An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise

Hongyu Li, Beijing University of Posts and Telecommunications; Liwei Guo, University of Electronic Science and Technology of China; Yexuan Yang, Shangguang Wang, and Mengwei Xu, Beijing University of Posts and Telecommunications

https://www.usenix.org/conference/atc24/presentation/li-hongyu

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.
July 10–12, 2024 • Santa Clara, CA, USA
978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by
An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise

Hongyu Li¹ *, Liwei Guo² *, Yexuan Yang¹, Shangguang Wang¹ and Mengwei Xu¹

¹Beijing University of Posts and Telecommunications
²University of Electronic Science and Technology of China

Abstract

Developed for over 30 years, Linux has already become the computing foundation for today’s digital world; from gigantic, complex mainframes (e.g., supercomputers) to cheap, wimpy embedded devices (e.g., IoTs), countless applications are built on top of it. Yet, such an infrastructure has been plagued by numerous memory and concurrency bugs since the day it was born, due to many rogue memory operations are permitted by C language. A recent project Rust-for-Linux (RFL) has the potential to address Linux’s safety concerns once and for all – by embracing Rust’s static ownership and type checkers into the kernel code, the kernel may finally be free from memory and concurrency bugs without hurting its performance. While it has been gradually matured and even merged into Linux mainline, however, RFL is rarely studied and still remains unclear whether it has indeed reconciled the safety and performance dilemma for the kernel.

To this end, we conduct the first empirical study on RFL to understand its status quo and benefits, especially on how Rust fuses with Linux and whether the fusion assures driver safety without overhead. We collect and analyze 6 key RFL drivers, which involve hundreds of issues and PRs, thousands of Github commits and mail exchanges of the Linux mailing list, as well as over 12K discussions on Zulip. We have found while Rust mitigates kernel vulnerabilities, it is beyond Rust’s capability to fully eliminate them; what is more, if not handled properly, its safety assurance even costs the developers dearly in terms of both runtime overhead and development efforts.

1 Introduction

As the de facto foundation for today’s computing infrastructure, Linux never ceases to eliminate memory and concurrency bugs [55, 61, 80, 109], which have been plaguing systems software for years. Yet, bugs keep emerging [15, 23–25] despite years of security hardening and engineering efforts from the Linux community [1, 19]. One root cause is the C language allows unfettered access to memory objects, which the Linux kernel exploits wild typecasting, raw pointer dereferencing, etc., to construct complex abstraction layers and generic frameworks (e.g. device drivers [60]) for modularity and performance.

How to ensure memory safety with little or no performance degradation? Rust seems to be a promising solution, which may finally resolve the aforementioned problems [13]. As an emerging, statically and strongly-typed systems programming language, Rust claims to deliver both safety and performance without runtime overhead [51]. Backing its claim is the ownership mechanism [48] for eliminating memory and concurrency bugs, underpinned by three key rules: 1) each memory location can only be owned exclusively by one variable at a time; 2) the ownership of variables may be permanently moved (i.e. transferred to other variables) or temporarily borrowed through references within a thread and across threads, using the Send and Sync traits respectively; 3) once the owner goes out of scope, its owning variable is dropped with its memory being freed. By relying on extensive static checks to enforce the above rules at compile time, Rust eliminates a heavyweight and costly memory checker [84, 90].

Figure 1: A few remarkable milestones of RFL.
as well as a garbage collector [75, 85], eschewing from being interrupted and having unpredictable delays at run time.

**Rust for Linux** The above intriguing properties of Rust led to the advent of Rust-for-Linux (*RFL*), which begins in 2013 as a hobbyist project [3] shown in Figure 1. As the first attempt to bring Rust into Linux, the project built a Rust object file against kernel headers and invoked one Rust function from the file in a loadable kernel module. The Rust function simply printed out “Hello from Rust!++”, yet was the first cry of Rust in the Linux kernel space. At the time, Rust was only Linux’s nice peripheral, whose main use case was to compose safety-critical, brief, and standalone snippets for kernel modules (i.e. still written in C language) to invoke. The situation started to shift in 2019. In the year, a proposal to write kernel modules fully in Rust emerged [8, 11] to lead Rust further into Linux. To achieve its goal, the proposal took a bold move by directly adding a thin Rust wrapping layer to kernel interfaces and data structures in upstream Linux. The proposal is catalyzing: a year later, the idea of further fusing Rust into Linux kernel tree debuted at the Linux Plumbers Conference (LPC) in 2020 and received widespread support [10], which incubated the first *RFL* RFC in 2021 [13]. Since then, *RFL* has attracted significant attention. With developers’ fond hope of revolutionizing kernel safety and with the thrust from Rust’s community-based collaboration model, *RFL* quickly gained momentum: all *RFL* code is managed by a Git repo, which runs a continuous integration (CI) pipeline as a gatekeeper for checking patch quality and correctness, boosting development efficiency; discussions happen more vividly and lively on an online forum called Zulip instead of the traditional mailing list. Hence, only a year after the RFC, *RFL* is officially merged into upstream Linux v6.1 as an experimental kernel feature [20]. When *RFL* gradually gets speed in the journey of robusting Linux, there emerge numerous attempts to use *RFL* to write drivers in various areas, such as network [62], block device [68, 69], file system [32, 36], android [59], and GPU [67]. Among them, one network driver [40] first makes it into the Linux mainline in v6.8 after 11 rounds’ co-review of the *RFL* and network community. This means *RFL* can receive feedback from users, which is a sign of *RFL* stepping into the real use cases from the experimental states. Despite still being in an early stage, *RFL* has become one of the most active kernel subsystems [47], on par with ebpf [63], and io_uring [65].

**Motivations** While Rust has been well understood [71, 103, 110] as a programming language, its synergy between Linux, i.e., *RFL*, is rarely studied. Understanding how Rust is fused into Linux is critical, as it provides practical guidance for implementing safe and efficient drivers under active development, as well as valuable lessons for incorporating Rust into more kernel subsystems in the near future.

Not less importantly, *RFL* also provides a unique angle for us to look into how a new programming language blends into a giant, old-school codebase which has been already shipped on billions of computing devices. This process is profound and complex, involving not only massaging self-contained kernel libraries into Rust libraries (i.e. crates), and also adaptations in the language syntax/semantics level, e.g., adapting C paradigms such as generative and functional-like macros to Rust semantics and interfaces. It also concerns judicious design decisions and engineering practices to fit strict safety rules into the kernel execution environment which needs to be more relaxed in executing memory ops. Moreover, together with the new language, it introduces new collaboration models and development tools, which have slowly but surely shifted the development process of Linux. This process, as we will show, is a mixture of success, dissatisfaction, and compromise.

With over 20K lines of *RFL* code being officially merged into Linux mainline since 2022, we believe it is good timing to reflect on this process and draw necessary insights from it. To this end, we conduct the first comprehensive study examining the ongoing fusion between Rust and the Linux kernel. We thoroughly look into the *RFL* project and have collected 6 Rust drivers, 269 issues, 763 PRs, and 1540 commits in the *RFL* GitHub, 3611 mails in the *RFL* mailing list, and 12501 discussions in the *RFL* Zulip forum.

In this paper, we answer the following three key research questions (RQs):

**RQ1: what is the status quo of *RFL*?** Based on the collected data, our study first presents a comprehensive analysis of the current landscape of *RFL*. We further dive into the individual drivers for gauging the implementation gap between Rust and the Linux kernel. Our key observations are: 1) while *RFL*’s toolchain (e.g., Kbuild) has mostly matured, it lacks major drivers and file systems, which is bottlenecked by a slow code review process; 2) although Rust Traits lifts the significant burden of manual kernel security audit, it is no silver bullet in affirming full memory safety. Additionally, the mismatch between Rust and the kernel on memory operations results in complicated workarounds, which leads to both runtime and development overhead.

**RQ2: does *RFL* live up to the hype?** We re-evaluate the three critical goals set by the kernel community when initiating *RFL*, which aims to make the kernel 1) more secure at 2) little overhead and enable 3) easier development. To do so, we analyze the collected bug reports, PRs, and patches and run 5 *RFL* kernel drivers on real hardware. Contrary to the common belief, our study shows Rust only makes the kernel more “securable” but not fully secure due to unsafe usage is inevitable in driver development. Moreover, as a side effect of generic Traits and smart pointers, Rust drivers incur a large number of icache misses and under-performs C drivers significantly in some cases. Delightedly, we have also found Rust greatly improves kernel code quality and has attracted new developers who are not even familiar with C into kernel programming.

**RQ3: what are the lessons learned from *RFL*?** Based
on the insights from previous two RQs, we provide practical guidelines in implementing and using RFL for both developers and the community. For developers, we suggest they take Rust safety assurance with a grain of salt as Rust safety rules do not detect semantic bugs and developers often opt for unsafe as the final resort. For the community which seeks to expand the RFL scope into future kernel subsystems, we model the benefits of rustifying a kernel subsystem as the tradeoffs between the security gain (i.e. bugs Rust may eliminate) and engineering efforts. With the model, we show that there are 7 out of 79 subsystems and their 25 related drivers have greater benefits, including ext4 and linux-block. Prioritizing RFL development on them likely has a higher value.

Contributions Compared with prior works studying Rust safety and performance [81, 97], we are the first to present a comprehensive study on Rust synergy with Linux. Overall, this paper makes the following contributions:

- We have collected a dataset of RFL development, constituting 1540 commits, 269 issues, 763 PRs and 3611 mail exchanges, which we will publicize upon publication.
- Based on the dataset, we developed a toolset for automatically processing APIs, analyzing their usage statistics, and reporting the results by the subsystem view.
- We present a detailed analysis of the current RFL development status, disclose the key difficulties in incorporating Rust into the kernel execution environment, and insightful advice on future RFL driver development.

The dataset and code is available at https://github.com/Richardhongyu/rfl_empirical_tools. We plan to continuously monitor RFL evolvement and update the repository.

2 A Primer on RFL in Linux

Rust safety model Rust is a statically and strongly-typed language. In short, its safety model regulates the accesses to memory locations: at one given time, only one variable may write to a memory location but many may read from it. It instantiates the following strict rules:

- Ownership and lifetime. It mandates each memory location (i.e., value) can have a single owner at a time, similar to affine Sync traits, respectively.

- Move and Borrow. Move transfers the ownership of a variable when it is consumed by another variable or passed to a function as an argument. Ownership can be Borrowed without being transferred, by generating the reference of a memory location. Once moved or borrowed, the memory location can no longer be modified by its original owner. Ownership can also be Moved and Borrowed across threads via Send and Sync traits, respectively.

The unsafe keyword While the above rules ensure memory and thread safety by eliminating sharing and aliasing, they critically restrict the expressiveness of the language. For instance, they prohibit complex data structures such as a doubly-linked list, where each node is simultaneously referenced (i.e., owned) by both its predecessor and successor, violating the exclusivity mandated by the ownership mechanism.

To overcome this, Rust introduces the unsafe keyword as an escape hatch to bypass static compiler checks on ownership of the variables. With the unsafe keyword, programmers are allowed to manipulate memory objects using a wide range of operations, such as raw pointer de-reference, calling functions via foreign function interface (FFI), and even inline assembly. The keyword serves as an agreement with the programmers that Rust compiler trusts the programmers for the safety of their unsafe blocks. It is proven by properly wrapping unsafe code under the safe APIs, it is possible for the whole program to still enjoy full safety guarantee of Rust [86].

Binding Rust with Linux To implant Rust-written device drivers into the C-written Linux kernel, RFL first preprocesses kernel APIs on which Rust drivers depend, e.g., kmalloc. Using rust-bindgen, it generates Rust APIs corresponding to the kernel APIs; the newly generated kernel APIs conform to Rust calling convention and form kernel crates (i.e., Rust libraries), which RFL may directly invoke via FFIs. Since the APIs eventually land in kernel address space (i.e., unchecked by Rust compiler) and is unsafe, RFL wraps them with a safe abstraction layer; by design, Rust drivers shall only invoke safe APIs exported by the layer.

Figure 2 shows an actual example of integrating a Rust char device driver into Linux. Later sections uncover more complex drivers such as NVMe, NIC, and GPU drivers. In this case, rust-bindgen looks at the header file cdev.h included by the driver and generates FFIs in three crates as shown in Figure 2 (a). The developer then manually constructs the kernel crate as the safe abstraction layer for wrapping unsafe APIs, e.g., alloc wraps around the unsafe invocation to bindings::cdev_alloc as in Figure 2 (b). Notably, RFL uses comments as the contract between developers to reason about the safety of unsafe blocks, which specify preconditions and usage of the input variables of underlying FFIs. At last, the driver follows Rust programming conventions and invokes the APIs in the safe abstraction layer to access the kernel infrastructure (Figure 2c).

RFL goals Table 1 summarizes the goals that RFL has declared in the initial RFC [13] to the mailing list and official presentations [14]. In general, RFL attempts to improve the safety of Linux drivers with zero overhead, and brings the modern development experience and efficient development tools into the kernel to attract more developers.
3 RQ1: What is the status quo of RFL

In this section, we first present an analysis on current development status of RFL based on collected commits, issues, and mail exchanges. Next, we reveal the tension between Rust and kernel programming by diving into the construction of safe abstraction layer and Rust drivers.

3.1 RFL development status

Our key observation is: although RFL is still young, the infrastructure (e.g., irq, mm, sched) has matured; while individual drivers and file systems (i.e., relatively independent subsystems) are currently lacking and will be the next focus, the shortage of qualified reviewers bottlenecks RFL development.

3.1.1 Methodology

We collect RFL code from PRs/commits on GitHub and patches on Linux mailing list, as RFL leverages GitHub as its code repository before submitting to the kernel community. Depending on the RFL community collaboration mode [37], we further categorize the RFL code into three stages: 1) pending: the commit is still in PR of RFL repo, pending for the first round of review, 2) staged: it has passed PR and been moved to the mailing list for formal review by the maintainer of the kernel subsystem, 3) merged: the commit has been formally merged into the upstream kernel, i.e Linux mainline or Linux-next repo.

In total, we collected 160+ merged RFL commits (19K LoC1), 1300+ staged commits (112K LoC) and 500+ pending commits (186K LoC). We further extract kernel data structures and functions from them to gain insights into their composition and distribution along RFL development timeline. To do so, we develop a tool which automatically parses the code with regular expressions and reports the results by respective subsystems and aforementioned categories.

3.1.2 Results

(1) Development progress. The goal is to gauge the gap between the current RFL codebase versus a completely Rustified Linux kernel. Overall, RFL is still at a very early stage in blending with the Linux kernel: in terms of LoC, the merged code (7.1%) only constitutes 0.125% of the kernel code, while the rest (92.9%) is still pending review or is staged for merging. We further break down the merged code by their respective kernel subsystems to understand individual status and show the results in Figure 3.

Insight 1: drivers, netdev, and file systems are the long tail of RFL code.

We have observed a clear long tail: most code resides in scheduling, memory management, and IRQ infrastructure. By contrast, drivers, file systems, netdev, and security subsystems

---

1All LoC results are reported by cloc [78], if not specified otherwise
which account for most kernel code (i.e., 78% in Linux v6.2) only have received little RFL code, constituting the “tail”.

The results are sensible. As scheduling, memory management, and IRQ subsystems are most commonly used by all drivers, optimizing RFL development for them has a high value and priority. In comparison, drivers and file systems have more specific use cases (e.g., for a particular model of a device), which require more programming and reviewing effort. For instance, reviewers from netdev communities spend 6 months on 11 versions of draft patches, before settling one the final merged network PHY driver [46].

(2) Patch distribution. To study how individual RFL components develop, we categorize the code into three types depending on their use cases, i.e. for building safe abstraction, Kbuild system, and Rust compiler. We show the results in Figure 4 and conclude the following insight:

Insight 2: RFL infrastructure has matured, with safe abstraction and drivers being the next focus.

We base the insight on two key pieces of evidence: 1) as time passes, Kbuild undergoes a clear recession in its portion of the RFL cake, indicating the foundation of RFL has been laid; 2) in the meantime, abstraction takes up more portion, e.g., from 20% to 60% in 18 months. Interestingly, a surge in the number of Rust commits appeared half a year after RFL started, as seen by the 23/4 timestamp of Figure 4. It belongs to a patch that directly modifies the RFL lib code for supporting a safe implementation of Rust initializer for pinned objects; prior to the fix, early RFL has been using unsafe initializer in gpio_pl061, bcm2835_rng drivers.

3) The trend. To understand how RFL development progresses with time, we project the commits and the email exchanges onto the timeline and highlight the reviewing time length of the PRs, as shown in Figure 5.

Insight 3: RFL is bottlenecked by code review but not by code development.

From the slope changes of the left figure, we are witnessing the committed/staged RFL code start to plateau after the initial steep when RFL first started. Yet, the number of email exchanges shows RFL is an increasingly active community. Besides, the PR reviewing becomes significantly slower as time goes by observed from the right figure. For instance, the PRs between Jan. 2023 and Jul. 2023 take 280 hours to be reviewed on average, which is 200× as long as 3 years ago. This suggests the speed of producing RFL code is much faster than that of consumption, i.e., reviewing and eventually merging RFL code into the upstream kernel. This can also be confirmed by the huge imbalance between merged code versus the rest of the code, where the latter is 13 × larger than the former and represents a huge bulk of the area.

The reasons are multifold. First, there is clearly a lack of qualified reviewers who must be familiar with both Rust and kernel programming. This is backed up by many posts we found from the mailing list [27, 29]. Second, there is a mismatch of collaboration conventions between the RFL and Linux subsystem communities [28], including response time and review cycle. Such a mismatch slows down the development speed. Third, there is a deadlock of RFL development: the subsystem communities are unwilling to review the...
patches about safe abstractions without real Rust drivers as motivating examples; yet without such abstractions, the RFL community is not able to construct the drivers in Rust [39]. Fortunately, such deadlock has drawn attention from the communities, and early solution has been proposed [34].

An inspiring observation is that RFL is gradually embraced by the kernel community, seeing the increasing engagement of traditional kernel developers. For example, the recently developed NVME, NULL block, V4L2, and e1000 drivers using RFL are all driven by the Linux community.

### 3.2 Rustify Linux with safe abstraction

Safe abstraction is the key ingredient towards rustifying Linux kernel and among the largest portions of RFL code (§ 3.1). As the name itself implies, the layer extends a C kernel safely into Rust drivers: it abstracts kernel data structures and interfaces, so that upon invocation they may still ensure memory and thread safety.

**Challenge: taming kernel programming conventions** To construct the safe abstraction layer, RFL first translates kernel data structures and interfaces into Rust style; it then wraps them with Rust interfaces and exports them to drivers (§ 2). While the process seems straightforward, RFL needs to address a unique challenge – how to tame kernel programming paradigms with Rust safety rules? For instance, the kernel extensively uses typecasting, pointer arithmetic, bit operations, etc., many of which contradict Rust philosophy. As we will show, to construct the safe abstraction layer, RFL deploys various workarounds, systematically manages kernel states in the Rust style, and even evolves the language itself to cater to the kernel.

**Converting kernel data structures** RFL leverages bindgen to automatically generate Rust bindings of C kernel struct prior to use. The generation is rule-based and syntax-directed, which translates the C types and symbols into their Rust counterparts. The translation is mechanical, following the rules in Table 2. For instance, uint32_t of C translates into c_uint in Rust namespace core::ffi, which aliases to the Rust primitive u32. However, not every C type translates into a corresponding Rust primitive. We have found such incompatibility exists especially in language features which kernel exploits for manually consolidating memory layout. We detail them as follows.

<table>
<thead>
<tr>
<th>Type</th>
<th>C</th>
<th>Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive types</td>
<td>foo</td>
<td>core::ffi::foo</td>
</tr>
<tr>
<td>Typed pointers</td>
<td>foo *</td>
<td>*mut foo</td>
</tr>
<tr>
<td>Attributes</td>
<td>aligned</td>
<td>#repr(c)</td>
</tr>
<tr>
<td></td>
<td>unused</td>
<td>ignored</td>
</tr>
<tr>
<td></td>
<td>weak</td>
<td>ignored</td>
</tr>
<tr>
<td></td>
<td>randomize</td>
<td>ignored</td>
</tr>
<tr>
<td>Function pointer</td>
<td>fn</td>
<td>option&lt;fn&gt;</td>
</tr>
</tbody>
</table>

Table 2: The translation rules from C to Rust in the rust-bindgen.

(1) **Emulated bitfields and unions.** The kernel extensively uses bitfields and unions for improving memory efficiency, e.g., e1000 driver uses a single word to store 4 flags to indicate link states. Bit operations on struct members contradict Rust memory safety principle and hence do not have native Rust support. As a workaround, RFL emulates bitfields with a byte array, which implements bit operations as accessors to the array. Although Rust has a union primitive, it cannot provide ABI compatibility with C union. Thus RFL implements a struct called __BindgenUnionField with the same memory layout as the C interface. Both workarounds are based on the transmute operation, which reinterprets memory at run time, hence are implemented in unsafe blocks. The major overhead of the emulation code is the increase of binary size, which we will discuss in Section 4.2.

(2) **Incomplete attribute support.** The kernel often relies on packed and aligned attributes for better locality and memory efficiency, e.g., task_struct groups most frequently accessed scheduling data in one cache line. Despite the attributes are supported by Rust repr(C), Rust may still mishandle them and cause bindgen to generate the wrong code [7, 9]. For attributes less commonly used, RFL does not support them, e.g., BTF tags [21]. Notably, RFL ignores randomize_layout attribute, which kernel utilizes to mitigate memory bugs [58]. This is reasonable because Rust has already mitigated such vulnerabilities through ownership and boundary checks.

Despite the generated bindings having the identical data layout as their C counterpart, the safe abstraction layer still cannot directly expose them to drivers. This is because the bindings involve numerous raw pointers (i.e., *mut), which are prevalent in kernel but are unsafe to use in Rust. To safely use them, Rust needs to manually reason about the pointer validity (i.e., not via borrow checker at compile time) and specify their ownership. To this end, RFL uses helper types to embed generated kernel data structure bindings and bakes Rust flavors into them.

**Insight 5:** RFL uses helper types to delegate management of kernel data to Rust while leaving the operation to kernel itself.

Specifically, RFL leverage two key Rust features to manage kernel data structures.
(1) **RFL** uses Type and Deref coercion to bound kernel pointer de-referencing and casting. For the embedded kernel data structures, **RFL** overrides memory accesses to them by combining Type and Deref coercion. By doing so, **RFL** stipulates how Rust interprets the kernel pointer and prevents direct memory operations such as de-referencing on raw pointers, and transmuting the memory objects. As an example, device struct often contains `void *` to point to device-specific data, which kernel typecast at run time. For all such structs, **RFL** implements deref traits on them, whose function body coerces the de-referencing to result in a correct type.

(2) Automate kernel struct life cycle management. **RFL** introduces three new low-level types and attach them to helper types for mechanizing the management of kernel struct, i.e., `ScopeGuard`, `ARef`, `opaque`. Essentially, the low-level types exploit Rust support on executing custom stub functions upon entering/exiting specific scopes e.g., `ScopeGuard` frees allocated resources of a `Task` by executing its drop traits when the Task’s life cycle ends; `ARef` automatically increments/decrements the refcount of the helper type whenever it is referenced/freed. Prior to **RFL**, kernel developers need to manually manage the resources and states, e.g., explicitly call `get/put_task` to increment/decrement refcount of `struct task`. With Rust, **RFL** solidifies the under-documented kernel programming convention and provides a safe management foundation for the rust driver.

With Rust, **RFL** separates the management from operation of a kernel object. An example is kernel-locking primitives. **RFL** re-writes the backend for mutex and spinlocks using the helper types. For management, the new backend leverages Deref coercion to safely de-reference the data protected by the lock and use `ScopeGuard` low-level type for runtime memory reclamation. For operation, **RFL** still invokes kernel locking/unlocking methods.

**Incorporating kernel functions** Following similar translation rules in converting the kernel data structures (Table 2), **RFL** generates FFI bindings of kernel functions. Then, **RFL** extensively leverages Rust traits to massage Rust features into the helper types. For management, the new backend leverages Deref coercion to safely de-reference the data protected by the lock and use `ScopeGuard` low-level type for runtime memory reclamation. For operation, **RFL** still invokes kernel locking/unlocking methods.

(3) Wrapping inlined functions and macros. They provide important and handy utilities to kernel drivers, e.g., `for_each_online_cpu` to enumerate available CPUs. **RFL** wraps static inline functions with non-inlined Rust functions. This is because current Rust lacks a convenient mechanism for inlining FFI’s of C functions; while it is still possible to inline them, it takes lots of efforts and hence is not encouraged by the community [44]. For function-like macros, **RFL** prefers wrapping them with helper functions instead of rewriting them with Rust macros. The main reason is **RFL** inclines not to maintain two sets of kernel interfaces known to be unstable [82].

### 3.3 Rustify device drivers

As the programming paradigm has shifted from C to Rust, the reasoning of data layout (as in C) has also shifted to that of driver data ownership. In this section, we put ourselves in the shoes of a developer and describe how the shift has impacted driver development.

**First, device probing.** The developer implements the device probing callback, which allocates kernel resource for device data and registers the IRQ handler if needed. Compared with driver development in C, the key difference is the developer must annotate the device data with ownership types, i.e., how the data might be used by what entity. For instance, she marks the data with `Arc` if it might be shared among threads (e.g., locks), and `Pin` if she wants to the data to be unmovable so it can be shared with the C side. To make matters more complicated, the annotations may be nested and hard to comprehend. As an example, the rx ring buffer of e1000 NIC driver contains an array of rx buffer descriptors and is protected by a spinlock; its type is specified as `Pin<Box<SpinLock<Box<RxDesc>>>> [62]` and each nested type needs to invoke its own initializer, e.g., `Pin::from and Box::try_new`.

**Second, implementing driver function.** The developer proceeds to implement driver functions required by the driver framework, e.g., to operate, all NIC drivers must implement `ndo_open`, `ndo_start_xmit` callbacks of `net_device_ops`. This step resembles most of driver development in C language, where the developer encodes core device logic to drive the device towards desired states, as specified by the hardware manual. Yet, Rust poses following implementation challenges for the developer to maneuver safety rules in kernel space:

(1) Implementing dynamically-sized arrays is non-trivial. The C drivers often uses pointers to implement adjustable arrays for hosting dynamic kernel objects, e.g., pages. In Rust drivers, the developer must introduce multiple extra layers and implement `dyn_num` trait on them. An actual example is shown in Figure 6 from RROS [96]. As can be seen, to realize the same
Figure 6: An example showing the inflexibility of writing dynamically-sized arrays in RFL drivers.

```rust
// In C
struct elements { int len; void* inner; }
struct factory {
    struct elements inner;
}

// In Rust with Fixed N
struct elements<const N: usize> {
    inner: [foo; N],
}
struct factory { inner: elements<256/> }

// In Rust with Dynamic change N
struct thread/proxy<const N: usize>{
    thread/proxy_elements: elements< N >,
}
impl dyn_num for thread<256>/proxy<8> {}
trait dyn_num { // fn use_elements(&self); }
struct factory { inner: &static dyn dyn_n }
```

4 RQ2: Does RFL live up to the hype?

In this section, we reflect on the initial goals set by the kernel community when starting RFL (Table 1) and focus on three questions:

1. Does Rust help Linux become safer? (§ 4.1)
2. Does Rust bring additional overhead? (§ 4.2)
3. How does Rust improve Linux development? (§ 4.3)

Table 3: The unsafe usage in RFL drivers. The second and third columns show the number of unsafe usage due to 1) Driver logic is too complex for Rust safety rules, and 2) Safety abstractions are currently missing in RFL, respectively.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Number of Unsafe usage</th>
<th>Driver logic</th>
<th>Safety abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU [67]</td>
<td>107</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>NVME [69]</td>
<td>44</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>Null block</td>
<td>44</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>E1000 [62]</td>
<td>4</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Binder [59]</td>
<td>45</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Gpio_pl061</td>
<td>0</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Semaphore</td>
<td>0</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

4.1 RFL makes Linux more “securable”

Methodology We focus on the bug reports of Rust and the use of unsafe code blocks in RFL and drivers. Our rationale is RFL safety hinges on the safety assurance of Rust lang, which concretely relies on the elimination of all unsafe blocks in drivers and the safe abstraction APIs. Therefore, they are the entry point to RFL safety vulnerabilities, if any. To this end, we first collect all bug reports and safety-related code reviews among staged and merged RFL code as categorized in § 3.1.

Based on the classification of the RFL issue label [66], we categorize the bugs as compilation and soundness bugs; for kernel concurrency bugs such as deadlock [53] and sleep-in-atomic-context [18], we count them as soundness bugs. We then examine all existing RFL drivers and Rust kernel crates in the upstream repo [22] and analyze the usage of unsafe code blocks.

Results In total, we have found 25 bugs from merged and staged RFL code. Among them, 15 of them are in the Linux mainline and 10 of them are in the stage rust branch. We list them in Table 4. Of the bugs within merged code, 11 are compilation bugs and the other 4 are related to safe abstraction. The compilation bugs do not introduce safety vulnerabilities; they are mostly caused by the misconfig of the kernel, incompatibility of various Clang toolchain versions and the mismatch between the Kbuild and rustc compiler [17]. Of the soundness bugs, 6 are in the safe abstraction layer and break memory safety and 3 break thread safety.

We have not found any unsafe usage of RFL drivers in the Linux mainline, because no serious driver has made into the mainline except for one with around 130 lines of code [50]. However, we have found unsafe cases in the drivers that are proposed to the RFL mailing list. We show the results in Table 3.

Post analysis We audit the bug reports and unsafe code to summarize our findings on RFL safety as follows.

Insight 7: with RFL, Linux becomes more “securable” but still cannot be fully secure.
Our verdict is reminiscent of that of Multics security audit [88] and is based on the following facts.

(1) **Rust safety mechanism constructs the pillar of kernel safety.** The language-level support helps kernel drivers fix existing bugs and eschew potential memory/concurrency bugs. We will present more details in Section 5. As a modern language with rich type specifiers, it facilitates more canonical safety checkers such as klint [31], and RustBelt [86] to further harden the kernel. Compared with C, RFL greatly reduces the vast attack space of kernel software caused by memory bugs. As a result, the developer has much less to reason about in terms of kernel security.

(2) **unsafe is inevitable, though vulnerabilities are optional.** In our audit, we have found unsafe code exist commonly in all major drivers and it is hard, if not impossible, to fully eliminate them. The reason is twofold. First, as kernel asserts full control over memory and hardware, their operations need to bypass Rust ownership checks. For instance, kernel exploῖts inline assembly for managing TLB and issuing memory barrier [67], raw pointers for de-referencing MMIO registers, and unions/bitfields as described earlier in § 3.3. Such operations are out of scope of the ownership mechanism, which essentially is an affine type system. Second, the community sometimes has to compromise on unsafe functions. This is because ownership sometimes introduces twisted implementation, which often requires long reviewing cycles. Prior to the settlement, the community must compromise on a usable API, even though it is unsafe. A notable example we have found is a memory initialization interface pin-init. Despite being known unsafe per the Rust checker, it remains in the safe abstraction layer for over two years. And only after rounds of debates by seasoned developers [12, 57], has the interface been finally fixed in a recent patch [49]. We acknowledge that unsafe code does not necessarily imply vulnerabilities; we argue that it involves sources for vulnerabilities (e.g., MMIOs) which cannot be fully eliminated.

(3) **Bugs do not disappear; they only hide deeper.** On the one hand, kernel functions invoked by the safe abstraction and Rust drivers may still contain bugs and be exploited. On the other hand, although Rust checker detects memory bugs on the spot, it does not detect semantic bugs, which are often caused by subtle differences in Rust and kernel memory allocation methods. Such bugs may take a long to fix and can only be detected by experts who are familiar with both Rust and kernel. For example, the C binder driver has a use-after-free [33] bug. When being re-implemented in Rust, it will not cause the same symptom as in C. Instead, it incurs a mapping bug putting the memory into the wrong place which passes all the checks by the Rust compiler.

### 4.2 Does rust bring any overhead?

This section evaluates RFL drivers against native kernel drivers in C. Our findings suggest Rust drivers incur significant increase in binary sizes; its runtime overhead is comparable to C, but is quite inconsistent across different drivers and configurations with large variations.

**Setup** By exhausting RFL PRs and repositories, we collect

<table>
<thead>
<tr>
<th>Source</th>
<th>Compilation bug</th>
<th>Soundness bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>GitHub [22]</td>
<td>4(1/3)</td>
<td>7(3/4)</td>
</tr>
<tr>
<td>Intel LKP [42]</td>
<td>8(6/2)</td>
<td>0</td>
</tr>
<tr>
<td>Mailing List [45]</td>
<td>4(4/0)</td>
<td>2(1/1)</td>
</tr>
</tbody>
</table>

Table 4: The bugs we have found in RFL. (x/y): x and y stands for number of bugs from merged and staged code respectively. We have not found bugs from Syzbot [41] and KernelCI [38].

<table>
<thead>
<tr>
<th>Driver</th>
<th>Benchmark</th>
<th>Metrics</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVME</td>
<td>fio</td>
<td>throughput</td>
<td>PC</td>
</tr>
<tr>
<td>Null Block</td>
<td>fio</td>
<td>throughput</td>
<td>PC</td>
</tr>
<tr>
<td>EN000</td>
<td>ping</td>
<td>latency</td>
<td>PC</td>
</tr>
<tr>
<td>Binder</td>
<td>ping</td>
<td>latency</td>
<td>Raspi4b</td>
</tr>
<tr>
<td>Gpio pl061</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semaphore</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: The benchmarks and metrics used to test Rust/C drivers. The PC configuration: Intel i54590 with 4 cores, q87 mainboard plus PCIe*16 to m.2, 32 GB DDR3, Samsung SSD 850 Evo, WD SN770, intel 82545 NIC.

![Figure 7: Comparison of Rust and C driver size. * indicates that the Rust driver has not implemented full features as C.](image-url)
4 drivers with serious use cases, which span diverse IO functions (e.g., network, storage). Notably, NVME and binder are considered the first batch of drivers to be merged in the Linux mainline. In addition, we include 2 more toy drivers (i.e., gpio and sem) from RFL rust branch which are often used to explore the difference between Rust and C implementation. Among the 6 drivers tested, only e1000 and the 2 toy example drivers have faithfully implemented full features as C drivers do; while the rest of them have only implemented a subset of the features. For each Rust driver that has a C counterpart, we compile both, compare their binary sizes, and run benchmarks following prior work [97] which represent their typical workloads. Table 5 shows the setup experiments.

**Binary size** As illustrated in Figure 7, Rust drivers with fully implemented features are significantly larger than C: 1.2× for binder, 2.4× for gpio, and 1.9× for sem. Since the text section is to be blamed for most of the increased binary size, we further look into it and find that Rust generates extra code (99%) to support its unique features that C does not have: generic programming, boundary checks, lifecycle management, etc. Even for wrappers that simply invoke kernel functions, Rust expands their sizes by 33%. Interestingly, we find that binder driver in Rust introduces less significant overhead of binary size because it frequently uses function pointers with unsafe keywords rather than generic programming, which instead sacrifices the safety as intended by RFL.

Usually drivers are stripped of debugging information before deployment, but embedded systems may not have enough resources to afford the overhead in debugging mode. We will discuss this in the § B.

**Performance** Mostly, Rust drivers show on par performance with C drivers within a 20% gap. Occasionally, Rust underperforms C significantly; in a few cases, Rust surprisingly outperforms C. Looking into individual drivers, we have found:

- For e1000, Rust driver is 11× slower than C driver for ping latency as presented in Figure 8. The reason we dugged out is that the Rust driver has not yet implemented many features that can accelerate data transmission as the C driver does, e.g., prefetch.
  - For binder, Rust driver shows similar performance with C, with only 10% gap in ping latency.
  - For storage devices (NVME and NULL block), Rust drivers lead to overall similar performance with C, with up to 61% degradation and 67% improvement in throughput, depending on the specific settings (e.g., job number and batch sizes) as shown in Figure 9 and Figure 10. We observe that Rust drivers favor smaller job numbers and blocksize, possibly because Rust often has smaller structs (reasons explained later) which are more likely to fit in the cache line.

We further dig into the potential causes of the performance gap between Rust and C drivers, even with the same feature sets implemented and code path executed. We perform extensive microbenchmarks with the help of kernel tools like vtune and ftrace. We summarize the potential reasons why Rust drivers could perform worse or better, which could help guide the performance diagnosis and improvements for RFL development in the future.

**Why Rust drivers may perform poorly?**
• Locks in Rust drivers are coarse-grained. Despite Rust off-loading the duty of ensuring thread-safety to the language itself (i.e., via rules), it does not lift the burden of high-performance concurrency programming of a developer.
• Rust runtime checks in accessing arrays (e.g., boundary checks) introduce extra performance costs. The results are consistent with the prior study which reports Rust program overhead can be 2.49× larger than C program [110]. Rust performs poorly in memory-intensive workloads.
• Rust uses the emulated bit fields. As introduced earlier in §3.2, bit field accesses are emulated via array accesses. It further gets exacerbated by the runtime boundary checks.
• Rust massively use pointers to share the ownership of objects, which results in a higher cache/TLB/branch miss rate.

Why Rust drivers may perform better?
• The Rust struct has a smaller size compared with the C due to the usage of smart pointers instead of allocating item memory inside the struct. We use pahole to identify that Rust structs use fewer cache lines than their C counterpart.
• Rust driver does not implement full features compared with C, thus some code paths may be omitted.

Insight 8: There is no free lunch for performance – it is the programmer that counts!

4.3 How does Rust improve Linux development?

In this section, we show Rust improves kernel code quality and readability and attracts more engagement from fresh developers.

Improved code quality and readability We use documentation coverage, and CI errors per KLoc as the two important indicators in software quality [76, 106]. For documentation coverage, we examine APIs exported via EXPORT_SYMBOL and EXPORT_SYMBOL_GPL; by kernel convention [43], all of them shall be documented. For CI errors, we report results from the LKP, Syzbot [41] and KernelCI [38]. We compare RFL with ebpf and io_uring, the two new and thriving kernel subsystems as RFL.

As summarized in Table 6, RFL shows superior code and documentation coverage (i.e 100%), compared with ebpf and io_uring. Its average CI errors are also much fewer, by 49% and 68% respectively. There are two key factors for such improvements. First, RFL leverages the rustdoc lints to bundle documentation with code. The feature allows RFL to mandate documentation for all or selected interfaces. In comparison, existing kernel development relies on unspoken contracts or manual review for documentation. Yet, developers often omit writing documentation [4]. For instance, io_uring_cmd_complete_in_task in io_uring is exported via EXPORT_SYMBOL_GPL and is used whenever an IO task needs to be completed asynchronously in the worker thread.

Table 6: The code quality measurement. % means coverage. RFL achieves 100% documentation coverage and least CI errors per 10K LoC.

<table>
<thead>
<tr>
<th>Subsystems</th>
<th>Docs%</th>
<th>CI errors/10K LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFL</td>
<td>100%</td>
<td>3.8</td>
</tr>
<tr>
<td>ebpf</td>
<td>15%</td>
<td>7.5</td>
</tr>
<tr>
<td>io_uring</td>
<td>31%</td>
<td>11.9</td>
</tr>
</tbody>
</table>

Due to its lack of documentation, many developers may mix it up with io_uring_cmd_done, which results in a deadlock [30]. In fact, the community has deemed lack of documentation the #1 bug of Linux [2].

Second, the built-in testing facility of Rust enables early testing for kernel code. Existing kernel development leverages testing framework KUnit [26] and Intel LKP [42], which happens only after the code is staged or already merged. With Rust, RFL incorporates test code easily as a #cfg(test) attribute, which is supported by GitHub CI and will be run automatically every time a PR is submitted. As the testing happens even before the PR is brought onto the table for discussion, RFL incurs much fewer CI errors, implying a higher code quality.

More young blood to the Linux community We compare the developer experience of RFL to recent kernel subsystem (ebpf and io_uring) and traditional subsystem netdev which inspired us to use git commit time as the experience metrics [52]. Similar to netdev, the commits are selected from Linux version 6.1 to 6.4. We quantify developer experience by counting the date of her first commit on the Linux git history to date and categorize them as novice (0 - 24 months), expert (24 - 120 months), and veteran (more than 120 months). Note this is not fully complete as some developers may use different mailing counts during their development process. We plot the distribution in Figure 11.

As we can see, RFL community has the highest percentage of novice developers (58%), 19% higher than ebpf(39%), 20% higher io_uring(38%), and 29% higher than netdev(29%). Interestingly, many of them (29) have never submitted one line of C code. This implies Rust is likely the main factor to attract them into the kernel community, which has been traditionally C-focused.

Such young blood are not yet becoming core Linux developer/maintainers Despite more novice developers being attracted by Rust to the kernel community, we have found their commits are mainly for constructing Rust-relevant toolchains as well as Rust crates alone; they do not, however, take part in kernel code development. By contrast, 5 out of 6 investigated drivers (as seen in Table 5) are mainly contributed by authors from the Linux community. This implies a disconnection between the young and the seasoned developers, and that the bar of kernel programming is not lowered by Rust language.
5 RQ3: What are the experiences and lessons?

We now summarize the key lessons learned in carrying out this empirical study.

To developers on building and using RFL. To make RFL and Rust drivers safer and more practical to use, the developers may consider: 1) not to treat Rust built-in checkers as the holy grail but to leverage other comprehensive analysis tools (e.g., RustBelt [86] and miri [54]), as we have demonstrated the insufficiency of Rust checker in detecting semantic bugs in real drivers (§ 4.1). 2) construct safe kernel abstraction and manage driver resources from the perspective of ownership, instead of memory as traditional kernel programming does. Have clear ownership relations among the structs before starting programming. Otherwise, the cost to fix bugs due to the wrongly used smart pointers in managing ownership will be extremely huge. 3) accept unsafe usage as the final resort. Harnessing kernel memory ops with Rust safety rules often involves massive use of smart pointers and generic programming, which incurs large memory footprints and overhead (§ 4.2). In such cases, the developers may opt for unsafe implementation, as long as they have reviewed its safety carefully.

To RFL community on expanding RFL scope. Prioritizing the development of RFL on future kernel driver/subsystem is crucial, as it takes lots of development efforts and commitment. To understand which driver/subsystem to rustify gives most benefits, we model such benefits as the ratio between accumulative bugs/vulnerabilities of a driver/subsystem which are fixable by Rust, and the scale of the driver/subsystem in terms of LoC. The intuition is simple and backed by the community decision [35]: the smaller the driver/subsystem is and the more memory/thread bugs it may have, the higher the value it has of being rustified. To do so, we traverse the git history of each driver and collect all safety issues by far; we then manually look into all fixed bugs and identify those memory/thread-safety related bugs fixable by Rust safety mechanism (§ 2).

We have analyzed over 2500 drivers spanning 79 different subsystems and plotted the results in Figure 12. On average, each subsystem contain 1.3 bugs/KLoC; across subsystems, the bugs ratio vary a lot. Notably, the linux-block subsystem has a high value due to it contains the most bugs per LoC: 113 data-race bugs plus 98 dangling pointer bugs out of 438 fixes, suggesting the subsystem shall be prioritized. Gladly, the community has confirmed our conjecture and has already rewritten its null block driver with RFL (as tested in § 4.2). Besides linux-block, our results suggest linux-ext4 subsystem also has a high value. Given that the safe abstraction on VFS is already proposed [56] and there also have emerged Rust file systems [32, 36], we expect that RFL next steps into the ext4 file subsystem and hope that RFL can help with the memory/thread safety bugs.

6 Related Works

Understanding Rust in the wild. Prior literature has studied unsafe usage of Rust by analyzing popular Rust projects such as Tock [94], TiKV [6], and Redox [5]. Some of them investigate the unsafe use cases and the rationales [71, 79, 83, 111]. Their findings and insights could guide Rust developers to use unsafe blocks more wisely. Some analyze the bugs in user programs and the Rust compiler and dig into the underlying reasons [103, 107, 108]. Instead, this work studies the RFL project, drawing key insights and lessons from how Rust as a language integrates with the Linux kernel codebase. We also examine whether the RFL project meets its original goals, such as safety improvement and zero-overhead abstraction.

Rust overhead. While designed to be as efficient as C, efforts still have been invested to understand the overhead brought by Rust [73, 81, 110]. They implement a program in both C and Rust and test the performance difference. However, the benchmarks used in those studies are mostly simple, e.g., classic algorithms in less than 100 LoC, and may not be able to reflect

---

2 We cluster the drivers and kernel subsystems into groups based on the dependency and maintainers. We still refer the clustered groups as subsystems for simplicity.
the cases in real-world complex programs, especially operating systems. [81] implements a non-trivial udp driver in Rust and C to compare their difference from an end-to-end perspective. Instead, we perform comprehensive measurements, e.g., runtime metrics and object size overhead, on Linux drivers written in Rust and C for comparison.

**Rust-based kernels.** Rust is a popular language for system software programming in recent years. Both industry and academia are exploring writing new operating systems and kernels in Rust for various use cases, including both embedded devices and personal computers [73, 92–95, 99, 101]. However, those works are mainly purposed to understand the pros and cons of developing Rust-only kernels, without concerning the process of integrating Rust into existing C-based software, which has been shown to be challenging and intriguing. Neither do they consider constructing the safe abstraction to reuse the legacy code. Nevertheless, the findings obtained in this study could be useful for developing Rust kernels as well.

**Rust enhancements.** Efforts have been invested to make Rust safer and more developer-friendly. For instance, some researchers have studied the factors that prevent Rust from being adopted by more programmers and provide guidance to help soften the learning curve of Rust [77, 112]. Others focus on providing defensive mechanisms to improve the safety of unsafe blocks, e.g., using formal verification to verify the correctness of using unsafe blocks, executing program static analysis, providing a sandbox mechanism for using unsafe Rust, [72, 74, 86, 89, 91, 98, 100, 104]. Our study also provides insights towards better integration of Rust with Linux.

7 Conclusions

In this paper, we thoroughly investigate RFL, the very first and increasingly popular project that aims to use Rust to enhance the Linux kernel. We first study the status quo of RFL by diving into the construction of the safe abstraction layer and Rust drivers, which reveals the tension between the language features of Rust and kernel programming. We then look into whether and to what extent RFL has delivered its promise in building a safer kernel with zero overhead. The results show RFL brings better safety but still has leaks which are stealthier and undetectable by the compiler, and that the overhead brought by the tension between Rust and Linux cannot be ceased totally. Last, we summarize key lessons learned in this study, hoping it may guide future development of RFL.

Acknowledgment

This work was supported in part by National Key R&D Program of China (2022YFB4501200) and Natural Science Foundation of Chongqing (No. CSTB2023NSCQ-LZX0115), China. The authors thank the anonymous reviewers for their insightful feedback. The authors also thank the rust-for-linux developers Andreas Hindborg, and FUJITA Tomonori, who kindly helped clarifying our questions in experiment settings and benchmarking, etc.

References


27. An envidence that RFL community lacks of reviewers understanding netdev. https://lore.kernel.org/rust-for-linux/CANiQ72mDVQg9dttbAYLSoxQo42TgyKke-DCe8itwgc0-H05w@mail.gmail.com/, 2023.


35. RFL driver selection reasons. https://lore.kernel.org/all/87ylofj5tt.fsf@metaspace.dk/, 2023.


40. The first rust driver merged into in the Linux mainline. https://lore.kernel.org/rust-for-linux/20231213004211.1625780-1-fujita.tomonori@gmail.com/, 2023.


[56] RFL vfs safety abstraction. https://lore.kernel.org/rust-for-linux/1DX1F6OL9IIDV@suppilovahvero/17234929609368010648.git-patchwork-notify@kernel.org/, 2024.


[68] The null block driver written in Rust. https://lore.kernel.org/rust-for-linux/20230503090708.2524310-1-nmi@metaspace.dk/, 2024.


A Case study: Rust programming inflexibility in the kernel space

We give a real example [96] when Rust safety rules get in the way of kernel programming conventions in Figure 13. The example shows the implementation of a dynamic array of char devices in the kernel and in Rust. The structure definition is in Figure 13 (a). While the two versions of implementation look alike, Rust suffers from a key drawback – the \( N \) which specifies the size of the \texttt{factory} is immutable once set, e.g. \( N \) equals 256 in this case. This immutable \( N \) applies to \emph{each} instance of \texttt{factory}, regardless of their particular needs. For example, a thread \texttt{factory} may want to have a larger size of 256 elements while a proxy \texttt{factory} may only need 8 elements, yet under Rust safety rules, both will have to be of the same size, i.e. either 256 or 8. Such limitation likely creates holes or fragmentation in memory. In comparison, the kernel simply modifies \texttt{len} field via pointers to achieve variable sizes of the array, easily allowing dynamic registration of char devices.

It is, however, possible to overcome the limitation. To do so, the developer may apply a series of workarounds, shown in Figure 13 (b). First, the developer must manually specify a \texttt{dyn_num} trait and its usage (i.e. the function \texttt{use_elements}); it \texttt{dyn} trait type suggests the object is dynamically dispatched. Then, she uses \texttt{const} generic type parameter \( \dagger \) to indicate the number of elements may vary from one instance to another. Lastly, she implements the trait over the generic type to specify the exact size of the instance (i.e. 256 for \texttt{thread_factory} and 8 for \texttt{proxy_factory}). The workarounds not only incur development overhead but also add runtime checks, lowering the performance.

B Driver overhead with debug information

For debugging purposes, the developer may preserve the debugging info of Rust drivers. As illustrated in Figure 14, with debugging information (e.g., debug_foo section), Rust drivers are \( 3.9 \times -6.6 \times \) larger than the C counterparts, even for the Rust drivers with partial features implemented. The debug info of Rust is much heavier than C because Rust exploits generic programming, which results in more symbols and longer symbol names. Such increased sizes of Rust drivers are non-trivial in resource-constrained embedded devices, whose flash and memory sizes are around MB level [102]. Thus, reducing the driver size is critical for enabling debugging in embedded systems [87].

C The kernel community view of Rust

To better understand how the kernel community views Rust, we collect the posts about writing Rust drivers from lwn and ycombinator until 2023/08/05, and use Chatgpt to analyze them. The analysis consists of two parts. The first is sentiment analysis which classifies the opinions into positive and negative ones. The second is opinion mining, which sets to find the reasons behind the opinions. The results can be seen in the Figure 15. In general, Rust is popular for its security and good performance. Yet, its steep learning curve is the biggest concern of the developers. The results suggest Rust still needs time to prove itself in the kernel.
Figure 13: An example explaining the inflexibility of writing RFL drivers.

Figure 14: Comparison of Rust and C driver size with the debug segment. * indicates that the Rust driver has not implemented full features as C.

Figure 15: The opinions of the developers towards using RFL in Linux.