Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel

Zhaofeng Li*, Vikram Narayanan†, Xiangdong Chen*, Jerry Zhang*, Anton Burtsev*
*University of Utah
†Palo Alto Networks

Abstract

Rust-for-Linux (RFL) is a new framework that allows development of Linux kernel extensions in Rust. At first glance, RFL is a huge step forward in terms of improving the security of the kernel: As a safe programming language, Rust can eliminate wide classes of low-level vulnerabilities. Yet, in practice, low-level driver code – complex driver interface, a combination of reference counting and manual memory management, arithmetic pointer and index operations, unsafe type casts, and numerous logical invariants about the data structures exchanged with the kernel might significantly limit the security impact of Rust.

This work takes a careful look at how Rust can impact the security of driver code. Specifically, we ask the question: What classes (and what fraction) of vulnerabilities typically found in device driver code can be eliminated by re-implementing device drivers in Rust? We find that Rust can eliminate large classes of safety-related vulnerabilities, but naturally struggles to address protocol violations and semantic errors. Moreover, to be fully eliminated, many classes of flaws require careful programming discipline to avoid memory leaks and runtime panics (e.g., explicit checks for integer overflows and option types), careful implementation of `Drop` traits, as well as correct implementation of reference counting. Our analysis of 240 driver vulnerabilities that are present in device drivers in the last four years, shows that 82 could be automatically eliminated by Rust, 113 require specific programming idioms and developer's involvement, and 45 remain unaffected by Rust. We hope that our work can improve the understanding of potential flaws in Rust drivers and result in more secure kernel code.

1. Introduction

Device drivers and kernel extensions have long been considered one of the primary sources of vulnerabilities in the kernel [19, 56]. Device drivers implement an interface between software and hardware, providing the means by which the upper layers of the kernel (e.g., network protocol stacks, file systems, power management, graphics, etc.) communicate with a diverse range of peripheral devices (e.g., network cards, solid-state and spinning disks, USB devices, PCI bus, etc.). In most cases, device drivers are developed by third-party device vendors that often have only a partial understanding of the kernel's programming and security idioms. Complexity and rapid development pace result in a historically higher rate of flaws and vulnerabilities in the driver code compared to the rest of the kernel. For example, an empirical study of faults in the Linux kernel by Chou et al. found that in 2001, device drivers contained 7 times more faults compared to other subsystems [19]. Despite the fact that improved testing, fuzzing [20, 37, 13, 57, 62, 58, 50, 72, 80, 78, 50] and static analysis improved device driver flaw density significantly in the next 10 years, device drivers remained among the top three subsystems (along with `arch` and `fs`) with the highest flaw densities in the kernel [56]. Our analysis of the CVE database for the period from 2014 to 2023 shows that device drivers account for 16-59% of all vulnerabilities in the Linux kernel (Table 1).

To improve security of the kernel, numerous projects explored the possibility to execute device drivers as isolated subsystems on top of minimal microkernels [40, 5, 36, 28, 48, 6, 7, 41, 26, 39, 33], virtual machines [9, 55, 30, 46, 32, 69, 10], backward-compatible driver execution frameworks [22, 43, 59, 75, 77, 2, 34, 12, 76, 27, 23, 35, 38], and within the kernel itself through both hardware [73, 52, 54, 31] and software [51, 15, 24] mechanisms. Unfortunately, isolation approaches did not take traction in mainline kernels due to high performance overhead and arguably minimal security benefits. Isolation of the driver in which the unmodified driver code is placed in a separate memory compartment, fails to provide practical security benefits due to a broad range of cross-interface attacks [14, 18].

Looking for a more practical solution, the Linux kernel community turned to an alternative approach – low-overhead safe programming languages, and, specifically the possibility of implementing device drivers in Rust. Rust-for-Linux (RFL) is a device driver framework for Linux that enables the development of device drivers in Rust. At a high level, RFL implements Rust bindings for each kernel subsystem, e.g., network, block, non-volatile memory on PCIe (NVMe), etc. These bindings allow Rust device drivers to interact with the kernel interface implemented in C. Moreover, RFL bindings expose a *safe* Rust interface via a collection of wrapper types and high-level abstractions (e.g., iterators) that internally implement run-time checks, ensure correct reference counting, synchronize access, etc. As a result, the driver can be implemented in a safe subset of Rust and hence enjoy the security benefits of spatial and temporal safety.

RFL offers a surprising mix of practicality and security. Early empirical studies suggest that nearly 70% of security issues in the low-level systems code are related to memory safety and hence can be potentially eliminated through the

† Work done at the University of Utah

use of Rust [70, 11]. Hence, while not being isolated in a traditional sense, a device driver implemented in a safe language has a much smaller chance of providing a platform for security attacks on the kernel. Spatial and temporal safety combined with control flow enforcement, eliminate many classes of low-level flaws. Moreover, a strong type system of Rust enforces additional correctness guarantees, e.g., lifetimes of reference counted objects, absence of races through synchronization primitives, and more.

Seemingly, RFL offers a huge step forward in terms of improving the security of the kernel yet at the cost of rewriting device drivers in Rust. A natural question, however, is what security benefits can be achieved by RFL? While the safe subset of Rust prevents a range of low-level vulnerabilities, multiple classes of vulnerabilities, e.g., violations of driver-kernel protocol and semantic data structure invariants, cannot be prevented by safety alone. Moreover, the safe subset of Rust is too restrictive to implement general code and unavoidably requires help from unsafe extensions to implement bindings for unsafe kernel functions, hardware device interfaces, and sometimes even to bypass performance limitations of the safe Rust subset.

In the past, several studies analyzed the use of unsafe code in Rust programs as well as the impact of unsafe code on the overall safety of the system [3, 60, 21, 61]. Xu et al. and Cui et al. conducted empirical studies of CVEs in real-world Rust programs [79, 21]. Following the best practices of experienced Rust teams, Astrauskas et al. formulated three principles for the "secure" use of unsafe code and then conducted an empirical study of whether these principles hold in practice [3]. While these studies provide a number of important insights into the safety and its security implications, they focus on the use of Rust in typical user applications. We argue that the complex execution environment of the kernel combined with the semantically rich kernel-driver interface poses a unique set of challenges for developing secure Rust code.

Device drivers implement the logic of the driver interface in a concurrent and fully reentrant execution environment of the kernel. The kernel invokes the driver interface on multiple CPUs while at the same time preempting execution of driver functions with asynchronous interrupts. Hardware unplug events can trigger a driver tear-down protocol that may deallocate the internal state used concurrently by other driver interfaces. To support a range of low-level optimizations, the driver-kernel interface utilizes a collection of unsafe programming idioms, e.g., co-location of multiple data structures with subsequent unsafe type casts, nesting of data structures with `container_of` and `member_of` idioms, tagged and anonymous unions, polymorphism via void pointers, overloading pointers with negative error codes, etc. Finally, the driver code is responsible for maintaining a multitude of high-level semantic invariants ranging from security checks to high-level invariants about the layout and state of each data structure (e.g., the `skb->tail` pointer of the `skb` data structure that represents a network packet in the Linux kernel stays within the bounds of allocated `skb->data` region).

Year	CVEs			SLOC (mil)	
	Total	Driver	%	Kernel	Driver
2014	130	22	16.9%	5.2	7.5
2015	80	15	18.8%	5.3	8.5
2016	216	65	30.1%	5.6	9.2
2017	452	267	59.1%	5.8	10.9
2018	180	44	24.4%	5.9	11.5
2019	291	154	52.9%	6.1	12.1
2020	129	38	29.5%	6.5	13.3
2021	162	42	25.9%	6.7	14.5
2022	310	92	29.7%	7.2	16.6
2023	263	75	28.5%	7.4	17.2

TABLE 1: CVEs in the Linux kernel versus those in its device drivers and the SLOC count of one of the major kernel release in that year.

Our work takes a careful look at how Rust can impact the security of the driver code. Specifically, we ask the question of what classes (and what fraction) of vulnerabilities found in device drivers can be eliminated by re-implementing device drivers in Rust? Since device drivers cannot be implemented in the safe subset of Rust and require some unsafe functions and code blocks, what classes of vulnerabilities are introduced or remain unmitigated by unsafe Rust extensions? What are the best practices for the use of safe and unsafe Rust in the kernel to eliminate the biggest fraction of vulnerabilities? And finally, since mitigation of some vulnerabilities, even in the safe subset of Rust requires the use of specific Rust idioms, we study whether existing RFL Rust drivers follow these idioms.

To answer these questions, we study a set of 240 vulnerabilities discovered in Linux device drivers in the last four years. We find that Rust can eliminate large classes of safety-related vulnerabilities but naturally struggles to address protocol violations and semantic errors. Moreover, to be fully eliminated, many classes of flaws require careful programming discipline to avoid memory leaks and runtime panics (e.g., explicit checks for integer overflows and option types), careful implementation of `Drop` traits, as well as correct implementation of reference counting. Our analysis shows that 82 vulnerabilities would be automatically eliminated if device drivers were rewritten in Rust, 113 would require specific programming idioms and developer involvement, and 45 remain present even in RFL drivers. We hope that our analysis can improve the security of the Linux kernel through a collection of development practices that are needed to amplify the security impact of Rust.

2. Background and Related Work

2.1. Rust

Rust is a programming language designed to provide *safety* while maintaining performance characterisics of unsafe languages [29]. By using a compile-time borrow checker, in contrast to commodity safe languages, Rust implements memory safety without a managed runtime and garbage collection. At a high level, Rust introduces the concept of *ownership*. Each value in Rust is assigned a single owner, which is the scope that it is defined in. The ownership of a value can also be *moved* to another function through

a function call, in which case the caller loses all access to the value after the call. During a value's lifetime, the owner may derive references (aliases) that grant temporary access to the value. A reference in Rust can be either immutable (`&T`) or mutable (`&mut T`). The borrow checker statically enforces several invariants, the most important being that a value can have *either* a single mutable reference or multiple immutable references. The idea of ownership allows static reasoning about the lifetime of each object on the heap and for explicit deallocation. At the end of its lifetime, the value is automatically dropped with its storage reclaimed.

Restricted ownership of Rust is too constraining for the development of general programs. For example, it is impossible to implement a doubly linked list relying only on basic ownership semantics. Hence, Rust makes a practical choice to provide an escape hatch known as *unsafe* Rust. In `unsafe` blocks, the borrow checker is still in effect but the language allows unsafe operations like dereferencing raw C pointers. The Rust standard library introduces a collection of types that internally use unsafe extensions and rely on runtime checks to establish safety for the exposed interface. To construct a doubly-linked list, one can combine `Rc<T>` and `RefCell<T>`. The former provides shared ownership and automatic deallocation through reference counting, and the latter allows mutating a value through immutable references (known as *interior mutability*).

2.2. Rust for Linux (RFL)

The idea of bringing Rust into the Linux kernel roots back to a hobbyist project started in 2013 [1]. RFL was officially added to the kernel in 2021. Developers started rewriting complex device drivers such as Binder [63], Nvme [67], network drivers [64] and also implemented new GPU drivers [66] in Rust.

In Linux, device drivers are implemented as dynamically loaded kernel extensions. Specifically, device drivers are object files (ELF binaries) that are compiled and linked separately from the core kernel. To illustrate the development of a Linux driver in Rust, we consider the example of an NVMe device driver that provides a high-speed interface to non-volatile memory attached over the PCIe bus.

At a high-level, RFL relies on bindgen [44] to automatically generate foreign function interface (FFI) bindings from the kernel header files (Figure 1 line 3 to 8). Bindgen takes in a C header file and automatically translates the C structures and function declarations to Rust-compatible structures and signatures (e.g., the `__pci_register_driver()` is a C-compatible Rust function generated by bingen).

The kernel developer then uses the bindings to construct a "kernel" crate that acts as a trusted layer between unsafe kernel and a device driver implemented in safe Rust. The kernel crate is written in idiomatic Rust, utilizing Rust features such as traits and templates to create a convenient abstraction for the actual Rust driver. For example, to implement registration of the driver, developer of the kernel crate implements the `DriverOps` trait (Figure 1 line 13 to 28). To register a driver with the PCI subsystem, the implementation of the safe `register` method calls the C `__pci_register_driver` function through unsafe Rust.

```
1  // Auto-generated: rust/bindings/bindings_generated.rs
2  extern "C" {
3     pub fn __pci_register_driver(
4        arg1: *mut pci_driver,
5        arg2: *mut module,
6        mod_name: *const core::ffi::c_char,
7     ) -> core::ffi::c_int;
8  }
9
10 // Safe Abstraction: rust/kernel/pci.rs
11 pub struct Adapter<T: Driver>(T);
12
13 impl<T: Driver> driver::DriverOps for Adapter<T> {
14    type RegType = bindings::pci_driver;
15
16    fn register(
17       pdrv: &mut Self::RegType,
18       name: &'static CStr,
19       module: &'static ThisModule,
20    ) -> Result {
21       // ...
22       pdrv.id_table = T::ID_TABLE.as_ref();
23
24       // SAFETY: `pdrv` is guaranteed to be a valid `
             RegType`.
25       to_result(unsafe {
26          bindings::__pci_register_driver(pdrv as _,
                module.0, name.as_char_ptr())
27       })
28    }
29 }
```

Figure 1: Partial implementation of the PCI driver adapter in RFL

Below, we discuss typical RFL idioms that are used to provide such safe interface on top of unsafe bindings.

Driver interfaces In Linux, each driver is responsible for registering a collection of function pointers with the kernel that implement the driver's interface. C code uses data structures with function pointers to implement driver interfaces. RFL, however, uses traits to provide type- and lifetime-checked versions of such interfaces. For example, the Rust NVMe driver uses `block::mq::Operations` trait to implement the interface of the multi-queue block IO subsystem (implemented with `struct blk_mq_ops` in C drivers).

Lifetimes Typically, the kernel follows two distinct patterns of managing lifetimes of dynamic data structures exchanged with the driver: reference counting and regular deallocation based on the invocation context (i.e., the object is known to be deallocated when a specific kernel or driver function is invoked). RFL is designed to support both paradigms in a safe manner. For reference-counted objects in C, RFL implements safe wrappers around the C functions that perform reference counting, like `get_device()` which increments the counter given a `struct device` pointer and its counterpart `put_device()` which decrements the counter. In RFL, the C `struct device` is wrapped by the safe Rust `Device` type which integrates with the C reference counting infrastructure by implementing the `AlwaysRefCounted` trait. In its `AlwaysRefCounted` implementation, the `inc_ref()` and `dec_ref()` methods call `get_device()` and `put_device()`, respectively.

For C data structures with specific functions for deallocation, RFL implements `Drop` for their safe wrappers. For example, `net::phy::Registration` in RFL models a registration of a PHY driver. Its `Drop` implementation calls `phy_drivers_unregister()` to automatically unregister the

driver from the kernel when the handle goes out of scope.

Driver registration and de-registrtion protocols RFL encourages the use of Rust `Drop` method to provide automatic implementation of driver registration and cleanup protocols (i.e., the cleanup is implemented inside `Drop`). However, it should be noted that Rust does not consider memory leaks to be safety violations. Even in the safe subset of Rust it is possible to "forget" an object to surrender the ownership without calling the `Drop` implementation. Furthermore, global variables (`static`) in Rust have lifetimes that span the entire program and `Drop` is not automatically called.

2.3. Vulnerabilities in Rust

Early empirical reports estimated that nearly 70% of security issues in the low-level code that are typically assigned a CVE are related to memory safety and hence could potentially be eliminated through the use of a safe programming language like Rust [70, 11, 65]. Unfortunately, while the safe subset of Rust prevents a range of low-level vulnerabilities, the security of the system rests on careful use of unsafe Rust. Qin et al. conducted a manual analysis of 850 instances of unsafe Rust to understand the reasons for using unsafe code in real programs as well as to analyze errors related to the use of unsafe Rust and concurrency primitives [61]. Subsequently, multiple studies tried to analyze the implications of unsafe Rust in commodity code [3, 60, 21]. For example, in their follow-up work, Qin et al. analyzed errors in Rust programs on the boundary of safe and unsafe code, e.g., a buffer overflow due to the use of incorrect bounds passed from the safe code and used without checks in the unsafe subset, races, etc. [60]. Xu et al. and Cui et al. conducted empirical studies of CVEs in real-world Rust programs [79, 21]. Following the best practices of experienced Rust teams, Astrauskas et al. formulate three principles for the "secure" use of unsafe code and then conduct an empirical study of whether these principles hold in practice [3]. An interesting observation is that 44.6% of unsafe function definitions in the Rust ecosystem are bindings to foreign function definitions used for linking against unsafe C code – something what we observe on the device-kernel interface as well. Evans et al. conducted a study of how unsafe code propagates through the program, i.e., how many functions transitively depend on unsafe code and hence become "possibly unsafe" [25].

In contrast to prior work, which provides many insights of how unsafe Rust is used in real programs and what are the typical cases for errors in both safe and unsafe Rust, our study aims to understand the impact Rust can have on the security of the Linux kernel (along with the programming practices which are essential for improving correctness of both safe and unsafe device driver code).

3. Methodology

Our study aims to answer the following specific questions: **Q1.** *What classes (and which fraction) of typical device driver vulnerabilities can be eliminated by re-implementing device drivers in Rust?* **Q2.** *How unsafe Rust is used in RFL and what typical vulnerabilities are introduced by it?* **Q3.** *Do existing Rust device drivers follow best practices for safety and security in both safe and unsafe code?*

To answer **Q1**, we collect the set of Linux kernel CVEs from the device driver code (i.e., under the `./drivers` subtree of the kernel tree). Specifically, we take the data from the CVE database from 2020 to 2024 and filter driver vulnerabilities by the source path [49]. To ensure a general approach to classification, we create a new taxonomy of driver vulnerabilities. Our taxonomy is based on the analysis of common vulnerabilities in the Linux kernel by Chen et al. [16] but extended with novel vulnerability classes that we observe in our vulnerability set. At a high level, we classify vulnerabilities into three big classes: 1) errors related to safety, 2) protocol violations (i.e., violations of expected behavior of the subsystems with respect to lock acquisition, order of function invocations, etc.), and 3) semantic violations (i.e., violations of high-level assumptions about the program state). We then manually analyze all device driver vulnerabilities and classify them (each vulnerability is analyzed by at least two people to ensure confidence in classification). We place each vulnerability in a single class to avoid double counting. For each class, we analyze whether a vulnerability can be eliminated by rewriting the driver code in Rust. Below, we label such cases as **Yes**. For some vulnerabilities, naive Rust implementation reduces the severity of the vulnerability, e.g., from an unrestricted read or write memory access to a denial-of-service due to panic, and requires additional programmer effort to handle the panic. We discuss the possible mitigation and label such cases as **Yes+P**. If Rust does not help to address a specific subset of vulnerabilities we label them as **No**.

To answer **Q2**, we analyze all instances of unsafe code in several recent device drivers implemented in Rust. We introduce a taxonomy that allows us to classify typical uses of unsafe code. Then we use Semgrep [71], a semantic grep tool that allows us to explore the abstract syntax tree of the program to mechanically classify uses of unsafe Rust in both the driver code and in the kernel crate.

To answer **Q3**, for each vulnerability class (and especially the ones that require programmer effort to eliminate vulnerability through a specific programming idiom) we analyze whether this idiom is followed in the code of existing Rust drivers.

4. Driver Vulnerabilities

To understand the impact of RFL on the security of the Linux kernel, we perform an analysis and classification of all driver CVEs in the last four years.

4.1. Safety Violations

Safety violations are a broad set of flaws related to improper use of indexes and pointers, integer overflows and underflows, divisions by zero, violation of object lifetimes, improper type casts, etc.

Buffer overflow Buffer overflows are one of the most common bugs in the Linux kernel, accounting for 20% of all CVEs from 2020 to 2024. Buffer overflow happens when

```
1  int detach_capi_ctr(struct capi_ctr *ctr) {
2   ...
3  + if (ctr->cnr < 1 || ctr->cnr - 1 >= CAPI_MAXCONTR) {
4  +  err = -EINVAL;
5  +  goto unlock_out;
6  + }
7  +
8   if (capi_controller[ctr->cnr - 1] != ctr) {
9     err = -EINVAL;
10    goto unlock_out;
11  }
```

Listing 1: Array out-of-bound read (CVE-2021-43389)

the code accesses memory beyond the allocated size due to incorrect computation of the array index (e.g., CVE-2021-43389), improper length validation (e.g., CVE-2021-42327), mismatch between allocated and the intended size of the object on the heap (e.g., CVE-2021-31916), etc. For example, line 8 of Listing 1 illustrates a patch for an array out-of-bound vulnerability (CVE-2021-43389), which adds explicit bounds validation for the controller number `ctr->cnr` before indexing into the `capi_controller` array.

Rust ► **(Yes+P)** Intuitively, Rust safety guards against buffer overflows by adding implicit bounds checks on each access. In practice, however, a straightforward rewrite of the driver code in Rust reduces the impact of out-of-bound access to a panic, which is still a critical denial-of-service attack. Hence, even in Rust, the developer has to add explicit bounds checks to avoid the panic (lines 3 to 7 in Listing 1).

Moreover, some buffer overflows (e.g., CVE-2021-42327) have a flavor of logical error in which a wrong variable is mistakenly used to specify the size of the buffer. In Listing 2, the `size` is used instead of `wr_buf_size` as the length of the `wr_buf` buffer causing buffer overflow Even in safe Rust, such logical error can lead to unsafe behavior if the variable flows into an unsafe function that performs operation on the buffer, e.g., a memory copy. In our example of CVE-2021-42327, the `parse_write_buffer_into_params()` eventually calls into `memcpy()`. Therefore, to ensure safety, a careful re-design of the driver abstractions is required. For example, the buffers need to use safe Rust data structures like arrays, slices, and vectors, and access to linked lists should be abstracted with iterators. Unfortunately, often establishing the safety of such unsafe accesses (e.g., establishing the safety of an iterator requires a non-trivial contract with an unsafe C world which potentially can lead to vulnerabilities).

Rust ► **(No)** One vulnerability in our dataset, CVE-2023-6238, triggers a buffer overflow via the hardware interface. Specifically, the user is able to specify the size of the DMA buffer for the device to use. This buffer is copied to the kernel, and if the buffer size is smaller than the device expects, the device will overwrite kernel memory causing a random crash.

Memory leak A memory leak occurs when all pointers to an allocation are discarded without freeing the allocation itself, leaving no way to free it later. They can be used by attackers to exhaust system memory and cause a denial of service. For example, in Listing 3, the macvlan driver omitted a deallocation when a packet is consumed in a

```
1  static ssize_t dp_link_settings_write(struct file *f,
2     const char __user *buf, ...) {
3  - if (parse_write_buffer_into_params(wr_buf, size,
4  + if (parse_write_buffer_into_params(wr_buf, wr_buf_size,
5                 (long *)param, buf,
6                 max_param_num,
```

Listing 2: Out-of-bound access (CVE-2021-42327)

```
1   if (macvlan_forward_source(skb, port, eth->h_source)) {
2  + kfree_skb(skb);
3     return RX_HANDLER_CONSUMED;
4   }
```

Listing 3: Memory leak (CVE-2022-3526)

specific configuration.

Rust ► **(Yes)** Heap-allocated types managed by Rust like `Box<T>` and `Arc<T>` implement automatic deallocation when all references are dropped. Safe wrappers of reference-counted C structs are typically consumed via `ARef<T>` types which force the implementation of a corresponding `dec_ref` method to decrement the counter in C.

Rust ► **(Yes+P)** To interface with custom C data structures that often have non-trivial cleanup and deallocation logic, the safe wrappers must implement the `drop` method correctly. Listing 19 is an example of a complex drop method in one of the current RFL drivers, which has to carefully invalidate the TLB mappings.

Integer overflow, underflow, and division by zero Integer overflow or underflow happens when an arithmetic operation results in a number that cannot be represented with a given number of bits. The typical cause is incorrect usage of type, such as use of a 32-bit integer type instead of 64-bit type or using a signed type instead of an unsigned type. For example, in Listing 4, the `words` variable is typed incorrectly as `int` (32 bits wide) rather than `usize_t` (64 bits wide), resulting in two potential bugs. First, the variable `words` suffers from an integer underflow when `count/4` is greater than $2^{32} - 1$. Second, the 3rd argument in `copy_from_user` (line 9) suffers from integer overflow when `words*4` is greater than $2^{31} - 1$ because the conversion between 32 bits signed to 64 bits unsigned will carry over the signed bit as the last bit for a 64-bit integer, causing the size of the `copy_from_user` to be extremely large.

Rust ► **(Yes)** Rust addresses a subset of overflow bugs through automatic type inference. For example, instead of having to explicitly specify the type of a variable, programmers writing idiomatic Rust let the compiler automatically infer the correct type:

```
1  let word = count / 4;
```

```
1  static ssize_t pxa3xx_gcu_write(...) {
2  ...
3  - int words = count / 4;
4  + size_t words = count / 4;
5  ...
6     ret = copy_from_user(buffer->ptr, buff, words * 4);
7  ...
8  }
```

Listing 4: Integer overflow (CVE-2022-39842)

```
1  static int spl2sw_nvmem_get_mac_address(struct device *dev, ...)
2   u8 *mac;
3   ...
4    if (!is_valid_ether_addr(mac)) {
5  -  kfree(mac);
6    dev_info(dev, "Invalid mac address in nvmem (%pM)!\n", mac);
7  +  kfree(mac);
8    return -EINVAL;
9   }
10  ...
11 }
```

Listing 5: Use-after-free due to incorrect ordering in CVE-2022-3541

If a developer accidentally picks a wrong type by specifying it explicitly, the compiler reports a type mismatch:

```
let word: u32 = count / 4; // compiler error
```

Rust ▶ **(Yes+P)** Despite support for type inference, there are cases when an explicit type conversion to a larger type is still required to avoid the overflow. For example, a recent Apple AGX GPU driver implemented in Rust had an overflow bug caused by integer multiplication. The expression `4 * rgn_size * mtiles` can overflow a 32 bit value, hence requiring an explicit conversion to a 64-bit type (`usize`):

```
- let tilemap_size = (4 * rgn_size * mtiles * layers) as usize;
+ let tilemap_size = (4 * rgn_size * mtiles) as usize * layers
      as usize;
```

By default, RFL enables runtime overflow checks through a compiler flag (`-C overflow-checks`), which triggers a panic when an arithmetic operation results in overflow. Unfortunately, panics still result in denial-of-service attacks. Hence explicit range checks are required to avoid panics. Rust provides safe wrappers for arithmetic operations (`checked_add`, `checked_sub`, `checked_mul`, and `checked_div`) that return `None` if the operation results in integer overflow, underflow, or division by zero. This allows the driver to gracefully handle the overflow instead of panicking.

Use-after-free Use-after-free (UAF) is a vulnerability in which a resource is accessed after it has been deallocated. For example, Listing 5 shows the resource `mac` gets freed in line 5, and then accessed by `dev_info` in line 6. This bug can be exploited by an attacker to gain control of the system.

Rust ▶ **(Yes)** Rust type system prevents a variable from being referenced outside of its lifetime. In an equivalent Rust program, the lifetime of `mac` will be extended up to the last access by `dev_info()`. If the `drop()` method is used to terminate the lifetime of `mac` explicitly, the future references to `mac` will generate a compiler error.

Rust ▶ **(Yes+P)** An option type (`Option<T>`) holds either `Some(T)` or `None` and is often used to handle NULL-able types. Instead of using a sentinel value that can be ignored, option types force developers to explicitly handle the case where the object is not present (hence checking for whether the pointer is valid). However, panics are possible unless careful programming discipline of explicit checking for `None` is implemented.

Null pointer dereference Null pointer dereference can occur when the kernel accesses uninitialized memory (e.g., CVE-2020-15437), or dereferencing into newly allocated memory without checking for allocation failure (e.g., CVE-2022-3115).

Rust ▶ **(Yes+P)** In general, safe Rust prevents most cases of null pointer dereference with the ownership model where having a valid reference to a value provides type invariant guarantees. Safe Rust does not allow uninitialized variables. Explicitly NULL-able values are modeled as distinct types (`Option<T>`) in the type system. When option types are used to represent explicitly NULL-able values, panics can still occur if the programmer calls `unwrap()` on a `None` value. Hence, careful programming discipline is needed to avoid panics.

```
1  --- b/drivers/net/can/slcan.c
2  +++ b/drivers/net/can/slcan.c
3  static void slc_bump(struct slcan *sl) {
4     struct can_frame cf;
5
6  -  cf.can_id = 0;
7  +  memset(&cf, 0, sizeof(cf));
8     ...
9
10    skb_put_data(skb, &cf, sizeof(struct can_frame));
```

Listing 6: Information disclosure (CVE-2020-11494)

```
1  --- a/include/uapi/linux/usbdevice_fs.h
2  +++ b/include/uapi/linux/usbdevice_fs.h
3  struct usbdevfs_connectinfo {
4     unsigned int devnum;
5     unsigned char slow;
6     (3-byte padding inserted by compiler)
7  };
8  --- a/drivers/usb/core/devio.c
9  +++ b/drivers/usb/core/devio.c
10 static int proc_connectinfo(struct usb_dev_state *ps, ...) {
11 -  struct usbdevfs_connectinfo ci = {
12 -     .devnum = ps->dev->devnum,
13 -     .slow = ps->dev->speed == USB_SPEED_LOW
14 -  };
15 +  struct usbdevfs_connectinfo ci;
16 +
17 +  memset(&ci, 0, sizeof(ci));
18 +  ci.devnum = ps->dev->devnum;
19 +  ci.slow = ps->dev->speed == USB_SPEED_LOW;
20
21    if (copy_to_user(arg, &ci, sizeof(ci)))
22       return -EFAULT;
23    ...
24 }
```

Listing 7: Information disclosure via struct padding (CVE-2016-4482)

Memory disclosure Memory disclosure vulnerabilities leak sensitive data from the kernel. Specifically, the allocation of new data structures on the heap might accidentally leak sensitive data. Information disclosure occurs when the uninitialized memory is not completely overwritten and gets copied to userspace or sent over the network. In Listing 6, the `can_frame` data structure gets partially initialized on the stack. Other struct fields contain uninitialized data, which is then sent over the CAN bus.

Rust ▶ **(Yes)** Rust enforces value initialization and all fields of a struct must be initialized to known values before the struct can be read.

Rust ▶ **(No)** Data structures may have implicit padding inserted by the compiler to ensure alignment. Such padding is transparent to the developer and can be left uninitialized even when all fields are explicitly set. In Listing 7, a

`usbdevfs_connectinfo` struct is allocated on the stack with all fields explicitly initialized. However, the compiler inserts 3 padding bytes at the end of this struct which still contain uninitialized data. To mitigate such problems, a driver developer writing C code needs to `memset()` stack allocations or use `kzalloc()` to acquire zero-initialized memory on the heap. Similar to C, Rust is susceptible to the padding problem, and requiring zero-initialization in all scenarios may not be feasible. Functions can then use trait bounds to constrain the allocator used for objects passed to their arguments.

Confused pointer error type (`PTR_ERR`) A somewhat unusual programming idiom widely used in the kernel is to encode errors for pointer return types as "negative" pointers with the most significant bit set. This results in a common error as checking for a non-`NULL` pointer will succeed even though it represents an error in this encoding. As a result, instead of following an error pass, the code uses an invalid pointer triggering unsafe memory accesses.

Rust ▶ **(Yes+P)** RFL leverages high-level enum types to represent pointer errors explicitly as `Result<T, E>`. Similar to option types, the caller must explicitly handle or ignore the potential error (`E`) to get the expected value (`T`).

Unsafe type cast Kernel often uses unsafe type casts for tagged unions and void pointer types to implement polymorphism. The kernel interprets the type of the data structure based on the context or the tag value associated with the union type. An error in type computation, e.g., wrong context or an erroneous tag, the driver code accesses a data structure using the wrong interpretation. In CVE-2022-36402, the `vmwgfx` driver converts an integer to `enum vmw_ctx_binding_type` using integer arithmetics. Denial-of-service occurs when userspace provides an invalid shader type as the input.

Another case of unsafe pointer casts is related to a popular kernel idiom: the `container_of` macro, which provides access to the parent data structure from a given field. This creates an assumption of the enclosing type which may not be correct in all execution contexts. In CVE-2023-1076, the `tap` driver calls `sock_init_data()` to initialize the low-level `struct sock` type given a `struct socket`. However, the function assumes that the `struct socket` passed is part of a `struct socket_alloc`. This assumption doesn't hold for `tap_open`'s usage, causing the `sk_uid` field to be initialized to an unrelated value.

Rust ▶ **(Yes)** The Rust type system enforces a strict type discipline that rules out unsafe type conversions. For example, conversion from an integer tag encoding the type provided by the user will require an explicit function that will be forced to handle all possible options of the enumerated type.

4.2. Protocol violations

Protocol violations are a set of vulnerabilities that leverage errors in the communication protocol between the driver and the core kernel subsystems. The errors can stem from missing or re-ordering protocol steps, breaking reference counting discipline, failing to correctly implement synchronization, violating rules of preemption and interrupt disabling, invoke interfaces of the kernel repeatedly, failing to yield, and so on.

```
1 static int rkvdec_remove(struct platform_device *pdev) {
2   struct rkvdec_dev *rkvdec = platform_get_drvdata(pdev);
3   cancel_delayed_work_sync(&rkvdec->watchdog_work);
4   rkvdec_v4l2_cleanup(rkvdec);
5   ...
6 }
```

Listing 8: Missing deinitialization routine (CVE-2023-35829)

Protocol steps Every driver follows a protocol, in which it creates and registers interfaces with the kernel (e.g., timers, work queues, driver-specific interfaces like character or a block device). Typically, these interfaces remain operational while the driver is active and have to be unregistered upon tear down and sometimes during power mode transitions. Naturally, errors in protocol implementation can result in unsafe memory accesses like use-after-free, null-pointer dereferences, double frees, etc.

Listing 8 shows an example that the programmer failed to deregister the `work` object and could potentially lead to a use-after-free (UAF) bug based on the interleaving of the deinitialization thread and the workqueue callbacks thread (the kernel accesses memory of the driver registered as part of the workqueue after the driver is already unloaded). Line 3 fixes the UAF bug by canceling the work in the deinitialization routine of the driver.

Rust ▶ **(Yes+P/No)** In general protocol violations are impossible to fix with Rust. Some specific errors can be avoided by using higher-level constructs like guards and safe reference counted types that explicitly implement correct de-registration as part of the `Drop` method, but of course a wide variety of errors remains. Arguably, an even larger set of errors can be avoided by relying on session types that explicitly encode allowed protocol transitions [74]. Unfortunately, there is no ergonomic way of using session types in Rust which hinders their adoption.

Reference counting Multiple subsystems in the kernel implement lifetime management of data structures exchanged with the driver via reference counting. For example, a network traffic classifier maintains multiple network namespaces and uses reference counters to determine the liveness of each network namespace. An error in the reference counting protocol, i.e., an extra or missing increment of a reference counter can alter the lifetime of the object and lead to memory leaks, use-after-free, and so on. For example, in CVE-2022-29581, the reference counter for current network namespace is mistakenly reduced in the error path, which can lead to an use after free of the `struct net`.

Rust ▶ **(Yes+P)** To implement safe reference counting (and ownership of objects shared across multiple contexts), Rust provides reference counted types like `Rc<T>` and `Arc<T>`, which represent a handle to a reference-counted object. The counter is automatically incremented when the handle is cloned, and decremented as it goes out of scope. RFL follows the same idiom with the `AlwaysRefCounted` trait whose implementations should contain the required calls to

```
// drivers/tty/tty_jobctrl.c
static int tiocspgrp(struct tty_struct *tty,
                     struct tty_struct *real_tty, ...)
{
...
-  spin_lock_irq(&tty->ctrl_lock);
+  spin_lock_irq(&real_tty->ctrl_lock);
   put_pid(real_tty->pgrp);
   real_tty->pgrp = get_pid(pgrp);
-  spin_unlock_irq(&tty->ctrl_lock);
+  spin_unlock_irq(&real_tty->ctrl_lock);
...
}
```

Listing 9: Lock on the wrong object (CVE-2020-29661)

```
// drivers/scsi/scsi_transport_iscsi.c
int show_transport_handle(struct device *dev,
   struct device_attribute *attr, char *buf) {
   struct iscsi_internal *priv = dev_to_iscsi_internal(dev);
+
+  if (!capable(CAP_SYS_ADMIN))
+     return -EACCES;
   return sprintf(buf, "%llu\n", (unsigned long long)iscsi_handle
         (priv->iscsi_transport));
}
```

Listing 10: Missing permission checks (CVE-2021-27364)

increment or decrement the counters in the Linux kernel. Rust therefore prevents a subset of reference counting errors ruling out unsafe memory accesses, e.g., use-after-free.

Rust ► **(No)** Note, however, that even safe Rust can leak memory with the `forget` function that consumes the ownership without running the destructor. This can skip the counter decrement and result in a denial of service (memory exhaustion) or more severe attacks like protocol violations similar to Listing 8 discussed above in which destructor is responsible for implementing resource deallocation protocol and can potentially trigger unsafe behavior.

Race conditions Due to the concurrent nature of Linux device interfaces, data races are very frequent (15% of CVEs in our data set). Data races manifest as a range of subtle bugs and unsafe behaviors, such as use-after-free, double-free, and null-pointer dereferences.

Rust ► **(Yes)** Rust type system eliminates nearly all data race vulnerabilities. In most cases, a non-trivial amount of human reasoning is required to get the synchronization scheme right such as knowing what data structure to use, what synchronization pattern to utilize. Nevertheless, we observe that Rust forces developers to avoid some common mistakes that frequently occur in C, such as failure to use synchronization primitives altogether, lack of atomicity while managing reference counted objects, accidental use of a wrong lock, etc. Listing 9 shows an example where the driver acquires incorrect lock to access the fields within the locked object. This is impossible in Rust as there is no way to access an object protected by a synchronization primitive, e.g., `Mutex<T>`.

Rust ► **(No)** Even though safe Rust can eliminate most data races, deadlocks are still possible. CVE-2023-2269 is one example of a possible deadlock in our dataset.

Sleeping in atomic contexts A well-known kernel invariant is that the kernel monopolizes the CPU and can only perform operations that cannot sleep during certain *atomic* contexts (e.g., holding a spinlock, executing an interrupt handler). Violating this invariant may block the CPU for a long time or result in a deadlock or a system hang. On a multicore system, sleeping in an atomic context does not always result in a system hang or crash, thus making it hard to statically detect such conditions. For example, during the invocation of kernel memory allocation functions (`kmalloc()`) under atomic contexts require passing the `GFP_ATOMIC` flag. Syzkaller reported a violation of this invariant in the `n_gsm` driver as CVE-2023-31082. In another example, a potential SAC bug has been reported in the kernel mailing list for the earlier version of the RFL NVMe driver implemented in Rust [68].

Rust ► **(No)** Reasoning about the atomicity of the invocation context is challenging [42]. Rust on its own would not solve these problems and would need developers' attention to avoid these mistakes when implementing a driver under RFL.

Failure to yield Failure to yield is a bug when the code fails to invoke one of the cooperative scheduling routines to yield the CPU which results in a denial of service, freezes of individual cores, and in the worst case, complete system freeze. The major causes of failure to yield vulnerabilities include infinite loops (CVE-2019-3900) and semantic errors failing to invoke a yielding function. For example, in CVE-2015-5364 the driver fails to invoke the `cond_resched()` function.

Rust ► **(No)** Rust cannot prevent this type of vulnerability as it is a logical error that cannot be enforced by neither a strong type system nor higher-level programming techniques.

4.3. Semantic Violations

Any program maintains a number of invariants (often implicit) about its state. Some invariants are simple, e.g., a pointer field of a data structure is not equal to NULL after the data structure is properly initialized. More complex invariants reflect the semantic properties of data structures, e.g., validity of references, relationships between indexes and buffers, high-level invariants about ordering, lack of cycles, single membership, balancing, etc. A violation of a semantic invariant results in an incorrect system behavior and often, especially in an unsafe environment, provides an attacker with powerful exploitation primitives: the ability to read and write memory, alter control flow, etc.

Security and permissions While not frequent, in some cases, device drivers are responsible for enforcing security policies of the kernel, e.g., checking permissions and capabilities on access to specific resources. For example, Listing 10 `show_transport_handle()` is a privileged operation that should only be accessible to processes with administrative privileges. However, the code fails to implement this security check.

Rust ► **(No)** Security vulnerabilities are logical errors that cannot be prevented by Rust. Arguably, some instances can be prevented by wrapping functions of the driver and implementing security checks in a trusted kernel crate, but of course such approach is limited to the checks that can be

```
--- a/fs/ntfs3/attrib.c
+++ b/fs/ntfs3/attrib.c
   if (!attr_b->non_res) {
-     u32 data_size = le32_to_cpu(attr->res.data_size);
+     u32 data_size = le32_to_cpu(attr_b->res.data_size);
```

Listing 11: Logic error due to the use of an incorrect variable (CVE-2022-4842)

```
void imx_register_uart_clocks(unsigned int clk_count) {
   ...
   imx_uart_clocks = kcalloc(clk_count, sizeof(struct clk *),
        GFP_KERNEL);
+  if (!imx_uart_clocks)
+     return;
   ...
```

Listing 12: Null pointer dereference due to potential allocation failure in CVE-2022-3114

performed before and after the function is invoked.

Logic errors Logic errors are failures to implement high-level behavior of the program, i.e., violate specific algorithms or break a high-level property of a data structure like a linked list. For example, CVE-2022-4842 is a vulnerability in the ntfs3 driver that resulted from the use of the wrong variable in place of a similarly-named one (Listing 11). Even though it manifests as a null pointer dereference, the vulnerability is caused by a logic error and cannot solved by language features like `Option<T>`.

Rust ► **(No)** High-level programming idioms can protect against certain classes of errors as we discussed above. Yet in general, Rust has no mechanisms to enforce the correct behavior of the driver code.

Missing return value check Missing return value checks can result in a null pointer dereference (e.g., CVE-2022-3115, CVE-2022-3114, CVE-2022-3113). For example, in Listing 12, `kcalloc()` is called but the argument goes unchecked, which results in a NULL pointer dereference if allocation fails.

Rust ► **(Yes)** Rust eliminates missing return value bugs through its support for the `#[must_use]` attribute which forces the compiler to check that the return value is used by the caller. For example, in RFL allocation functions return `Result<T>` marked as `#[must_use]`, requiring the developer to explicitly handle allocation failures.

Loop termination In many cases loops in driver code use user and sometimes device-provided values as the termination criteria. An absence of range checking can potentially lead to infinite loops and thus denial-of-service. In CVE-2022-48635, a command with zero `iovec` iterator size will cause `iomap_iter` to always return 1, resulting in an infinite loop. A more involved example is CVE-2024-26603, in which the potential infinite loop occurs when the kernel tries to restore floating point registers from the userspace. If the kernel fails to restore the registers, it uses `fault_in_readable` to check the number of faults that have occurred while reading the restore region, and only retries if no faults have occurred. However, the range of memory that is checked is passed from userspace and can be smaller than what is actually accessed, potentially resulting in a false negative. The kernel can thus be stuck in an infinite retry loop if part of the restore region is unreadable.

Rust ► **(No)** Rust does not guarantee loop termination, and hence cannot prevent such vulnerabilities.

4.4. Analysis

To answer the question of which classes and what fraction of typical device driver vulnerabilities can be eliminated by re-implementing device drivers in Rust, we analyzed CVEs for the past four years (2020-2024). Specifically we identified 240 vulnerabilities discovered in Linux device drivers and classified them into three large classes: safety (113 vulnerabilities), protocol violations (82 vulnerabilities), and semantic violations (45 vulnerabilities) in Table 2.

TABLE 2: CVE classification of device drivers in the Linux kernel.

Bug Class	Yes	Yes+P	No	Total
Safety Violation	**26**	**88**	**6**	**120**
- Buffer overflow	0	44	1	45
- Memory leak	6	7	0	13
- Integer arithmetic	0	6	0	6
- Use-after-free	15	7	0	22
- Null pointer dereference	0	17	0	17
- Memory disclosure	4	0	5	8
- Confused pointer error type	0	7	0	7
- Unsafe type cast	1	0	0	1
Protocol Violation	**41**	**25**	**6**	**72**
- Protocol steps	0	1	4	5
- Reference counting	0	24	0	24
- Race conditions	41	0	1	42
- Sleeping in Atmoic	0	0	1	1
Semantic Violation	**15**	**0**	**33**	**48**
- Security and permissions	0	0	7	7
- Logic errors	0	0	24	24
- Missing return value check	15	0	0	15
- Loop termination	0	0	2	2
Total	**82**	**113**	**45**	**240**

Safety vulnerabilities is the largest vulnerability class (49.5% of our data set). Among the safety violations only 21% of vulnerabilities can be eliminated by Rust alone. Other vulnerabilities require a specific programming technique and developer's involvement (74%), e.g., explicit unwrapping of return types, arithmetic checks for overflow and underflow, etc.). Together these two categories can eliminate 95% of safety vulnerabilities.

Protocol violations constitute 30% of our data set. Among protocol violations, 56% of vulnerabilities can be eliminated by Rust alone. In addition, 34% can be addressed through a specific programming technique and developer's involvement, e.g., explicit unwrapping of return types, arithmetic checks for overflow and underflow, etc. Together these two categories can eliminate 91% of safety vulnerabilities. Rust is able to address nearly all race conditions, and address reference counting vulnerabilities with programmer involvement.

Finally, semantic errors correspond to the 20% of all driver vulnerabilities. Among semantic vulnerabilities 31% can be addressed by Rust. While in general Rust cannot help with logical errors, a large class of semantic errors are missing return value checks (exactly 31%) and those are

TABLE 3: Lines of unsafe code for three different Rust drivers

Type	Lines of unsafe Rust							
	Binder		AGX		NVMe		Total	
	D	K	D	K	D	K	D	K
FFI func.	2	117	3	150	2	129	7	396
C callback	13	7	0	42	0	29	13	78
Unsafe func.	0	127	3	75	0	101	3	303
Marker trait	3	113	20	104	0	60	23	277
Union field	9	1	4	2	1	0	14	3
Foreign borrow	8	0	0	0	0	0	8	0
Pointer cast	16	167	24	112	3	131	43	410
Kernel crate func.	17	307	42	179	1	203	60	689

TABLE 4: Source lines of code (SLOC) in RFL driver branches

Project	Driver SLOC			Kernel Crate SLOC		
	code	unsafe	%	code	unsafe	%
Binder	4263	145	3.4	9116	884	9.7
AGX	15069	114	0.76	9428	915	9.71
NVMe	1758	27	1.54	7179	659	9.18

addressed by Rust through the `#[must_use]` types.

5. Use of Unsafe Rust

Previous studies concluded that the majority of bugs and vulnerabilities in Rust code are related to the use of unsafe subset of the language [61, 60, 21, 79]. Naturally, we expect that a large fraction of future vulnerabilities in RFL will be found in unsafe Rust. To gain a deeper understanding of how RFL uses unsafe Rust, we analyze kernel branches of the three most complete Rust drivers: Binder, AGX, and NVMe. We use Semgrep [71] on each branch to count and classify instances of unsafe code (Table 3), falling back to manual analysis for special cases. Specifically, for each class of unsafe code, we list the number of times we see it in the code of the driver ("D") and the kernel crate ("K"). Additionally, we used cargo-count to count the ratio of safe vs unsafe source code lines in Table 4.

All three drivers use unsafe code in both the driver and the kernel crates. Naturally, most of the unsafe code occurs in the kernel crate. Yet RFL falls short of the goal of avoiding unsafe in the driver itself (below we discuss common patterns of using unsafe code in both crates). While the percentage of the unsafe is less than 3.4% in all three drivers, arguably it's still high and a further effort is needed to reduce it possibly following the experience of research operating systems implemented in Rust [8, 47, 53]. The kernel crate has an even higher fraction of unsafe code (9.7%).

Moreover, in many cases, the logic of why a specific unsafe operation is safe is rather complex. RFL explicitly documents such safety contracts. Unfortunately, this is done informally (as a comment) instead of using a language formalism similar to specification languages in Rust-based automatic verification tools [45, 4].

Below we analyze common classes of unsafe Rust:

Kernel FFI functions RFL uses bindgen to automatically generate raw bindings to C functions in the kernel. The `extern "C"` bindings are unsafe. In most cases, a driver accesses this low-level unsafe interface through the safe abstractions provided by the kernel crate. Sometimes, however, the driver opts to invoke the raw FFI bindings directly if safe abstractions are not available.

```
#[no_mangle]
unsafe extern "C" fn rust_binder_compat_ioctl(
    file: *mut bindings::file,
    cmd: core::ffi::c_uint,
    arg: core::ffi::c_ulong,
) -> core::ffi::c_long {
    // SAFETY: We previously set `private_data` in `rust_binder_open`.
    let f = unsafe {
        Arc::<Process>::borrow((*file).private_data)
    };
    // SAFETY: The caller ensures that the file is valid.
    match Process::compat_ioctl(f, unsafe { File::from_ptr(file) }, cmd as _, arg as _) {
        Ok(ret) => ret.into(),
        Err(err) => err.to_errno().into(),
    }
}
```

Listing 13: Safe wrapper for the `compat_ioctl()` function

```
pub struct Pool<T> {
    ptr: *mut bindings::dma_pool,
    // ...
}

// SAFETY: A `Pool` is a reference (pointer) to an underlying C
// `struct dma_pool`. Operations on the underlying pool is protected
// by a spinlock.
unsafe impl<T> Send for Pool<T> {}
unsafe impl<T> Sync for Pool<T> {}
```

Listing 14: Explicit unsafe marker trait assignment in `dma::Pool`

Unsafe kernel crate functions Some functions of the kernel crate fail to implement a safe interface. For example, `List::remove()` removes an element from a linked list. It is unsafe because it requires the element being removed to be in the same list, an assumption that cannot be enforced through the type system.

C callback functions To implement a driver interface, every driver registers a collection of interface callback functions with the kernel. Callback functions interact directly with the kernel hence implementing a backward-compatible interface (in Rust they are marked as `extern "C"`).

Often callback functions exchange raw pointers with the kernel. Hence, Rust drivers mark callback functions as unsafe to prevent being invoked from safe Rust as invocation may allow safe code to pass and get raw pointers (Listing 13). While the entire interface function is marked as unsafe, the actual uses of unsafe in the functions are limited to type casts of raw pointers and converting reference-counted handles.

Marker traits Rust compiler tries to derive marker traits like `Send` and `Sync`. Derivation fails when the data structure contains pointers since it's not possible to guarantee thread safety when raw pointers are involved. If the developer determines that the tait invariants still hold, they must explicitly assign those marker traits using `unsafe` (Listing 14).

Tagged unions In Rust, `union` types are C-compatible unions that can only be accessed from unsafe code, since

```rust
pub(crate) fn as_ref(&mut self) -> BinderObjectRef<'_> {
    use BinderObjectRef::*;
    // SAFETY: The constructor ensures that all bytes of `self` are initialized, and all
    // variants of this union accept all initialized bit patterns.
    unsafe {
        match self.hdr.type_ {
            BINDER_TYPE_WEAK_BINDER | BINDER_TYPE_BINDER => Binder(&mut self.fbo),
            BINDER_TYPE_WEAK_HANDLE | BINDER_TYPE_HANDLE => Handle(&mut self.fbo),
            BINDER_TYPE_FD => Fd(&mut self.fdo),
            ...
        }
    }
}
```

Listing 15: Unsafe union access in Binder

```rust
#[no_mangle]
unsafe extern "C" fn rust_binder_new_device(
    name: *const core::ffi::c_char,
) -> *mut core::ffi::c_void {
    let name = unsafe { kernel::str::CStr::from_char_ptr(name) };
    match Context::new(name) {
        Ok(ctx) => Arc::into_foreign(ctx).cast_mut(),
        Err(_err) => core::ptr::null_mut(),
    }
}
```

Listing 16: A function producing a foreign-owned `Arc<T>`

there is no standard way to determine the variant stored in the union. RFL relies on an unsafe function to discriminate the type. An example from the Binder driver encapsulates union accesses in small unsafe blocks with textual explanations of the soundness (Listing 15).

Foreign `Arc<T>` borrow RFL often passes Rust-managed data structures as opaque pointers to C, including reference-counted containers like `Arc<T>`. In our classification, we treat function calls to perform this handoff separately from other unsafe calls into kernel crate since programmer mistakes can induce reference counting errors. RFL introduces the `ForeignOwnable` trait to model the transfer of ownership to C (Listing 16). The `into_foreign` method converts a `Arc<T>` handle to a raw pointer without decrementing the counter, essentially transferring the increment to C.

Device interface Unsafe operations are inevitable in low-level hardware interfaces of the device drivers that communicate with hardware, e.g., NVMe. For example, in the `dev_add()` function (Listing 17), 4096 bytes of DMA memory are allocated, and later retyped as a reference to `NvmeIdNs` with an unsafe type cast.

6. Use of Safe Programming Idioms

Mitigation of many vulnerabilities requires careful use of safe programming idioms which we classify as **Yes+P**. To understand how safe programming idioms are used in RFL, we analyze kernel branches of Binder, AGX, and NVMe for how they implement suggested programming techniques aimed to mitigate vulnerabilities in Rust code. Again we use Semgrep [71] on each branch to identify the use of each specific idiom along with the manual analysis for special cases.

```rust
fn dev_add(
    cap: u64,
    dev: &Arc<DeviceData>,
    pci_dev: &mut pci::Device,
    admin_queue: &Arc<nvme_queue::NvmeQueue<nvme_mq::AdminQueueOperations>>,
    mq: &mq::RequestQueue<nvme_mq::AdminQueueOperations>,
) -> Result {
    ...
    let id = dma::try_alloc_coherent::<u8>(pci_dev, 4096, false)?;
    ...
    let id_ns = unsafe { &*(id.first_ptr() as *const NvmeIdNs) };
```

Listing 17: Unsafe casting from raw DMA memory to NVMe namespaces id

```rust
impl ScatterGatherState {
    fn validate_parent_fixup(&self, ...) -> Result<ParentFixupInfo> {
        // ...
        let sg_idx = self.ancestors[ancestors_i];
        let sg_entry = match self.sg_entries.get(sg_idx) {
            Some(sg_entry) => sg_entry,
            None => {
                pr_err!(
                    "self.ancestors[{}] is {}, but self.sg_entries.len() is {}",
                    ancestors_i,
                    sg_idx,
                    self.sg_entries.len()
                );
                return Err(EINVAL);
            }
        };
        // ...
    }
}
```

Listing 18: Manual bounds check in scatter-gather list validation in Binder

Out-of-bound access For regular arrays and slices, accessing an out-of-bounds index via the index operator, i.e., `[]`, will trigger a panic due to an implicit bounds check. To avoid the panic, the driver developer needs to either perform a bounds check or use fallible accessors like `get` for array indices that may be out-of-bounds. In the case of user-provided indices, we observe explicit bounds checks being employed in the Rust Binder driver when validating a scatter-gather list (Listing 18).

Memory leak Section 4 suggests that drivers should carefully implement the `Drop` method to avoid memory leaks as well as support the correct implementation of the kernel de-registration protocol. While we observe that each of the three drivers follows the suggested discipline, the semantics of `Drop` implementation is often challenging. Memory leak often occurs when the programmer forgets to call free over the allocated memory. By implementing `Drop`, the programmer allows the Rust compiler to reason about the lifetime of a variable and correctly deallocate resources.

When writing a custom `Drop` implementation that interacts with externally-managed resources such as C objects and objects in hardware, care must be taken to correctly deallocate the resources given the current state. The AGX driver implements the `Mapping` abstraction that corresponds to a memory mapping in the AGX GPU which has its own MMU and TLB. Its `Drop` implementation carefully reasons

```rust
impl Drop for Mapping {
    fn drop(&mut self) {
        // prot::CACHE means "cache coherent" which means
            *uncached* here.
        if self.0.prot & prot::CACHE == 0 {
            self.remap_uncached_and_flush();
        }

        let mut owner = self.0.owner.lock();

        if owner
            .unmap_pages(self.iova(), UAT_PGSZ, self.size()
                >> UAT_PGBIT)
            .is_err()
        {
            // ...
        }

        if let Some(asid) = owner.slot() {
            // invalidate tlb
            mem::tlbi_range(asid as u8, self.iova(), self.
                size());
            mem::sync();
        }
    }
}
```

Listing 19: Drop implementation in the AGX GPU driver

```rust
fn handle_fault(&self) {
    ...
    let error = match self.get_fault_info() {
        Some(info) => workqueue::WorkError::Fault(info),
        None => workqueue::WorkError::Unknown,
    };
    self.mark_pending_events(None, error);
    self.recover();
}
```

Listing 20: Explicit result check in the AGX GPU fault hanlder

about the state of the mapping and invalidates the TLB entries (Listing 19). Arguably, complex `Drop` traits might be prone to potential vulnerabilities. Ideally, formal methods should be used for careful reasoning about the correctness of such code. We argue that low-burden verification of Rust code can become practical in the near future [17].

Option type (NULL pointer dereference, use-after free, pointer errors) Option types are used in RFL drivers to represent possible non-values, e.g., NULL references, uninitialized data structures, pointer errors, etc. For example, Listing 21 shows a common programming idiom used in Rust to handle option type. The driver uses `ok_or` to convert the optional value to `Result<T>` to signal the "Operation not supported by device" error if `id_info` is `None` (similar to `NULL` in C). To ease error handling, Rust provides the `?` operator to unwrap a `Result<T, E>` type. It's syntax sugar to return early with the error value in case of an error.

Alternatively, the driver can choose to handle each case of the `Option<T>` type explicitly with a `match` construct, like in the fault handler of the AGX GPU driver (Listing 20).

Integer overflow and underflow To mitigate potential panics resulting from integer overflow, the driver code should use explicit arithmetic checks such as `checked_add`. The three drivers only occasionally implement such checks, which means that panics due to overflows are likely possible. Listing 22 shows an example of `checked_add` on line 2 in android binder allocation code. The `read` method takes an `offset` and an argument of type `T` and adds the size of `T`

```rust
fn probe(
        pdev: &mut platform::Device,
        id_info: Option<&Self::IdInfo>,
    ) -> Result<Arc<DeviceData>> {
        ...
        let cfg = id_info.ok_or(ENODEV)?;
        ...
}
```

Listing 21: Implicit error check with `?` in AGX GPU driver

```rust
pub(crate) fn read<T>(&self, offset: usize) -> Result<T
    > {
    if offset.checked_add(size_of::<T>()).ok_or(EINVAL)?
        > self.limit {
        return Err(EINVAL);
    }
    self.alloc.read(offset)
}
```

Listing 22: Use of checked arithmetics in Binder

with the offset. It then converts the `Option<T>` type into `Result<T, E>` to signal the "Invalid Argument" error when an overflow occurs.

Reference counting Rust supports automatic reference counting and resource cleanup. With this extra feature comes extra complexity at the kernel-driver boundary. A developer must take great care when implementing reference-counted abstractions that are shared with C. For example, in the Rust NVMe driver, block requests are modeled as `Request<T>` types which can be moved to and from C. The main complexity in its `AlwaysRefCounted` implementation lies in its own reference counter which has special semantics depending on whether C or Rust owns the object. When Rust owns the object, the atomic reference counter must be strictly greater than zero.

7. Conclusions

The development of device drivers in a safe programming language could potentially change the balance of security in the kernel. Our work analyzes the impact of safety on the security of the driver subsystem if re-implemented in Rust. We agree that the promise of RFL is attractive: Rust can eliminate a significant fraction of vulnerabilities currently present in unsafe device drivers. Nevertheless, most vulnerabilities require careful programming discipline to be fully mitigated by Rust. We hope that our work can improve understanding of potential flaws and vulnerabilities in RFL and, hopefully, result in a more secure kernel.

Acknowledgements

We would like to thank ACSAC'24 reviewers and an anonymous shepherd for numerous insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 2313412, 2341138, and 2239615.

References

[1] A minimal Linux kernel module written in Rust. https://github.com/tsgates/rust.ko.

[2] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, R Wisniewski, and Jimi Xenidis. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, 2002.

[3] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe Rust? *Proc. ACM Program. Lang.*, (OOPSLA), November 2020.

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, October 10, 2019.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (SOSP), 2009.

[6] D. W. Boettner and M. T. Alexander. The michigan terminal system. *Proceedings of the IEEE*, (6):912–918, June 1975. ISSN: 0018-9219.

[7] Bomberger, A.C. and Frantz, A.P. and Frantz, W.S. and Hardy, A.C. and Hardy, N. and Landau, C.R. and Shapiro, J.S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[8] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: An experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[9] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *2010 USENIX Annual Technical Conference (USENIX ATC '10)*, 2010.

[10] Bromium. Bromium micro-virtualization, 2010. http : / / www . bromium . com / misc / BromiumMicrovirtualization.pdf.

[11] Browser rewrite in rust - Mozilla. https : / / hacks . mozilla . org / 2019 / 02 / rewriting - a - browser - component-in-rust/.

[12] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, (4):412–447, November 1997. ISSN: 0734-2071.

[13] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the Linux kernel without system-call descriptions. In *Proceedings of the 30th Network and Distributed System Security Symposium* (NDSS), 2023.

[14] 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* (HOTOS), Providence, RI, USA, 2023.

[15] 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), 2009.

[16] 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 Second Asia-Pacific Workshop on Systems (APSys '11)*, 2011.

[17] Xiangdong Chien, Zhaofeng Li, Jerry Zhang, and Anton Burtsev. Veld: Verified linux drivers. In *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification* (KISV '24), 2024.

[18] 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 (KISV '23), 2023.

[19] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (SOSP), October 2001.

[20] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[21] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. Is unsafe an achilles' heel? a comprehensive study of safety requirements in unsafe Rust programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (ICSE), Lisbon, Portugal, 2024.

[22] DDEKit and DDE for Linux. http://os.inf.tu-dresden.de/ddekit/.

[23] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, 2004.

[24] 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), 2006.

[25] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. Is Rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE), Seoul, South Korea, 2020.

[26] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. Technical report TUD-FI06-07, 2006.

[27] 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)*, 1997.

[28] Alessandro Forin, David Golub, and Brian N Bershad. An I/O system for Mach 3.0. Carnegie-Mellon University. Department of Computer Science, 1991.

[29] Mozilla Foundation. The Rust programming language. https://doc.rust-lang.org/book/.

[30] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[31] 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*, number 1. ACM, 2008.

[32] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (SOSP), 2003.

[33] 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* (EW), Kolding, Denmark, 2000.

[34] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, 1996.

[35] David B Golub, Guy G Sotomayor, and Freeman L Rawson III. An architecture for device drivers executing as user-level tasks. In *USENIX MACH III Symposium*, 1993.

[36] Google. Fuchsia project. https://fuchsia.dev/fuchsia-src/getting_started.md.

[37] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. SyzDescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy* (S&P), 2023.

[38] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical report, TU Dresden, Dresden, Germany, 2003.

[39] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, (4):3–11, 2007.

[40] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, (3):80–89, 2006.

[41] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, 2004.

[42] Jia-Ju Bai and Yu-Ping Wang and Julia Lawall and Shi-Min Hu. DSAC: Effective static analysis of Sleep-in-Atomic-Context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference* (ATC), July 2018.

[43] Antti Kantee. *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. PhD thesis, 2012.

[44] The Rust Programming Language. rust-bindgen. https://github.com/rust-lang/rust-bindgen, 2024.

[45] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, (OOPSLA1):85:286–85:315, April 2023.

[46] 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), 2004.

[47] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[48] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *SIGOPS Oper. Syst. Rev.*, (2):51–62, April 1991. ISSN: 0163-5980.

[49] Linux kernel vulnerabilities. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[50] 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*, 2022.

[51] 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), 2011.

[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)*, July 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)*, November 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)*, 2020.

[55] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (SOSP), 2013.

[56] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), Newport Beach, California, USA, 2011.

[57] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. POTUS: Probing off-the-shelf USB drivers with symbolic fault injection. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies* (WOOT), 2017.

[58] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *Proceedings of the 29th USENIX Security Symposium* (USENIX Security), 2020.

[59] Octavian Purdila. Linux kernel library. https://lwn.net/Articles/662953/.

[60] B. Qin, Y. Chen, H. Liu, H. Zhang, Q. Wen, L. Song, and Y. Zhang. Understanding and detecting real-world safety issues in Rust. *IEEE Transactions on Software Engineering*, (01):1–19, March 5555. ISSN: 1939-3520.

[61] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), London, UK, 2020.

[62] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), 2012.

[63] Rust Binder driver. https://github.com/Darksonn/linux/tree/rust-binder-rfc.

[64] Rust E1000 driver. https://github.com/fujita/rust-e1000.

[65] Rust for curl. https://timmmm.github.io/curl-vulnerabilities-rust/.

[66] Rust GPU driver. https://github.com/AsahiLinux/linux/tree/asahi.

[67] Rust Nvme driver. https://github.com/metaspace/linux/tree/rnvme-v6.9-rc3.

[68] Rust NVMe driver: potential sleep-in-atomic-context.

[69] Rutkowska, J. and Wojtczuk, R. Qubes OS architecture. *Invisible Things Lab Tech Rep*, 2010.

[70] Safer System Language - Microsoft. https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/.

[71] SemGrep. https://github.com/semgrep/semgrep.

[72] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *Proceedings of the 31st USENIX Security Symposium* (USENIX Security), 2022.

[73] 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*, 2002.

[74] Kaku Takeuchi et al. An interaction-based language and its typing system. In *PARLE*, 1994.

[75] Hajime Tazaki. An introduction of library operating system for Linux (LibOS). https://lwn.net/Articles/637658/.

[76] Kevin Thomas Van Maren. THE FLUKE DEVICE DRIVER FRAMEWORK, 1999.

[77] 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), 2008.

[78] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. DEVFUZZ: Automatic device model-guided device driver fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[79] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. Memory-safety challenge considered solved? an in-depth study with all Rust CVEs. *ACM Trans. Softw. Eng. Methodol.*, (1), September 2021. ISSN: 1049-331X.

[80] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Proceedings of the 29th Network and Distributed System Security Symposium* (NDSS), April 2022.