N/A	N/A	Hardware watchpoint	A
Sfitag [60]	Asia CCS'23	E	Tag	N/A	Tag	WF+tag	N/A	N/A	Memory tagging	A
Bulkhead [26]	NDSS'25	E	PKS	N/A	PKS	PE	N/A	N/A	PKS	L

TABLE 2: Existing driver isolation frameworks, their threat models (B: Buggy, E: Exploitable), and enforcement techniques for security properties P1–P6. Integrity is split into Kernel (KData) and Interface (IData). ICFI and DCFI stand for interface CFI and driver CFI. Techniques: SC (Selective copy), PE (Predefined entries), PT (Page table), PM (Privilege Mode), TR (Trampoline), MS (Memory Safety), Instr (Instrumentation), MAC (Mandatory Access Control), SVM (Software Virtual Memory), GVA (Guest Virtual Address), CAP (Capability), SS (Shadow Stack), WF (Wrapper Function). OS: L (Linux), W (Windows), A (ARM-based).

within a shared address space using static analysis and inline software *guards* for runtime checks [64], [65]. Recent advances in SFI have achieved remarkably low overheads (under 10%) while maintaining strong isolation guarantees [51], [69]. Notable examples include XFI [17], BGI [11] for Windows, and LXFI [46] for Linux. Recent advancements focus on enhancing SFI performance using hardware features like ARM MTE [60] and PAC [48].

To enforce P1, SFI instruments driver memory instructions to restrict access within the driver's domain or explicitly granted memory ranges, configured by a trusted monitor or capabilities [46]. P4 is enforced by instrumenting *jump* instructions, restricting targets to the driver's domain or preconfigured addresses. Violations trigger a fault and module restart. XFI and LXFI enforce additional properties, assuming potentially malicious drivers. XFI incorporates a verifier for P5 but don't check interface data, thus fail to enforce P3 [17]. LXFI addresses this with *API integrity* via developer-supplied annotations for P3 and ensures these checks cannot be bypassed by enforcing P5 [46]. LXFI achieves this by using a combination of capability and shadow stack. BGI assumes a buggy driver model, implementing a superset of XFI's runtime checks while assuming unaltered control flow, balancing security and performance for bug-focused scenarios. Since BGI allows assigning capability at byte-granularity level, it can theoretically achieves memory safety if all the memory access rights are configured correctly by the developer. In general, SFI-based architectures ignore instrumentation on memory reads for performance, thus not enforcing P2.

3.2. Language-based driver isolation

Language-based driver isolation enforces memory safety (P6) within drivers using type systems or safe programming languages [24], [53], [70]. SafeDrive [70] uses a type system to prevent pointer bounds errors through compile-time and runtime checks, relying on annotations that can be partially inferred and supplemented by developers to ensure the safety of memory access types, including pointer bounds and union selectors. With complete annotations, SafeDrive detects and prevents all memory errors within the driver, enforcing P6. Consequently, it achieves key security properties of other driver isolation frameworks: P1 and P2 are enforced because the driver cannot directly corrupt or read kernel private data through memory bugs, and P4 is easily enforced since the driver cannot corrupt function pointers or return addresses, ensuring communication with the kernel only through predefined interface functions. However, SafeDrive does not enforce the remaining security properties. Similarly, rewriting drivers in safe languages like Rust [24], [53] achieves comparable isolation guarantees.

3.3. User-mode drivers

User-mode drivers run driver code in userspace processes, leveraging hardware privilege separation (Ring 0/3) to prevent direct driver access to kernel memory, inherently enforcing P1 [6], [21], [56]. Communication between userspace drivers and the kernel is performed through message-based channels, such as system calls in Microdrivers [21] and Decaf [56], or RPC in SUD [6]. This restricted interface approach inherently enforces P4, limiting drivers to invoke kernel code through predefined interfaces.

In user-mode drivers, data is passed via messages, maintaining separate object copies in the driver and kernel,

synchronized during cross-domain invocations. However, deep copying large and complex kernel objects at each domain crossing incurs high runtime overhead. Therefore, object synchronization across boundaries is typically selective, focusing on fields critical for correct execution. Microdrivers [21] and KSplit [33] employ static analysis to automate the identification of necessary fields.

Although primarily motivated by performance, object copying also facilitates enforcing P1 and P2 by not synchronizing kernel private data. P3 is not enforced for most of user-mode driver isolation projects, but an extension to the Microdriver architecture [8] infers and checks data invariants during domain crossings using the dynamic invariants inference tool Daikon [19]. User-mode driver architectures generally do not explicitly address P5 or P6.

3.4. Page table switching

Multiple projects utilize MMU to establish distinct page tables for different compartments. Nooks [63] employs page tables to enforce isolation, allowing kernel code unrestricted access to the driver's memory while restricting the driver's write access to kernel private data (P1). However, to synchronize kernel updates, Nooks allows the driver to read the kernel's page table and copy the data to its compartment, therefore violating P2 (kernel data confidentiality). Nooks provides wrapper functions for cross-domain invocations but do not enforce control flow integrity. Invocations from the driver to kernel are not guaranteed to jump to these wrapper functions, and thus both P4 and P5 are not enforced, allowing attacks like Page-Oriented Programming [27] to hijack control flow. Nooks does not enforce P3 as it doesn't limit or check data passed through the interface.

SIDE [62] uses MMU to run drivers within the kernel address space, allocating a specific region for each driver with user-level privilege. The isolated driver has its private stack and heap within this region. SIDE enforces P1 and P2, as unprivileged driver access to the kernel triggers a ring exception, invoking a handler to validate the access. Control transfers across isolation domains generate ring exceptions, allowing the handler to verify the transfer's legitimacy (P4). However, SIDE does not address P3, P5, or P6.

3.5. Virtualization

Virtualization provides a natural way to run untrusted code, and various driver isolation frameworks based on virtualization have been developed [39], [49], [52], [54], [68]. In an early demonstration, LeVasseur et al. [39] used Xen virtualization to run buggy device drivers along with a native OS in an unprivileged domU VM, with a translation module added to the driver's OS to serve as a server for external requests. This design achieves P1 and P2 due to VM isolation, and P4 by using a specified set of functions for communication between the driver and the external system.

Twindrivers [49], also built on Xen, employs an isolation boundary similar to [21], running performance-critical functions within the hypervisor domain and the remaining functions in the dom0 domain. It uses *software virtual memory (SVM)* to restrict hypervisor instance accesses to a single object in the dom0 instance's address space, enforcing P1, P2, and P4 through predefined upcalls and hypercalls.

Other hypervisors, such as KVM in LXDs [52] and Bareflank in LVDs [54], have also been used for driver isolation. LVDs optimize cross-VM communication using Intel VMFUNC instructions, avoiding the overhead of trapping to the hypervisor on every interaction. In these approaches, the memory accessible by the driver is limited to the VM virtual address space, and control transfers out of the driver must go through the hypervisor, enforcing P1, P2, and P4. However, it is worth noting that some of these projects perform selective copying, such as LVDs and KSplit. This approach may reduce the fields that required integrity checking (P3). However, P3 is generally not considered enforced.

3.6. Memory tagging

Memory tagging augments code and data with *tags*, which function as security metadata on memory. These tags are used to enforce memory access policies [34]. At a high level, memory tagging establishes isolation domains similar to SFI-based techniques. However, memory access operations are checked via hardware with much lower overhead (with an average of $< 5\%$ compared to native systems). The most commonly used memory tagging mechanisms are ARM MTE [3] and Intel MPK [15].

Sfitag [60] utilizes ARM MTE for driver isolation, enforcing the same security properties as SFI techniques (P1 and P4). The lower overhead of hardware-based checking enables Sfitag to instrument memory read instructions, thus enforcing P2, which SFI-based approaches typically omit for performance. HAKC [48] employs both MTE and Pointer Authentication Codes (PAC). MTE assigns 4-bit "colors" to mark memory ownership, while PAC cryptographically signs pointers to prevent tampering. The system organizes code into cliques (partitions with assigned colors) and compartments (collections with unique identifiers). Before dereferencing pointers, HAKC verifies signatures computed from color tags and the compartment context; access is defined for mismatched signatures, enforcing P1 and P2. Cross-compartment transfers recolor memory and resign pointers, with entry tokens defining valid transitions to enforce P4. However, P3, P5, and P6 are not enforced. Multiple frameworks utilize Intel MPK for compartmentalization. Bulkhead [26] uses protection keys for supervisor mode (PKS) with registered switch gates whose metadata is stored in write-protected tables. Each gate validates the source, installs target key rights, switches to a private stack, then jumps to the vetted entry, providing P1, P2, and P4 but not P3, P5, or P6. IskiOS [25] repurposes Intel PKU within the kernel (PKK) to support execute-only code and protected shadow stacks. The design primarily hardens intra-driver control flow through shadow stacks (P5) and can protect selected regions from writes (partial P1 for those regions), but does not define compartment boundaries or validate interface data (P2, P3, P4, P6 not enforced).

Summary In summary, nearly all driver isolation projects enforce kernel data integrity (P1) and interface control flow integrity (P4), which are fundamental for preventing driver faults from corrupting kernel data or hijacking kernel control flow. Frameworks addressing exploitable drivers typically also consider kernel data confidentiality (P2). However, interface data integrity (P3) is seldom fully enforced due to the complexity of inferring data invariants for validation at domain crossings, with only a few projects [8], [45] offering partial solutions. Driver control flow integrity (P5) is mainly considered by SFI techniques to ensure their instrumentation cannot be bypassed. Finally, driver memory safety (P6) can be achieved through language-based approaches, such as rewriting drivers in safe languages.

4. Compartment Interface Vulnerabilities

Driver isolation confines potentially buggy drivers' memory accesses. However, a compromised or malicious driver can still attack the kernel by misusing the kernel/driver interface. Such attacks are termed *Compartment Interface Vulnerabilities* (CIVs) [14], [38] and are specific to compartmentalized applications assuming a malicious compartment.

Researchers have long recognized potential attacks on the kernel/driver interface in compartmentalized systems. For instance, early microkernels like MINIX 3 showed that buggy isolated modules could violate IPC protocols, causing issues like deadlocks [28]. LXFI [45] used capabilities for interface function argument integrity, and Butt et al. [8] aimed to infer and enforce data invariants against corrupted driver data. Despite this, most current driver isolation frameworks do not address CIVs. As shown in Section 2.5, security properties P3 and P5, which cover interface and shared data misuse, are not widely enforced. This section categorizes CIVs into: (1) *shared data*, (2) *concurrency*, and (3) *control transfer*, based on an examination of prior work, notably RLBox [50] and Conffuzz [38].

4.1. CIV Taxonomy

The taxonomy we consider is presented in Table 3. It is based on previous work [14], [38], [50], with an extension of five new CIVs, highlighted in **bold** text. We also provide citations for the CIVs that have been studied by previous work. To avoid redundant discussion, we provide code examples only for the new CIVs. In addition, we note that while we have made our best effort to summarize and extend the existing taxonomy, we do not claim that this list covers all possible CIVs. As research in this area progresses, future work may discover and add newer CIVs to the taxonomy. Next, we discuss each category and the new CIVs in detail.

4.2. Shared data CIVs

As discussed in Section 2.4, shared data refers to the data that can be accessed by both the driver and the kernel after isolation. These include arguments, return values of interface functions, and global variables.

Category	CIV	SubCat.	Detailed Instances
SHARED DATA	Leakage		Leaking non-pointer values [14], [38] Leaking pointer values [38]
	Corruption	Memory Safety	Corrupted pointer value [14], [38] Corrupted pointer offset or buffer index [38] **Corrupted union type selectors** Parameters used in kernel memory API [14] Corrupted string [38]
		Decision Making	Corrupted guard (data attack) [38] Return wrong/invalid error code [57] **Corrupted loop condition**
		Arithmetic Error	Divided by zero [44] Integer overflow/underflow [44]
CONCURRENCY	Race Condition		Corrupted synchronization primitive [38] Callback state exchange [50] Shared memory TOCTTOU [38], [50]
CONTROL TRANSFER	Interface Bypass		Corrupted function pointer invoked by driver [38] Corrupted function pointer invoked by kernel [38]
	Interface Temporal Violation		**Sleep in an atomic context** **Lock and never unlock** **Unbalanced allocation/deallocation**

TABLE 3: CIV Taxonomy. The first column presents the high-level categories; the second presents the main CIV categories; the third further classifies the CIVs into more specific subcategories; the fourth column lists specific types of vulnerabilities that fall under a subcategory, along with references to relevant literature.

4.2.1. Shared data leakage

Shared data leakage CIVs expose kernel confidential information to isolated drivers. This leakage commonly manifests in two primary forms. Firstly, **Data Oversharing** occurs due to a lack of fine-grained access control over large, complex aggregate type objects. For instance, an entire structure might be shared when the receiving domain only requires a subset of its fields. Secondly, **Uninitialized Data** can lead to leakage when kernel-allocated shared objects are not fully initialized, which may expose sensitive data to the driver compartment during synchronization. This can happen either through incomplete initialization or compiler-added padding bytes for alignment [38], [43].

Leaked data can be of two types: (1) a pointer type, where leaked pointer values can expose address layout information, allowing the subversion of memory layout randomization techniques; and (2) a non-pointer type, which may contain sensitive kernel information, such as cryptographic keys, authentication tokens, or user data.

4.2.2. Shared data corruption

Shared data corrupted by malicious drivers can affect kernel operations when used in critical tasks such as memory API calls [14]. Conffuzz's CIV taxonomy classifies shared data corruption based on data type, e.g., corrupted pointers, indices, and objects like strings. However, this classification does not fully convey the end kernel uses of the corrupted data. We classify shared data corruption CIVs based on the kernel operations that use corrupted data, categorizing them as: (1) *memory safety* violations, (2) *decision making* violations, and (3) *arithmetic errors*.

Memory safety Memory safety CIVs occur when the kernel uses driver-corrupted data in memory operations. For example, a corrupted shared pointer or offset can cause corrupted kernel memory [14], enabling advanced attacks like DUI [30], which achieve the equivalent of controlling

```
1  union acpi_object {
2    acpi_object_type type;
3    struct {acpi_object_type type; u64 value;} integer;
4    struct {acpi_object_type type; ...; u8* pointer}
     ↪  buffer;
5    struct {..., union acpi_object *elements;} package;
6    ...
7  };

8  acpi_status acpi_extract_package(union acpi_object
   ↪  *package, ...) {
9    ...
10   for (i = 0; ... ; i++) {
11     union acpi_object
       ↪  *element=&(package->package.elements[i]);
12     switch (element->type) {
13       case ACPI_TYPE_BUFFER:
14       ...
15     }
16   }
17 }
```

Listing 1: Example of corrupted union type selector.

kernel read/write primitives.

Decision making Attackers can manipulate kernel control flow by corrupting data used in execution decisions, steering execution to paths containing sensitive operations.

Arithmetic error Arithmetic operations on corrupted data can lead to kernel faults. For example, a malicious driver supplying a zero divisor can trigger a divide-by-zero exception and subsequent kernel panic. Integer overflow or underflow can generate unexpected values, potentially bypassing security checks or causing API misuse when propagated.

New shared data corruption instances We identified two new shared data corruption CIVs: (1) corrupted type selectors in unions, and (2) corrupted loop condition.

Corrupted union type selector The kernel often uses tagged unions for polymorphism, determining data structure types via a tag or context. In a type confusion attack, the kernel is tricked into misinterpreting a data structure. For example, the `acpi_power_meter` driver's `acpi_object` union (Listing 1) can be exploited. If a malicious driver corrupts an extracted element's type field (line 12) after `acpi_extract_package` is called, it can cause a misinterpretation of the memory layout. For instance, an `integer` struct could be treated as a `buffer` struct. By crafting the `integer` struct's `value` field with a chosen address and then corrupting `type` field to `ACPI_TYPE_BUFFER`, attackers may trick kernel into arbitrary read/write operations via the `value` field, now misinterpreted as the `buffer`'s `pointer` field.

Corrupted loop condition Altering loop conditions can lead to various vulnerabilities. A simple case is a Denial-of-Service (DoS) attack, where forcing excessive loop iterations degrades kernel performance or causes unresponsiveness. A more severe vulnerability arises if an attacker manipulates a loop termination condition that also controls buffer indexing, potentially leading to a buffer overflow. For example, in the `mgag2000` driver, corrupting the `n_layers` field, which dictates loop iterations and access to the `mci->layers` array, could cause a buffer overflow, as detailed in Listing 2.

```
1  void edac_mc_handle_error(..., struct mem_ctl_info
   ↪  *mci, ...) {
2      for (i = 0; i < mci->n_layers; i++) {
3              if (pos[i] >= (int)mci->layers[i].size)
               ↪  {...}
4      }
5      ...
6  }
```

Listing 2: Example buffer overflow with corrupted loop conditions.

4.3. Race conditions

Race conditions involve concurrent accesses to shared data. We consider three types of concurrency CIVs.

TOCTTOU Time-of-Check-to-Time-of-Use vulnerabilities (TOCTTOU) can occur when an isolated driver modifies shared data between a kernel's check and use. This creates a window for the driver to manipulate the checked value post-validation. To address the TOCTTOU issue, the check and the use of the check must be atomic. For example, RLBox [50] provides a mechanism for making a copy when a compartment receives data and ensures that all checks and uses are performed on the copied data.

Shared Lock Corruption An isolated driver with the ability to modify shared lock values can compromise atomic regions dependent on these locks. This represents a specialized form of shared data corruption [38].

Callback State Exchange RLBox [50] identifies a multi-threading attack where a malicious compartment provides corrupted object instances to multiple threads in the trusted compartment. For example, a compromised driver could supply identical device objects to different threads handling distinct devices, leading to unexpected race conditions in kernel API invocations.

4.4. Control transfer CIVs

Control flow transfer CIVs fall into two categories: (1) *interface bypass* and (2) *interface temporal violations*.

4.4.1. Interface bypass

Driver isolation frameworks restrict the driver and kernel to interact through a set of predefined interfaces. In the kernel-driver context, interface functions are accessed via function pointers. If an isolated driver can corrupt these function pointers, it may either use a corrupted function pointer to invoke arbitrary kernel functions or trick the kernel into jumping to arbitrary code locations when invoking driver callbacks through these pointers. However, if P4 (interface control flow integrity) is enforced, these attacks would be prevented even with corrupted function pointers. We discuss this assumption further in Section 5.6 when addressing control transfer CIV quantification.

4.4.2. Interface temporal violation

An interface temporal violation is normally caused by calling interface functions in the wrong order. That is, although a driver may invoke only allowed kernel interface functions, doing so in the wrong order is possible without

further defenses. While there are many kinds of violations, we discuss three such CIV types in this study: (1) Sleep in atomic context, (2) Lock and never unlock, and (3) Unbalanced allocation/deallocation.

Sleep in atomic contexts (SAC) This is a CIV type specific to the kernel context [5]. Atomic contexts, such as spinlock-protected regions or interrupt handlers, require the driver to complete its operations promptly and without blocking. However, if a driver invokes a sleepable kernel interface while holding a spinlock, such as calling kernel memory allocation functions (e.g., `kmalloc()`) without passing the `GFP_ATOMIC` flag, the system can deadlock or crash. Such vulnerabilities can be detected using static analysis [5].

Lock and never unlock A malicious driver can hold shared locks required by other kernel threads for making progress, causing system-wide hangs or denial-of-service attacks.

Unbalanced allocation/deallocation Drivers are responsible for correctly manage object lifetime via kernel memory management APIs, e.g., kmalloc/kfree. A malicious driver can intentionally create memory leaks by bypassing calls to "`free`" on allocated objects. This can potentially lead to DoS attacks by depleting system memory. We show an instance of this CIV in Appendix C (Listing 3).

5. Evaluating Driver Isolation Effectiveness

Our discussion thus far has covered existing driver isolation techniques and the classes of CIVs that can compromise driver isolation security. However, several key questions remain unanswered: (1). Which CIVs remain possible under driver isolation?; (2). How prevalent are these CIVs? and (3). How certain security properties can be leveraged to mitigate CIVs? To answer these questions, we perform our evaluation under different threat models. At a high level, we first begin with a baseline isolation model common to most driver isolation frameworks to answer Q1 and Q2. Then, we explore how extra properties such as P3, P5 and P6, can be applied to reduce the number of CIVs.

5.1. Threat models

We define a baseline threat model for an isolated driver framework that enforces the following security properties: (1) P1: kernel data integrity, (2) P2: kernel data confidentiality, and (3) P4: interface control flow integrity. We also assume selective sharing of data, where only necessary fields in objects are synchronized. This partially enforces P3 by avoiding synchronizing unnecessary data; however, we do not assume a defense that infers and validates invariants of interface data, which is an open problem. This driver isolation model captures the majority of driver isolation frameworks that aim to confine a buggy driver by controlling data sharing. We consider that an exploitable driver contains bugs that allow an attacker to execute arbitrary code within the driver. Consequently, all driver private data and shared data can be read and modified by the driver, the control flow within the driver can be hijacked, and arbitrary code can be executed, enabling the attacker to jump to any driver location and invoke any kernel interface function.

To investigate how CIVs may be mitigated by additional security properties, we compare numbers of CIVs in the baseline model with the numbers when a security property is additionally enforced. This approach allows us to understand the effect of enforcing a security property on CIVs.

5.2. Methodology of quantifying CIVs

In this section, we present our methodology for quantifying CIVs at the kernel-driver isolation boundary. We summarize the CIV classes, and the corresponding metrics in Table 4. Our quantification methodology relies on static analysis, which identifies CIVs that can potentially be exploited given our threat model. Since static analysis may produce false positives, we present how we validate our results using manual analysis and proof of vulnerabilities in Section 6.1. We next explain the detailed method for quantifying each CIV class.

Category	CIV Class	Metrics (Counts)
Shared data	Data leakage	Total fields - Shared fields
	Data corruption	Taint paths
Concurrency	Race	Corruptible shared locks
Ctrl transfer	Sleep in atomic context	(spinlock, sleepable func) pairs
	Lock never unlock	Lock/unlock pairs on shared locks
	Unbalanced allocation	Allocation/deallocation pairs

TABLE 4: CIV classes and their metrics.

5.3. Quantifying shared data leakage CIVs

As described in Section 4.2.1, kernel sensitive information can be leaked due to overshared or uninitialized data. For our evaluation, we focus on measuring the degree of overshared data and omit the quantification of uninitialized data for two reasons: (1) it is hard to determine uninitialized data due to compiler padding and (2) in certain object-copying based techniques, access is granted to only the private object copy in the driver and thus the driver cannot access uninitialized data in the kernel's copy to learn confidential data.

To measure the degree of oversharing, we utilize the *shared field analysis* in KSplit [33] to identify those struct fields whose states are required for correct execution. This analysis is built on top of the algorithm used by Microdrivers [21], and further improves precision by determining shared fields. Our algorithm takes all structure-typed interface function parameters, return values, and shared global variables as input, and computes the following metrics: (1) the total number of structure fields if deep copy is used, (2) the number of accessed structure fields, and (3) the number of shared and accessed structure fields (via shared field analysis). The difference between the field numbers computed by (1) and (3) captures the degree of oversharing.

5.4. Quantifying shared data corruption CIVs

Our quantification involves two steps: taint analysis to identify paths from interface data (sources) to kernel operations (sinks), followed by pruning to focus on controllable paths. This extends CIVScope [14] to a more complete CIV classification while addressing the taint path explosion.

Taint analysis Algorithm 1 (Appendix A) presents the taint analysis, and Table 7 (Appendix F) lists the included source/sink types. The analysis takes all the arguments and return values of interface functions as well as global variables as taint sources, and a set of specified kernel operations as taint sinks. It outputs a set of taint paths from the sources to the sinks. Our taint analysis propagates taints on a field-sensitive, path-, and context-insensitive, interprocedural *program dependence graph* (PDG) in LLVM [42]. The PDG implementation captures the data and control dependencies among LLVM IR instructions and variables. It utilizes the SVF alias analysis [61] to capture intraprocedural pointer aliasing. Interprocedural aliasing, on the other hand, is captured by interprocedural dependence edges in the PDG. Although the PDG lacks path and context sensitivity, it allows us to identify all potential risky operations that could operate on tainted data.

Finding likely attackable paths with pruning A taint trace produced by our analysis indicates a potentially risky operation that can be affected by values controlled by the attacker. However, the risky operation may be protected by various checks; consequently, whether the risky operation is exploitable depends on whether the attacker can bypass those checks. Thus, to identify paths that are likely to be exploited, we employ the following two heuristics: (1). there is no check in the taint trace, or (2). there are checks in the taint trace, but none of them directly checks the data used in the sink. The first heuristic identifies those taint traces that are easily exploitable by the attacker, because the attacker can reach the sink without going through any check. The second heuristic identifies those traces whose sinks are guarded by some checks but those checks may not be effective as they do not directly check data used by the sinks. If a taint trace meets one of these heuristics, we preserve the trace. However, these heuristics may produce both false positives and false negatives. We provide evaluation of these heuristics in the evaluation section (Section 6).

5.5. Quantifying concurrency CIVs

We focus on quantifying corrupted shared locks, a specific type of concurrency-related CIV. To identify shared data susceptible to this type of CIV, we extend KSplit's shared data analysis [33]. This is a field-sensitive analysis that computes shared data based on the accesses in both domains. And then, we identify data that are used within lock APIs, to obtain the shared lock instances.

5.6. Quantifying Control Transfer CIVs

We focus on quantifying a specific category of control transfer CIVs: **interface temporal violations**. The other category, interface bypass via function pointer corruption, is considered infeasible in our threat model, which assumes the enforcement of P4 (interface control flow integrity) and trusts that drivers only use predefined kernel interfaces.

Statically detecting all interface temporal violations is challenging. Driver protocols can be complex, often involving device-specific signals beyond the kernel-driver interface, and formal state-machine descriptions [2], [57], [58] are not commonly used in practice. Since our static analysis is limited to the kernel-driver interface, fully capturing all protocol violations is difficult. Therefore, **we limit our scope** to quantifying three specific types of interface temporal violations: Sleep in Atomic Context (SAC), holding shared locks indefinitely (Lock and Never Unlock), and Unbalanced Allocation/Deallocation.

SAC Violations A driver can trigger an SAC violation by calling a sleepable kernel function while holding a spinlock, violating kernel synchronization rules. We quantify potential SAC instances by identifying drivers that *can* invoke both a spinlock-acquiring interface function and a sleepable interface function. We identify sleepable functions using DSAC [5]. Each unique pair of such callable functions within a driver is counted as one potential SAC CIV instance.

Lock and Never Unlock A malicious driver could indefinitely hold a **shared kernel lock** by acquiring it through an interface function but failing to call the corresponding unlock function. Holding *private* driver locks is considered only a driver-level Denial-of-Service and is excluded. To quantify this, we first identify shared locks (as described in Section 5.5). We then identify pairs of lock/unlock API functions within the driver's interface that operate on these shared locks. Each pair represents a potential CIV instance.

Unbalanced Allocation/Deallocation Drivers might cause memory leaks or use-after-free errors by invoking sequences of interface functions leading to unbalanced calls to kernel allocation and deallocation APIs for the *same* object. Precisely tracking individual objects across interface calls typically requires complex alias analysis [40], which is challenging to scale to the entire kernel. To balance accuracy and scalability, we employ a **type-based approach** for quantification. This approach involves first identifying all interface functions that can invoke kernel memory allocation or deallocation APIs. Then, for potential pairs of allocation and deallocation API calls reachable via the interface, we check if they operate on objects of the *same data type*. This check includes resolving types for `void*` pointers by tracking type casting operations. Each identified alloc/dealloc pair operating on the same type constitutes a potential CIV instance.

6. Quantifying CIV Attack Surface

In this section, we present our classification results on CIVs. Our evaluation goals are as follows:

- Assuming the baseline threat model, how prevalent is each type of CIV at the driver/kernel isolation boundary

across different driver classes?
- How do the baseline CIV statistics change based on the enforcement of extra security properties?

For evaluation, we select 11 drivers (Table 5) from 7 different driver classes. The drivers are selected by referencing previous studies on driver isolation, e.g., Ksplit [33]. The chosen drivers represent major OS subsystems (networking, storage, USB, graphics), include rich interface boundaries (bidirectional function pointer exchanges, tagged unions, linked lists), and span multiple levels of complexity. This diversity ensures our evaluation captures a representative spectrum of driver–kernel interaction patterns.

6.1. CIV Statistics for the Baseline Threat Model

Shared data leakage Table 5a presents the quantification result of data oversharing. The first row shows the number of fields directly accessible to the driver, assuming that all fields are deep-copied to the driver. The second row shows the number of fields needed for correct execution, computed based on the parameter access analysis by Microdrivers [21] (and Decaf [56]). The third row shows the number of kernel/driver shared fields, computed using the shared field analysis by KSplit [33].

Insight 1: Despite the intended isolation between kernel and drivers, many private kernel fields remain accessible due to overly permissive data sharing. Deep-copying entire objects often results in over 99% of shared fields being unnecessary, and even more selective methods still expose about twice the needed data. Limiting sharing to only essential fields can greatly reduce this leakage and improve kernel data confidentiality.

Shared fields are typically under 1% of deep-copied fields and about 50% fewer than accessed fields, indicating significant oversharing when unrestricted sharing is assumed. Even fields accessed (Microdrivers) are roughly double the shared fields (KSplit). These findings support restricting shared data to mitigate confidentiality-related CIVs.

Shared data corruption Table 5b presents the statistics of shared data integrity CIVs. Statistics for each subcategory are collected using the taint analysis described in Section 5.4. Due to the pruning strategy outlined in Section 5.4, we organize the statistics in two columns for each driver: *P* (Pruned) presents the number of paths after applying the pruning strategy, while *M* (Manual) presents the number of CIVs remaining after manual inspection.

1) **Memory Safety:** This subcategory shows the highest diversity of CIVs. MEM1 (pointer value corruption) is particularly prevalent in `ixgbe` (48 instances) and `nvme` (15 instances). MEM2 (pointer offset/buffer index) also shows significant occurrences in `ixgbe` (32 instances). MEM3 (type selector) is rare, with only 3 instances in the `power_meter` driver. MEM4 (sensitive kernel APIs) is the most prevalent memory safety CIV, although less common than in CIVScope [14] due to the consideration of selective data copying in our experiment. MEM5 (corrupted string) is less common, with only 26 instances in total.
2) **Decision-making data:** DM1 (corrupted branch guard) instances are found in every driver, indicating that isolated drivers can still affect kernel control paths. DM2 (invalid/wrong error code) is prevalent, with `ixgbe` (1,234 instances) and `sfc` (1,135 instances) standing out, and notable occurrences in `nvme` (53 instances) and `mgag200` (28 instances). Some drivers, like `ixgbe`, have too many DM2 instances for manual verification. DM3 (Corrupted loop condition) is relatively less common compared to other DM classes (85 instances in total), indicating that many loop conditions within the kernel can potentially be corrupted and leads to buffer overflow when the buffer inside the loop body is accessed.
3) **Arithmetic Errors:** AE1 (division by zero) is absent across all investigated drivers, indicating that the kernel rarely uses driver-supplied data as a denominator. AE2 (integer overflow/underflow) is more prevalent, with the highest number of vulnerabilities in `ixgbe` (28 instances) and `nvme` (20 instances), suggesting that arithmetic operations using driver-corrupted data can have potential overflow/underflow issues.

Our static analyses identifies CIVs, which are potential vulnerabilities exist at the interface. However, since static analysis approximates, it cannot confirm that the identified CIVs can cause actual harm to the kernel. To gain confidence on static analysis results, we performed two types of validation: manual validation and construction of proof-of-concept (PoC) exploits for a random sample of cases; these efforts will be detailed later.

Concurrency As presented in Table 5c, the `ixgbe` driver has the highest instances of corruptible shared lock (3 instances). This is followed by the `null_net` and `sfc` drivers, each with 1 occurrence. But overall, corruptible shared locks are pretty rare across the studied drivers. This matches KSplit's result [33] that shared locks are rare across device drivers.

Control transfer Table 5d presents the frequency of interface temporal violation CIVs. SAC vulnerabilities (first row) are relatively infrequent, with `ixgbe`, `sfc`, `nvme`, `mgag200`, and `usb_f_fs` drivers exhibiting 3 instances each, and `null_blk` with 2 instances. Lock and never unlock issues (second row) are slightly more prevalent, with `null_blk` and `usb_f_fs` having 4 instances each, and `ixgbe` having 3 instances. Unbalanced allocation/deallocation issues are prevalent across several drivers, with `sfc` leading (64 instances), followed by `ixgbe` (48 instances). The prevalence of control transfer temporal violations in almost all drivers underscores the importance of specialized detection and dedicated prevention, especially given that we tested only a small subset of such temporal violation issues.

Insight 2: We observed far fewer CIVs than prior work [14], thanks to pruning strategies and focusing on shared data. This suggests that securing interfaces is feasible without major redesign.

	ixgbe	null_net	sfc	msr	null_blk	nvme	sb_edac	mgag200	usb_f_fs	power_meter	dm-zero
Deep copy	999K	48K	846K	24K	227K	321K	15K	467K	92K	20K	11K
Microdrivers field access [21]	4K	231	6K	66	562	643	91	597	641	144	29
KSplit shared field [33]	1983	73	1K	13	247	249	31	333	173	29	46

(a) Quantify overshared struct fields (data leakage CIVs).

CIV Classes	ixgbe		null_net		sfc		msr		null_blk		nvme		sb_edac		mgag200		usb_f_fs		power_meter		dm_zero	
MEMORY SAFETY (MEM1)	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M
MEM1: Pointer value	48	48	2	1	0	0	0	0	1	1	15	15	2	1	5	3	12	3	0	0	1	1
MEM2: Pointer offset/buffer index	32	24	2	0	0	0	8	1	0	0	10	10	0	0	9	4	9	0	2	2	0	0
MEM3: Type selector	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3	0	0
MEM4: Sensitive kernel memory APIs	77	70	5	3	3	3	13	9	2	2	10	10	6	6	26	18	11	9	2	2	2	1
MEM5: Corrupted string	5	5	2	2	2	2	2	2	4	3	3	2	2	2	3	3	1	1	2	2	0	0
DECISION-MAKING VARIABLE	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M
DM1: Corrupted guard	104	100	45	45	73	73	81	81	20	20	15	13	66	66	72	70	51	51	37	37	2	2
DM2: Invalid/wrong error code	1234	N/A	20	13	1135	N/A	9	9	13	10	53	N/A	1	1	28	28	24	19	4	4	0	0
DM3: Corrupted loop condition	31	25	1	1	2	2	0	0	0	0	10	10	12	9	13	1	7	7	7	7	2	2
ARITHMETIC ERROR	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M	P	M
AE1: Divided by zero	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AE2: Integer overflow/underflow	28	28	2	0	0	0	9	2	0	0	20	11	0	0	4	2	9	4	0	0	0	0

(b) Quantify shared data corruption CIVs. N/A stands for not verified due to large manual effort.

	ixgbe	null_net	sfc	msr	null_blk	nvme	sb_edac	mgag200	usb_f_fs	power_meter	dm-zero
No. shared lock	3	1	1	0	0	0	0	0	0	0	0

(c) Quantify corruptible shared lock (concurrency CIV)

	ixgbe	null_net	sfc	msr	null_blk	nvme	sb_edac	mgag200	usb_f_fs	power_meter	dm-zero
Sleep in an atomic context	3	0	3	0	2	3	0	3	3	0	0
Lock and never unlock	3	1	2	0	4	1	0	1	4	1	0
Unbalanced alloc/dealloc	48	6	64	6	2	47	5	42	14	7	0

(d) Quantify control transfer CIVs.

TABLE 5: CIV quantification results. Table 5a shows the degree of overshared fields that are not read by the driver; Table 5b shows the number of shared data integrity CIVs; Table 5c shows the number of corruptible shared lock instances; Table 5d shows the number of control transfer CIVs.

Manual validation First we conducted a two-phase manual validation on the identified cases. In the first phase, a student evaluated each case to determine if it represented a true positive under our threat model. For each case, the student provided a concise justification, detailing how corrupted shared data could lead to potential attacks. The second phase involved a peer review to confirm the validity of the cases. Our manual validation reveals that the taint analysis achieves a high level of accuracy, with a precision of 90% across the evaluated cases. This demonstrates the effectiveness of our approach in identifying potential security vulnerabilities. However, the analysis revealed a small number of false positives, primarily due to limitations in our path pruning strategy. Specifically, the second heuristic relies on data flow analysis to determine whether the data used in taint sinks are subject to proper checks. The imprecision inherent in this heuristic accounts for the observed false positives.

Construction of PoC exploits To validate exploitability of identified CIVs, we randomly sampled diverse instances across CIV classes and drivers. For each, we created two programs: *K'* (kernel code snippet) and *D'* (attack code), then executed them to verify exploitation potential. For shared data-related CIVs, we instrumented sink operations to check if malicious values could control operations (example in Listing 5, Appendix E). For control transfer CIVs, we used class-specific verification: success for SAC and Lock/never unlock means system hang; for unbalanced malloc/dealloc, success means memory consumption approaching system capacity (128GB). Table 6 (Appendix D) shows high PoC exploit success rates (60-100%) across CIV classes, indicating most instances identified by our static analysis are genuine vulnerabilities. Some attacks failed, primarily in shared data corruption cases, due to post-sink validation guards (a detailed example in Appendix C).

6.2. Impact of enforcing security properties

We examine how three additional security properties, namely interface data integrity (P3), control flow integrity (P5), and memory safety (P6), influence the prevalence of CIVs. Complete statistics are provided in Appendix G.

Interface data integrity (P3) Interface data integrity (P3) can be enforced using data invariants, developed manually or automatically [8], [19]. However, the challenge in invariant inference, coupled with uncertainties in specifiable invariants, make quantifying P3's impact on mitigating shared data corruption CIVs difficult. Consequently, we do not quantify CIV reduction for P3.

Control flow integrity (P5) Control flow integrity (P5) restricts driver execution to its control-flow graph (CFG), preventing arbitrary code execution and limiting attacks to data corruption within the CFG (e.g., control-flow bending [9]). We assess P5's impact on shared data corruption by memory-unsafe driver operations, leveraging recent analyses of memory-safe objects [31], [32]. These methods isolate memory-safe objects, rendering them immune to memory errors and reducing shared data CIVs on such objects (Table 8). While P5 significantly mitigates control-transfer CIVs by enforcing CFG adherence, it does not address interface temporal violations if such paths exist in the CFG. For instance, existing SAC CIVs within driver code persist despite P5 enforcement (Table 9).

Memory safety (P6) Memory safety (P6) enforcement in drivers prevents memory error exploitation for driver memory corruption, thereby eliminating shared data corruption

CIVs stemming from such errors. This, however, does not address shared data corruption due to inherent correctness issues in buggy drivers. P6 also mitigates control-transfer CIVs, offering stronger protection than P5; thus, any CIVs P5 mitigates are also addressed by P6.

Results summary Enforcing P5 leads to a 5% to 20% overall reduction (average 15%) in shared data corruption CIVs across all classes and drivers (details in Appendix G and Table 8). The Decision Making (DM) class, especially DM2 (Invalid/wrong error code), exhibited the most significant improvement, with up to 186 and 170 fewer instances in the ixgbe and sfc drivers, respectively. The Memory Safety (MEM) class, particularly MEM4 (Sensitive kernel memory APIs), also saw substantial reductions (e.g., up to 12 fewer instances in ixgbe). Notably, the ixgbe and sfc drivers demonstrated the greatest CIV reduction across multiple vulnerability classes.

Insight 3: Isolation works best after the driver is hardened. Enforcing P5 or P6, for example through CFI or a safe language such as Rust, greatly reduces the attack surface exposed at the isolation boundary.

7. Guidelines for Secure Driver Compartmentalization

Our evaluation of 11 representative drivers reveals several critical patterns and actionable insights for designing secure driver isolation systems. We distill our quantitative findings into concrete guidelines for practitioners and researchers.

Guideline 1: Minimize interface data exposure Our analysis reveals that sharing entire objects across isolation boundaries exposes over 99% unnecessary fields, while even access-based sharing approaches [21] expose approximately twice the data required for correct execution. This oversharing directly undermines kernel data confidentiality/integrity, as drivers gain access to sensitive fields they never use, including potential exposure of cryptographic keys, authentication tokens, or uninitialized memory. Isolation frameworks should employ fine-grained shared field analysis, such as KSplit [33], to identify and synchronize only the minimal set of fields required for correct execution. Such analysis should be a baseline security requirement rather than merely a performance optimization.

Guideline 2: Validate memory operations and data interpretation at isolation boundaries Parameters passed to sensitive kernel APIs represent the most prevalent vulnerability class, appearing in every driver we studied. These vulnerabilities arise when isolated drivers supply corrupted pointers, buffer indices, or size parameters to memory management functions (kmalloc, memcpy, DMA operations). Isolation frameworks should wrap all sensitive kernel APIs with validation layers that verify pointer bounds, size parameters. Beyond direct memory operations, type confusion in tagged unions poses a particularly severe risk by causing the kernel to misinterpret memory layouts. For tagged unions crossing isolation boundaries, frameworks must validate that type tags fall within valid ranges, match the expected calling context, and cannot be modified between validation and use.

Guideline 3: Validate data used in control flow decisions Corrupted guard variables appear in every driver we evaluated, indicating kernel control flow frequently depends on driver-supplied data. Malicious drivers can exploit this to steer execution along unintended paths, bypassing security checks, triggering privileged operations, or accessing unauthorized resources. Frameworks should identify security-critical branch points where execution paths have different security implications and validate guard conditions using trusted kernel state. CFI (P5) provides a crucial second layer of defense, ensuring that even with corrupted guards, the kernel cannot be redirected to arbitrary code locations.

Guideline 4: Combine isolation with driver hardening Driver isolation alone leaves substantial attack surface at the interface boundary. Enforcing driver CFI reduces interface vulnerabilities by approximately 15%, with significant improvements in decision-making and memory safety categories. However, CFI does not address temporal violations CIVs, which remain exploitable within legitimate control flow paths. The most effective approach combines multiple defensive layers: CFI to restrict driver execution to valid control flow paths, comprehensive interface validation (Guidelines 2 and 3), and temporal integrity enforcement through resource tracking and protocol state machines. For new driver development, memory-safe languages such as Rust eliminate entire classes of memory corruption vulnerabilities at their source, providing stronger security guarantees than retrofitting runtime checks to C code.

8. Conclusion

We have systemized existing driver isolation frameworks to understand their enforced security properties. We also investigated existing taxonomy of CIVs, and estimate their impact under a baseline threat model. In addition, we explore how enforcing security properties can mitigate CIVs. We believe that future development on driver isolation frameworks can benefit from our findings.

Acknowledgments

This material is based on research sponsored by NSF CNS-1801534, NSF 2239615, DARPA HR0011-19-C-0106, AFRL FA8750-25-C-B041 and a grant from MIT Lincoln Labs. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

References

[1] LKDDb: Linux Kernel Driver DataBase. https://cateee.net/lkddb/. Accessed on 04.23.2019.

[2] Sidney Amani, Leonid Ryzhyk, Alastair F Donaldson, Gernot Heiser, Alexander Legg, and Yanjin Zhu. Static analysis of device drivers: We can do better!

[3] Arm. Armv8.5-A Memory Tagging Extension. *Whitepaper*. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[4] Ahmed Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[5] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the linux kernel. *ACM Trans. Comput. Syst.*, 36(4), apr 2020.

[6] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux.

[7] Anton Burtsev, Vikram Narayanan, Yongzhe Huang, Kaiming Huang, Gang Tan, and Trent Jaeger. Evolving operating system kernels towards secure kernel-driver interfaces. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, page 166–173, Providence RI USA, June 2023. ACM.

[8] Shakeel Butt, Vinod Ganapathy, Michael M. Swift, and Chih-Cheng Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *2009 Annual Computer Security Applications Conference*, page 301–310, Honolulu, Hawaii, USA, December 2009. IEEE.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 161–176, 2015.

[10] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 45–58, 2009.

[11] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2009.

[12] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264. Association for Computing Machinery, 2013.

[13] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[14] Yi Chien, Vlad-Andrei Bădoiu, Yudi Yang, Yuqian Huo, Kelly Kaoudis, Hugo Lefeuvre, Pierre Olivier, and Nathan Dautenhahn. Civscope: Analyzing potential memory corruption bugs in compartment interfaces. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, KISV '23, page 33–40, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Intel Corporation. Intel® 64 and ia-32 architectures software developer manuals. Landing page for the Intel® 64 and IA-32 Architectures Software Developer's Manuals.

[16] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, page 292–307, San Jose, CA, May 2014. IEEE.

[17] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[18] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 75–88, 2006.

[19] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[20] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 38–51, 1997.

[21] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178, 2008.

[22] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[23] Gilbert Neiger, Barry E. Huntley, Ravi L. Sahita, Vedvyas Shanbhogue, Jason W. Brandt,. Instruction-Set Support for Invocation of VMM-Configured Services without VMM Intervention, U.S Patent 9804871B2, Oct. 2017.

[24] Amélie Gonzalez, Djob Mvondo, and Yérom-David Bromberg. Takeaways of implementing a native rust udp tunneling network driver in the linux kernel. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, page 18–25, Koblenz Germany, October 2023. ACM.

[25] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Iskios: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[26] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. Bulkhead: secure, scalable, and efficient kernel compartmentalization with pks. *arXiv preprint arXiv:2409.09606*, 2024.

[27] Seunghun Han, Seong-Joong Kim, Wook Shin, Byung Joon Kim, and Jae-Cheol Ryou. Page-Oriented programming: Subverting Control-Flow integrity of commodity operating system kernels with Non-Writable code pages. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 199–216, Philadelphia, PA, August 2024. USENIX Association.

[28] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Countering ipc threats in multiserver operating systems (a fundamental requirement for dependability). In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, page 112–121, December 2008.

[29] Herder, J.N. and Bos, H. and Gras, B. and Homburg, P. and Tanenbaum, A.S. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[30] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. *Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software*, volume 9327 of *Lecture Notes in Computer Science*, page 312–331. Springer International Publishing, Cham, 2015.

[31] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In *Proceedings 2022 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2022. Internet Society.

[32] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. Top of the heap: Efficient memory error protection of safe heap objects. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 1330–1344, New York, NY, USA, 2024. Association for Computing Machinery.

[33] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

[34] Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowyra, Roger Khazan, Howard Shrobe, and Hamed Okhravi. Tag: Tagged architecture guide. *ACM Computing Surveys*, 55(6):1–34, July 2023.

[35] Asim Kadav and Michael M Swift. Understanding modern device drivers. page 12.

[36] P.A. Karger and R.R. Schell. Multics security evaluation: vulnerability analysis. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, page 127–146, Las Vegas, NV, USA, 2002. IEEE Comput. Soc.

[37] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., and others. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[38] Hugo Lefeuvre, Vlad-Andrei Badoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. Assessing the impact of interface vulnerabilities in compartmentalized software.

[39] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI '04)*, pages 17–30, 2004.

[40] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel.

[41] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two Years of Experience with a μ-Kernel Based OS. *ACM SIGOPS Operating Systems Review*, 25(2):51–62, April 1991.

[42] Shen Liu, Gang Tan, and Trent Jaeger. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 2359–2371, 2017.

[43] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 920–932, Vienna Austria, October 2016. ACM.

[44] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. Printfuzz: fuzzing Linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 404–416. ACM, July 2022.

[45] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 115–128, 2011.

[46] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP*, pages 115–128, 2011.

[47] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[48] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[49] Aravind Menon, Simon Schubert, and Willy Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. *ACM SIGARCH Computer Architecture News*, 37(1):301–312, March 2009.

[50] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Sorin Lerner, Hovav Shacham, Deian Stefan, and Eric Rahm. Retrofitting fine grain isolation in the firefox renderer.

[51] Shravan Narayan, Tal Garfinkel, Evan Johnson, Zachary Yedidia, Yingchen Wang, Andrew Brown, Anjo Vahldiek-Oberwagner, Michael LeMay, Wenyong Huang, Xin Wang, et al. Segue & colorguard: Optimizing sfi performance and scalability on modern architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 987–1002, 2025.

[52] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs : Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[53] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, pages 21–39, 2020.

[54] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*, pages 157–171, 2020.

[55] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 116–132, 2013.

[56] Matthew J Renzelmann and Michael M Swift. Decaf: Moving Device Drivers to a Modern Language. In *2009 USENIX Annual Technical Conference (USENIX ATC '09)*, 2009.

[57] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 275–288, 2009.

[58] Leonid Ryzhyk, Ihor Kuz, and Gernot Heiser. Formalising device driver interfaces. In *Proceedings of the 4th workshop on Programming languages and operating systems*, page 1–5, Stevenson Washington, October 2007. ACM.

[59] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[60] Jiwon Seo, Junseung You, Yungi Cho, Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. Sfitag: Efficient software fault isolation with memory tagging for arm kernel extensions. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, page 469–480, Melbourne VIC Australia, July 2023. ACM.

[61] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266, 2016.

[62] Yifeng Sun and Tzi-cker Chiueh. Side: Isolated and efficient execution of unmodified device drivers. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[63] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. Association for Computing Machinery.

[64] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3):137–198, 2017.

[65] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 203–216, New York, 1993. ACM Press.

[66] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 241–254, 2008.

[67] Huamao Wu, Yuan Chen, Yajin Zhou, Yifei Wang, and Lubo Zhang. Driverjar: Lightweight device driver isolation for arm. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, page 1–6, San Francisco, CA, USA, July 2023. IEEE.

[68] Xi Xiong, Donghai Tian, and Peng Liu. Practical protection of kernel integrity for commodity os from untrusted extensions.

[69] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 649–665, 2024.

[70] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.

Appendix

1. Taint analysis algorithm

Input: Taint sources S; PDG G
Output: Taint traces from sources to sinks
Function `TaintAnalysis`:
  T = risky kernel operations (Table 7);
  $Tr = \{\}$;
  foreach $s \in S$ **do**
    Propagate taint from s along data-dependency edges of G;
    if *node* n *becomes tainted and* $n \in T$ **then**
      $Tr \leftarrow Tr \cup \{\text{trace from } s \text{ to } n\}$;
    end
  end
  return Tr

Algorithm 1: Taint Analysis

2. Unbalanced allocation and free example

```
1  int bond_create(struct net *net, const char *name)
2  {
3      struct net_device *bond_dev;
4      struct bonding *bond;
5      int res = -ENOMEM;
6      rtnl_lock();
7      bond_dev = alloc_netdev_mq(sizeof(struct
       ↪  bonding), ...);
8      if (!bond_dev)
9          goto out;
10     bond = netdev_priv(bond_dev);
11     dev_net_set(bond_dev, net);
12     bond_dev->rtnl_link_ops = &bond_link_ops;
13     res = register_netdevice(bond_dev);
14     if (res < 0) {
15         free_netdev(bond_dev);
16         goto out;
17     }
18     ...
19     out:
20     rtnl_unlock();
21     return res;
22 }
```

Listing 3: Vulnerable Linux ethernet bonding driver with unbalanced memory allocation.

Listing 3 shows part of the code of the Linux ethernet bonding driver. In the `bond_create` function, memory is allocated for the bonding network device using `alloc_netdev_mq` (Line 7). If the registration is not successful, the allocated object is freed and in the error-handling path. However, a malicious driver, with the ability to execute arbitrary code, can deliberately skip the error handling, causing the unregistered object to persist in memory. This

results in a memory leak, as the allocated memory for the network device remains occupied but unreachable. Over time, repeated allocation by the malicious driver can accumulate significant amounts of leaked memory, potentially exhausting system resources.

3. Failed shared data corruption example

```
static void *kmalloc_reserve(unsigned int *size,
↪  gfp_t flags, ...) {
    ...
    size_t obj_size = SKB_HEAD_ALIGN(*size);
    obj = kmalloc_node_track_caller(obj_size, flags,
    ↪  ...);
    if (obj || !(gfp_pfmemalloc_allowed(flags)))
        goto out;
    ...
    out:
    ...
}
```

Listing 4: Example of check comes after potentially corrupted sink for shared data corruption.

During the manual verification process, we discovered instances where guards are implemented after the sink to validate the correctness of sink operations, as illustrated in Listing 4. In the example, both the `obj_size` and flags parameters to the `kmalloc_node_track_caller` function could be corrupted by the driver. However, after the callsite, the allocated object and flags are both checked. In these cases, we classify such instances as benign and mark them as non-exploitable in our verification process.

4. CIV PoC Analysis

CIV Class	Success Rate
Memory Safety (MEM)	
MEM1: Pointer value	9/10 (90%)
MEM2: Pointer offset/buffer index	8/10 (80%)
MEM3: Type selector	3/3 (100%)
MEM4: Sensitive kernel memory APIs	7/10 (70%)
MEM5: Corrupted string	8/10 (80%)
Decision-making variable (DM)	
DM1: Corrupted guard	10/10 (100%)
DM2: Invalid/wrong error code	6/10 (60%)
DM3: Corrupted loop condition	10/10 (100%)
Arithmetic error (AE)	
AE1: Divided by zero	N/A
AE2: Integer overflow/underflow	6/10 (60%)
Control transfer (CT)	
CT1: SAC	10/10 (100%)
CT2: Lock and never unlock	10/10 (100%)
CT3: Unbalanced alloc/free	10/10 (100%)

TABLE 6: PoC results for CIV classes; n/a for "divided by zero" since no such cases were identified during static analysis.

5. Example PoC Exploit

```
void __bitmap_clear(unsigned long *map, unsigned int
↪  start, int len) {
  // Vulnerable pointer arithmetic
  unsigned long *p = map + BIT_WORD(start);
  const unsigned int size = start + len;
  int bits_to_clear = BITS_PER_LONG - (start %
  ↪  BITS_PER_LONG);
  unsigned long mask_to_clear =
  ↪  BITMAP_FIRST_WORD_MASK(start);
  while (len - bits_to_clear >= 0) {
    // Crash with illegal p or a malicious write
    *p &= ~mask_to_clear;
    len -= bits_to_clear;
    bits_to_clear = BITS_PER_LONG;
    mask_to_clear = ~0UL; p++;
  }
}
// manually constructed maliious "driver" code
int main() {
  unsigned long bitmap[2] = {0xFFFFFFFF, 0xFFFFFFFF};
  // Large value to corrupt pointer calculation
  unsigned int large_start = 0x7FFFFFFF;
  // Crash due to invalid pointer arithmetic
  __bitmap_clear(bitmap, large_start, 1);
  return 0;
}
```

Listing 5: An example PoC exploit.

6. Shared data CIV Classification

TABLE 7: Shared data CIV classification. **Src**: data type passed to interface functions as taint source. **Propagation**: control/data dependency. **Sink**: taint analysis sink.

CIV Class	Description	Src Prop	Sink
Memory Corruption CIVs			
MEM1	Corrupted pointer value	Pointer Data	Ptr dereference operations
MEM2	Corrupted pointer offset/buffer index	Scalar Data	Buffer access or ptr arith operations
MEM3	Corrupted union type selector	Tagged union Data	Uses of the union data
MEM4	Parameters in kernel memory APIs	All Data	Sensitive kernel APIs
MEM5	Corrupted string	String Data	String operations
Data/Control Flow CIVs			
DM1	Corrupted guard	All Data	Branch instructions
DM2	Return invalid/wrong error code	Driver return val Data	Branch instructions
DM3	Corrupted loop condition	Scalar Data, Control	Buffer access inside loop body and loop condition
Arithmetic Exception CIVs			
AE1	Divided by zero	Scalar Data	Divide operations
AE2	Integer over/underflow	Scalar Data	Overflowing arithmetic ops

7. Shared data corruption/Control transfer CIV number after enforcing CFG (P5)

CIV Classes	ixgbe	null_net	sfc	msr	null_blk	nvme	sb_edac	mgag200	usb_f_fs	power_meter	dm_zero
MEMORY SAFETY (MEM1)											
MEM1: Pointer value	41	2	0	0	1	13	2	4	10	0	1
MEM2: Pointer offset/buffer index	27	2	0	7	0	8	0	8	8	2	0
MEM3: Type selector	0	0	0	0	0	0	0	0	0	3	0
MEM4: Sensitive kernel memory APIs	65	4	3	11	2	9	5	22	9	2	2
MEM5: Corrupted string	4	2	2	2	3	3	2	3	1	2	0
DECISION-MAKING VARIABLE											
DM1: Corrupted guard	88	38	62	69	17	13	56	61	43	31	2
DM2: Invalid/wrong error code	1048	17	965	8	11	45	1	24	20	3	0
DM3: Corrupted loop condition	26	1	2	0	0	8	10	11	6	6	2
ARITHMETIC ERROR											
AE1: Divided by zero	0	0	0	0	0	0	0	0	0	0	0
AE2: Integer overflow/underflow	24	2	0	8	0	17	0	3	8	0	0

TABLE 8: Quantify shared data corruption CIV.

	ixgbe	null_net	sfc	msr	null_blk	nvme	sb_edac	mgag200	usb_f_fs	power_meter	dm-zero
SAC	0	0	0	0	0	0	0	0	0	0	0
Lock and not unlock	0	0	0	0	0	0	0	0	0	0	0
Unbalanced allocation/deallocation	2	1	2	0	0	0	0	0	0	0	0

TABLE 9: Control transfer CIVs after enforcing CFI (P5).