Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel

Amélie Gonzalez, Djob Mvondo, Yérom-David Bromberg

Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel

Amélie Gonzalez
Univ. Rennes - Inria - CNRS - IRISA
Rennes, France
amelie.gonzalez@inria.fr

Djob Mvondo
Univ. Rennes - Inria - CNRS - IRISA
Rennes, France
barbe-thystere.mvondo-djob@inria.fr

Yérom-David Bromberg
Univ. Rennes - Inria - CNRS - IRISA
Rennes, France
david.bromberg@inria.fr

Abstract

C is the primary programming language used in the Linux kernel. Recently, the Linux developer community oversaw the experimental addition of Rust into the kernel's build system. Networking is one of the areas often mentioned when discussing the adoption of Rust. In networking, both perfect memory management and performance are critical.

In this paper, we present a Rust UDP tunneling driver for Linux, which provides UDP encapsulation between two peers. We use this driver to discuss design considerations of writing Rust networking code for Linux. We then compare the performance of our driver against a similar driver written in C. We find that our Rust driver performs slightly worse on a gigabit link for both latency (+0.1906%, p-value = 1.464e−15) and throughput (−0.00090%, p-value = 6.004e−5). We then discuss potential causes for that loss.

1 Introduction

Since its creation by Denies Ritchie in 1972, C has remained the standard programming language for the development of operating systems. It was created with the goal of reducing the amount of assembly used for programming UNIX. C gives developers access to low-level functionality, like bitwise operators, pointers management, manual memory management, and inline assembly. However, improper usage of C is known to lead to unwanted memory issues.

The most representative example of using C for an operating system is the Linux kernel, which also features some assembly language code. It also showcases the issues around writing C source code: dangling pointers, pointers used after free, leaked memory, are all issues that have been found in Linux in particular [6, 27] for decades. Linux is not alone in that respect, but despite its rigorous peer-review process for patch submission, errors still end up into the code base.

Other languages have emerged since C such as Rust, which provides access to similar, low-level operations while also imposing strict verification of its memory safety model at compile-time, and handling as much memory management as possible for its developer. It benefits from being a compiled language, while restricting the potential damage stemming from improper memory management.

Rust is a contender in the realm of programming system languages. Operating systems have already been written with it (Redox [29], Tock [21], RedLeaf [26], Theseus [4]), and it can be used in Linux for driver development since version `6.1-rc1`. Rust has been used to explore the development of Linux drivers for storage devices [16], or direct rendering media [23]. Another area of interest for Rust usage is networking; network drivers allocate and deallocate chunks of memory according to inbound and outbound packets that fly between the kernel and network cards at extremely high speeds. Maintaining performance, all the while avoiding memory handling errors, is critical, and encourages using a language like Rust. These kinds of errors may have catastrophic consequences: they may either freeze portions of the kernel, or kill it entirely [15].

In this paper, we evaluate the use of Rust against that of C for networking drivers in Linux, notably in terms of performance impact. We focus our study on the development of a UDP tunneling driver. Developing such a driver has the advantage of requiring no hardware considerations. At the same time, UDP tunneling is a key building block of technologies such as virtual private networks. UDP is preferred for tunneling over TCP to avoid TCP-over-TCP retransmissions snowballing, or "TCP meltdown". UDP is also simpler of a transmission protocol, as it performs its task with "best effort" guarantees.

We implemented a native Rust UDP tunneling driver in Linux, by extending proposals for code providing access to networking features in Rust for Linux[1]. We then evaluated this driver against another version, written in C. We evaluated both throughput and latency for both drivers over a gigabit duplex link. Our results suggest that Rust suffers from a very slight performance penalty when compared to C, while remaining very close from its level of performance. The Rust driver transfers an average of 914.78 Mbps with an average latency of 127.1 μs, while the C driver transfers 915.79 Mbps for an average latency of 126.8 μs.

2 Background

We will first discuss the use of Rust as a systems programming language, followed by a presentation of the experimental project which brought support for it in Linux, and, finally, the features we targeted for our driver, in the original Linux

[1]Code available at https://gitlab.inria.fr/WIDE/linux-rust-tunnel-driver, tag `plos23-driver`

kernel C code. For the sake of remaining legible, we will not delve too deeply into the behavior of Rust. Readers already familiar with the language can focus on its integration in Linux, while others will benefit from our explanation of the core memory safety rules of Rust.

2.1 Rust for Systems Programming

Rust [11, 17] promises to address the memory safety problems of previous programming languages by combining two concepts baked into its type system: ownership and borrowing. The owner of a memory allocation (a structure, enumeration, scope of code, ...) oversees its initialization, destruction (as soon as the compiler deems it possible), and may move the ownership to another owner. It can also provide *references* to the owned allocation, which may allow modifications or not. The enforcement of memory safety relies on tracking the lifetime of all allocations at compile-time, and checking that at no time there exists a mutable reference alongside any other reference, mutable or not, to the same allocation.

Rust also provides *traits*: sets of properties a type can implement, which guarantee to the compiler, that said type behaves a certain way. For example, the `Copy` trait guarantees that a byte-per-byte copy of a memory allocation of a type that implements it will results in an identical copy of the previous allocation being created. Traits can be combined with generics to provide these features in combination with other types that can, themselves, verify certain constraints. If you implement `Add<T: Foo>` over a type `Bar`, you provide an implementation of addition between your type and any type that implements the trait `Foo`.

There is however a caveat to Rust's safety when programming operating systems and drivers. Since an operating system performs operations such a Direct Memory Access or context switching, there must remain within the code base of a Rust operating system a fraction of source code that cannot comply with compiler checks [22]. Another example is memory-mapped I/O operations: an operating system will have to write at arbitrary addresses in the memory to interact with hardware. In order for developers to perform such operations, Rust has an `unsafe` keyword. Programmers mark functions or blocks of code with `unsafe` to tell the compiler to forego compile-time memory safety checks. In these situations, as long as `unsafe` code is guaranteed to be sound by the developer, all other safety guarantees of the language are preserved. This is one of the challenges that comes with programming using `unsafe` in Rust: only `unsafe` code is known to be able to cause memory problems, but those could manifest somewhere else, including safe code [28].

2.2 Rust for Linux

The Rust API provided in Linux combines core Rust language code for primitive types and standard types implemented by Rust for Linux developers, as well as *abstractions* built around regular C kernel code called through FFI. *Bindings* to

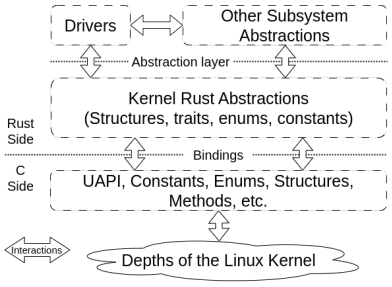

Figure 1. Schematic View of Interactions Between Rust Drivers, Rust Abstractions, and Original Linux C Code

that C code are mechanically generated using `bindgen` [8]. These bindings are then called through `unsafe` Rust code encapsulated in abstractions that provide features for the Rust side of the kernel. In Figure 1, which summarizes this mechanism, we can see a divide between the Rust and C sides of Linux kernel code. Features need to be exposed as bindings, and we have to manually add code in the abstraction layer before Rust drivers can use them. Otherwise, driver developers would need to interact directly with `unsafe` code, which could negate the safety benefits of Rust.

In Rust, the conventional way to hide `unsafe` from end-users of your code is through *abstractions*. Abstractions wrap your `unsafe` code and perform conversions and checks [7]. This is especially useful for `unsafe` code performing foreign function interface (FFI) calls.

Linux C code cannot be verified by the Rust compiler. Rust drivers which use the safe interface provided by abstractions have to trust that the underlying C code and the abstractions are not faulty. As long as the C code is safe, and `unsafe` Rust in the abstractions sound, all other Rust code should be safe [28]. If abstractions are unsound, they might cause memory errors in the Rust drivers. If the C code mishandles memory, it needs to be fixed for the sake of the whole kernel. Fixing errors in C then returns both the C and Rust code to stability. In Linux, Rust developers are forced by style conventions to write a comment, above every `unsafe` block, about why their `unsafe` blocks are sound. Type invariants and `unsafe` function safety prerequisites must also be documented with comments.

Currently, the Rust for Linux (RFL) community focuses on abstractions for core kernel features such as file-system access [25], media peripherals [1], data structures [24], etc.

At the time our work began, none of the few networking abstractions available provided features we required. The available abstractions only allowed to write simple network filters and create TCP sessions.

2.3 Linux and its Network Stack

Memory safety is critical in the `net` subsystem. Network drivers and the network stack have to manipulate packets

as memory on the NIC, and later as memory in the kernel in the form of *socket buffers*. Improper handling could lead to out of bounds access, use-after-free errors, etc. These errors have catastrophic consequences in Linux, usually leading to kernel bugs (an entire subsystem going offline) at best, or kernel panics (entire and sudden crash of the operating system) at worse [15].

Network tunnels are a key building block to create more complex network topology, such as *virtual private networks* (VPNs), so that computers which do not share a common network can still communicate over a shared, virtual, private network. Currently, the `wireguard` Linux kernel module provides VPN functionality using the WireGuard protocol [9].

3 Our Driver and Its Abstractions

As a first step in the exploration of the performance of Rust for native network drivers in Linux, we implemented a UDP tunneling driver written in Rust. Abstractions were necessary to provide these features in Rust. Since we wanted to write a UDP tunneling driver that is efficient, we decided to base our structure on WireGuard's kernel module. WireGuard is a state-of-the-art VPN protocol [9], and its kernel implementation has been thoroughly tested and checked.

3.1 Abstractions

In order to write the drivers, abstractions had to be incorporated on top of the source code made available by the RFL community. We based our work on the October 2022 pull request by Tomonori Fujita [14]. who added a few methods to the wrapper around Linux's network device structure, and a stub for the network device operation factory. We only used and expanded said stub. Approximately 2500 more lines of Rust code were needed, for a total of under 4000 lines added in the `rust` sub-directory. We wrote 45 new wrapper structures, traits and enumerations in order to write our UDP tunneling network driver.

While we could consider upstreaming this work to the Linux kernel, the current policy is to merge code only if it has a user (i.e. a driver). For a driver to be accepted, it needs to provide features not already present in the kernel. We consider that this proof of concept driver does not fit the requirements to be upstreamed, and therefore the abstractions made alongside it cannot be either.

3.2 Initialization

When a C module is loaded, the kernel executes a callback provided in the module declaration that performs all initialization tasks. In Rust, this is represented as a method to be declared for the implementation of the trait `Module` over a type. That type is understood to represent your module, and is instantiated once when your module is loaded by the kernel, and dropped when the module is unloaded. The implementation of the `Drop` trait is the idiomatic Rust equivalent to a C module exit callback. The RFL community has already provided foundations for module declaration that implement basic Linux API in Rust in an idiomatic fashion. Here, we give examples of how we adapted this philosophy while designing our driver and abstractions.

```
1 pub fn register_locked(&mut self) -> Result {
2   to_result(unsafe {
3     bindings::register_netdevice(self.0.get())
4   })
5 }
```

Listing 1. Device Registration Method

Our driver declares a type in its top-level file that represents the module itself, which will simply hold the declaration of a *Router NetLink* type: an interface family you can define in a driver with custom implementation of operations such as interface creation, deletion, queue number gathering, and so on. These operations are useful for drivers that want to provide features not backed immediately by hardware, such as virtual private networks, TUN, TAP, etc.

To provide the set of functions that define a RTNL type for our driver, we use the concept of traits in Rust, by creating a trait called `LinkOperations` which is implemented for one of our types. The trait handles fully abstracting away all bindings from Rust users, only exposing data types that are abstractions around the argument types used in the C version of these callbacks. The implementer type is then passed to a factory type, which will perform the work of creating a descriptor in memory, and register it with the C kernel API. We hold an instance of the factory specialized over our RTNL implementer type to show that the registration is effective.

Before an interface can be shown to users of the system, it has to register itself to the kernel. We use one of the RTNL callback functions we just declared to perform that operation, through a wrapping around Linux's `register_netdevice`. We name that wrapper method `Device::register_locked` (indicating that an environment lock for RTNL data is locked before the method is called). Device configuration methods can be created by wrapping around C bindings, and simply extracting arguments, as shown in Listing 1. Methods like `to_result` are made available by the foundation work of RFL developers. Here, it helps transform `errno` error numbers to a Rust `Result` enumeration. We aimed to remain idiomatic in our driver to comply with the RFL project coding style.

3.3 Network Interface Private Data

Every network interface in Linux can have an area of memory allocated when it is initially created, for the purpose of storing its own information. That area of memory lies at the first 32-byte aligned address beyond the `net_device`

```
let wrap = unsafe {dev.get_priv_data::<Data>()?};
let pref = wrap.initialize_with(Data::new(dev))?;
// Or, if sure that initialization happened
let pref = unsafe {wrap.assume_initialized()};
```

Listing 2. Access to the Private Data Area in Rust

structure, and takes the size declared in the RTNL descriptor registered for our type. It can be accessed in C with `netdev_priv()`, which returns a pointer.

In our Rust driver, we use that memory area to store various information about the peer at the other end of our tunnel, as well as a buffer of socket buffers received and waiting to be processed. Accessing that memory area comes with challenges in Rust. We need to return a mutable reference to a memory area of an unconstrained size at an arbitrary address, that may or may not have been initialized. Our RTNL Rust trait declares an associated type constrained to implement the `Sized` trait (indicating a size known at compile time), which is used to determine the size of private area allocation. We then use a structure called `NetPrivateData` which can be generic over any `Sized` type, and wraps around the pointer returned by `netdev_priv`. In the driver, we then decide whether or not to initialize the area with an instance of the private data type, or assume initialization. Because we cannot ensure that the area is already initialized, or that a zeroed memory area (since the entire area is zero-allocated by the kernel) makes sense for the Rust private data type, both of these operations are `unsafe`. They require that the driver programmer make an informed and conscious decision to use a potentially uninitialized area, as well as what type they provide. An example is given in Listing 2.

In our driver, we know that our access and usage of the device's private data area is safe because the kernel guarantees the order in which RTNL, and other callbacks are called, and we can guarantee that initialization of the memory area happens (and does not fail) before it is used in the next callbacks. This is also true in for network drivers written in C.

3.4 Processing Socket Buffers

Linux's network system manipulates packets through a structure called *socket buffers*, which contains metadata parsed from packet data, and various pointers to a memory area that contain the actual contents of the packet. Manipulation of said pointers allow drivers to rewrite and cut parts of a packet, such as headers.

The networking API grants network drivers access to socket buffer through pointers. The Rust pendant is a mutable reference to a socket buffer abstraction structure created from a pointer at the abstraction border. Said structure reference is passed in the data path of our driver, and wrapper methods help us manipulate the data buffer pointers to remove encapsulation headers at reception.

The current implementation of that abstraction in the `kernel::net` module means we cannot drop socket buffers (as happens upon errors detected in the data path) as easily as we would in C. Rust's type system prevents us from dropping a instance of an object through a reference to it, which is all that is available to drivers in the data path.

In order to compromise on that restriction, we implemented manual drop methods that would free the structure on the kernel side through a mutable reference. This violates the invariant placed on our socket buffer abstraction: "during the lifetime of an instance, the underlying kernel structure must exist and contain a valid socket buffer". The driver tested at the time of submission had no safeguard against a reuse of the abstraction's reference after a drop call, we return from the data path in our driver to avoid reuse. This leaves the type invariant still violated, as the abstraction instance remains valid and potentially usable by other parts of the program.

An actual solution to that issue would involve passing the socket buffer abstraction as an owned instance to functions in the data path. This is possible in one way, if we define the wrapper to contain a pointer to the C socket buffer, instead of being a Rust representation of the C socket buffer structure. Functions in the data path therefore have access to an owner instance (which contains the C pointer), which they are free to drop. For functions branching out from the main chain of function calls, the instance ownership can be moved to these functions, which would then return the instance back if it has not been dropped. Another option for auxiliary functions is to obtain a mutable reference, and return a code signaling to the instance owner (i.e. the main data path caller function) that the socket buffer must be dropped. An improved version of our driver will test this design approach.

Socket buffers show how Rust's type system forces developers of safe wrappers to carefully consider the way data is handled, and how difficult that design step can be.

4 Evaluation

We will present the final rounds of evaluation against the current version of the driver, our setup, results, and interpretation. The evaluation will cover performance in terms of network latency and throughput for our Rust driver, a C implementation, and the link they use for tunneling.

4.1 Evaluation Setup

We used two Intel NUCs, model `NUC7i7BNH`, connected to a Cisco Catalyst 2960-S switch with category 5.E RJ45 cables. They each have 4 Intel Core i7–7567U CPUs, paced at 3.50 GHz. Their NIC is an Intel Corp. Ethernet Connection I219—V, revision 21. The link is negotiated as 1000 Base-T full duplex. We used a build of Linux `6.3.0`, based on the configuration of Linux `4.15.0-197-generic` available for

Interface	Mean	Min	Max	σ	Points outside 95% interval
Baseline	122.2	117	127	1.10	176
C	126.8	120	132	1.37	273
Rust	127.1	121	134	1.34	176

Values computed over n = 4000 latency mean measurements.

Table 1. Latency Measurements in Microseconds

Ubuntu `18.04.6`. The configuration was updated with default values and we enabled Rust and our modules. Our tree is based on RFL's `rust` branch (commit `bc22545f`).

Performance tests consist of a latency test and a throughput test. Latency is measured with the `netperf` [18] utility's `TCP_RR` test over 100 runs consisting of 5 seconds of warmup with the `ping` command from the `iputils` suite (version `iputils-s20161105`), 60 seconds of measurement, and 5 seconds of cooldown. The mean latency, 50%, 90% and 99% percentiles are taken each run. Throughput is measured with `netperf`'s `TCP_STREAM` test, with the same number of runs, structured the same way. Average throughput is measured each run. For latency, `netperf` reports percentiles with three significant figures, but keeps five to compute the mean. We round that mean to only keep three significant figures. Throughput means are kept at five significant figures as given by `netperf`, considering that it gives a precision to the tens of kilobits per second. We do not run download tests in both directions because our setup is symmetric.

As for our drivers, we argue that our C and Rust drivers are comparable for the following reasons: the Rust driver makes use of the same C Linux networking API provided by Linux, but wrapped in safe abstractions. When writing the C driver, we simply called the same methods wrapped for the Rust side, in the same order, for the same processing steps.

4.2 Benchmark Results

In out experiments, the baseline represents performance of the link used to connect both test devices.

Latency. Summarized results for the latency tests are provided in Table 1. These values are computed over the output mean for each individual test run. For latency, lower is better. Overall, the C and Rust drivers are very close to one another in terms of latency, with only a slight gap of under half a microsecond between their means. A t-test shows that this difference is significant over our n = 4000 points, with p-value = 1.464e−15. Our Rust driver has slightly worse minimal and maximal observed latency as well. The standard derivation σ is slightly higher for C by about 0.03 ns, but we cannot reject the hypothesis of a statistical error (p = 0.9449). We can therefore assume the latency distributions of both

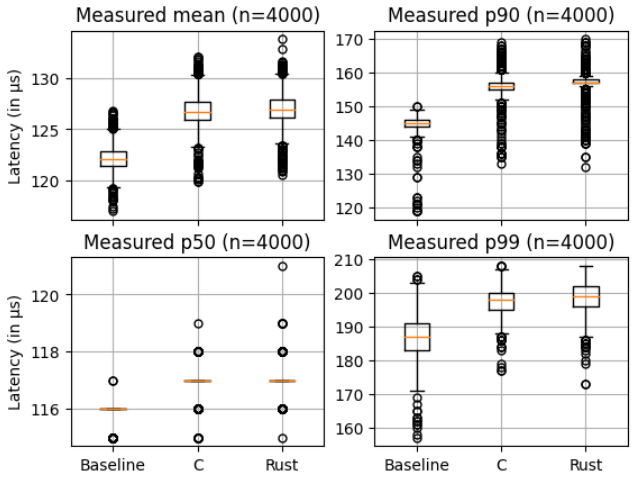

Figure 2. Latency Metrics for the C and Rust Drivers

Interface	Mean	Min	Max	σ
Baseline	934.30	930.95	934.39	7.97e−2
C	915.79	913.92	915.93	8.15e−2
Rust	915.78	911.89	915.92	1.03e−1

Table 2. Throughput Statistics in Mbps

drivers share the same standard derivation. Both drivers are slower than the baseline. These results concord with Figure 2, which shows that the trend for latency in Rust is slightly higher than C, not only in terms of means, but also medians, as well as 90$^{\text{th}}$ and 99$^{\text{th}}$ percentile. These results, in particular the significant statistical difference observed for latency means, suggests a slightly higher processing time taken in the data path of our driver. That difference could be attributed to poor programming on our part, as well as the presence of some performance concerns observed by the RFL community, which we discuss in subsection 4.3.

Throughput. Throughput measurements statistics presented in Table 2 show very close performance between the Rust and C drivers. The last column of that table shows the number of points in each category outside of the 95% confidence interval.

The baseline interface has throughput values measured between 930.95 Mbps and 934.39 Mbps. With abnormal values removed, the C driver transfers around 915.79 Mbps, while the Rust driver is a bit behind, with only 915.78 Mbps on average. The bitrate variation between both drivers is around ten kilobits per second on average. Statistical testing show that this difference is still significant (p = 6.004e−5). We interpret this to mean that our Rust driver reaches the same level of performance as the C one, but with a small

overhead of about 0.00090%. However, Rust usage in our driver, whether from the language or our use of it, slows down traffic ever so slightly.

Overall, our Rust driver, as a whole, is slower in both latency and throughput.

4.3 Missing Bits & Limitations

One limitation identified in our tests is that a one gigabit per second of throughput is not enough to put devices involved in the test under stress. This was the best bare-metal hardware available to us at the time. It is very clear to us that further experiments must take place with better hardware.

Regarding our driver and its integration within Rust for Linux, we discern problems existing with our current implementation (missing features, or shortcuts taken to counter strict Rust typing), as well as known limitations of RFL as it exists today.

Missing Features. It is currently impossible to delete an interface created with our Rust driver. Reference counting is improperly handled, as we do not delete all resources in the private data area, including a counted reference to the device itself. Any attempt will result in the kernel's entire network subsystem freezing while waiting for references to be freed. We currently rely on shortcuts taken in the Rust driver code because of time constraints. One of them is a cast from a socket buffer reference to a pointer, which is converted back through `unsafe` code later on. This shortcut was taken to get around lifetimes of socket buffers and quickly put together a working driver. While this stems from our trying to push problems to later, it illustrates the fact that Rust's strict security model may feel like an obstacle to programmers, who may be tempted to use and abuse `unsafe` to bypass it.

Current limits of RFL. A limitation we can attribute to using Rust in Linux is the use of `unsafe` in our driver for a direct call to the binding of `udp_tunnel_xmit_skb` (a method of the `udp_tunnel` kernel module). It is currently impossible to write abstractions for `udp_tunnel` in the `kernel` crate of the `rust` subsystem. Said crate is linked into the standalone `vmlinux` kernel binary, while `udp_tunnel` and its method stay alone in their own module binary file. The linker cannot resolve the call to a binding to `udp_tunnel_xmit_skb`. The only solutions appear to be either writing a kernel module that uses bindings to provide abstractions to the C UDP tunneling module in Rust, or otherwise rewriting `udp_tunnel` using Rust networking abstractions which do not exist at the moment.

Furthermore, we can only empirically verify that our Rust code is valid and provides the wanted features. While there is existing work trying to create verification tools for `unsafe` Rust [28], it is not known whether they can handle constraints specific to methods calls via FFI. None of the abstractions written for Rust for Linux are mechanically and rigorously proven sound.

Finally, the problem of inlining through FFI is Rust is still under investigation [2]. Inlining by LLVM is automatically done within Rust code. However, the current Rust build infrastructure of the kernel makes it such that inlining is not available between C code compiled with a C front-end of LLVM (eg. `clang`), and a Rust front-end of LLVM (eg. `rustc`). *Link Time Optimization* (LTO) would need to be enabled in the build system. Developers of RFL have already discussed supporting that feature [2].

5 Other Driver proposals and Related Work

Usage of Rust in systems programming is not new, even in terms of networking. However, we contribute a first look into using Rust for developing networking drivers in Linux.

5.1 Other Abstraction and Driver Proposals

Others have undertaken the task of making network drivers for Linux in Rust, albeit with different goals.

In July 2021, Finn Behrens opened a request to inspect and merge his RTNL and network device manipulation abstractions [3]. It proposed changes that now require rewrites of code later merged in Rust for Linux's experimental tree.

Tomonori Fujita developed his own `e1000` out-of-tree Rust driver [12]. It uses direct calls to bindings, and relies on his original October 2022 RFL proposal [14]. He posted a first draft of his more complete networking abstraction patch set in May 2023. The patch set covers basic representation of network devices, network namespaces, handling of socket buffers, network device operations and Ethernet operations. While there is an intersection with our own abstractions, it is slim. Tomonori Fujita focuses on writing hardware drivers, while we wanted to focus on tunneling. His proposal is currently being reviewed by the `netdev` community [13]. We chose not to rewrite our code until the Linux community decides on what set of abstractions to adopt into mainline.

5.2 Related Work

Emmerich et al. have previously made network drivers for Linux in higher level languages [10]. Rust was included in that list. Their drivers all resided in userland, and used kernel bypass with DPDK alongside custom-made wrappers. They also showed that Rust reaches performance close to, but ever so slightly under C. They also address the cost of safety in Rust to explain that performance gap.

Redox [29] is a general purpose UNIX-like operating system developed in Rust. It is still actively developed, and can provide a functional desktop environment. Its developers have chosen a micro-kernel approach, and run their drivers in userspace. Redox is network-capable. The work of Emmerich et al. also evaluated Redox's networking. They noted, at the time, that these drivers were not optimized and a poor performance compared to their C userspace driver. It could

be interesting to study the current performance of Redox's network stack [30], which is based on smolTCP [31].

Tock [21] is an operating system for embedded hardware. It is the fruit of research around the benefits of writing kernels in Rust [22]. In their design, however, Levy et al. focused on devices such as security keys, operations such as USB access or DMA, and did not discuss design considerations or examples of networking in Tock.

Isolation of trusted and verified code has been explored, including in Rust [5, 26]. Kirth et al. developed PKRU-Safe [19], a combination of a compiler plugin and allocator for the Rust toolchain that leveraged Intel Memory Protection Keys (MPK) to protect safe memory from potential errors or exploits stemming from the unsafe code. With their design, developers annotate frontiers between unsafe and safe code. Combined with profiling, PKRU-Safe can determine which allocations are used in the unsafe region, and which are not. These mechanisms could be applicable in our situation, with heavy modification of the Linux build system. Sandcrust [20] is a sandboxing system built around process isolation and accessible in Rust with a macro crate. The approach involves serialization and transmission of data through pipes, which incurs high overhead. Sandcrust relies on process isolation, meaning it can only exist above the kernel.

6 Conclusion & Further Work

In this paper, we present our work on a UDP tunneling driver written in Rust in Linux. Rust is especially useful for network drivers, as they need to manage memory perfectly while also remaining extremely efficient.

Overall, once abstractions are available to use the kernel's networking API, writing a network driver in Rust in Linux is possible. Yet, we note that Rust's type system and safety checks make that process complex, and requires a lot of careful planning to safely wrap C API. Handling pointer structures, such as with socket buffers, is especially complicated. We show that our Rust driver performs only very slightly worse in terms of performance than a C equivalent. The Rust driver exhibits a statistically significant difference of means across samples for both latency (+0.1892% compared to the C driver) and throughput (−0.0011%) on our gigabit link.

We envision exploring our driver scalability, notably on better hardware and bigger links. These bigger links (10, 25 or even 100 gigabit per second) will likely create higher CPU stress and allow us to get a clearer picture of current processing bottlenecks of Rust networking in Linux.

At the same time, we will address shortcuts and missing features discussed above, with the goal of closing the performance gap with C while providing as safe a driver as possible.

References

[1] Daniel Almeida. 2023. [PATCH 0/6] Initial Rust V4L2 support. https://lore.kernel.org/rust-for-linux/20230406215615.122099-1-daniel.almeida@collabora.com (Message to the Linux Kernel Mailing List).

[2] Emilio Cobos Álvarez. 2023. Re: [PATCH 0/5] Rust abstractions for network device drivers. https://lore.kernel.org/rust-for-linux/48a98d0c-bfd1-68a9-5d1f-65c942b7c0ef@crisal.io (Message to the Linux Kernel Mailing List).

[3] Finn Behrens. 2022. *DRAFT: [RFC]: Add Rust net_device wrappers*. GitHub. https://github.com/Rust-for-Linux/linux/pull/439

[4] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[5] Anton Burtsev, Dan Appel, David Detweiler, Tianjiao Huang, Zhaofeng Li, Vikram Narayanan, and Gerd Zellweger. 2021. Isolation in Rust: What is Missing?. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 76–83. https://doi.org/10.1145/3477113.3487272

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01)*. Association for Computing Machinery, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

[7] Rust Community. 2023. FFI — The Rustonomicon. https://doc.rust-lang.org/nomicon/ffi.html#foreign-function-interface

[8] Rust Community. 2023. `rust-bingen` — *Automatically generates Rust FFI bindings to C (and some C++) libraries*. The Rust Programming Language. https://github.com/rust-lang/rust-bindgen/

[9] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. NDSS. , 12 pages. https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_04A-3_Donenfeld_paper.pdf

[10] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. 2019. The Case for Writing Network Drivers in High-Level Programming Languages. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, Trinity Hall, Cambridge, UK, 1–13. https://doi.org/10.1109/ANCS.2019.8901892

[11] The Rust Foundation. 2023. *The Rust Programming Language*. The Rust Foundation. https://rust-lang.org

[12] Tomonori Fujita. 2023. *fujita/rust-e1000*. GitHub. https://github.com/fujita/rust-e1000

[13] Tomonori Fujita. 2023. [PATCH v2 0/5] Rust abstractions for network device drivers. https://lore.kernel.org/rust-for-linux/20230710073703.147351-1-fujita.tomonori@gmail.com

[14] Tomonori Fujita. 2023. *rust: add initial netdevice support*. GitHub. https://github.com/Rust-for-Linux/linux/pull/908

[15] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. 2003. Characterization of linux kernel behavior under errors. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* 459–468. https://doi.org/10.1109/DSN.2003.1209956

[16] Andreas Hindborg. 2022. Linux Rust NVMe Driver Status Update. https://lpc.events/event/16/contributions/1180/

[17] Graydon Hoare. 2010. *Project Servo*. Mozilla Annual Summit, Whistler, Canada. http://venge.net/graydon/talks/intro-talk-2.pdf

[18] Rick Jones. 2015. *Care and Feeding of Netperf 2.7.X*. Hewlett Packard. https://hewlettpackard.github.io/netperf/doc/netperf.html

[19] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between

Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[20] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS'17)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[21] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[22] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

[23] Asahi Lina. 2022. *Tales of the M1 GPU*. Asahi Linux. https://asahilinux.org/2022/11/tales-of-the-m1-gpu

[24] Asahi Lina. 2023. [PATCH v3] rust: xarray: Add an abstraction for XArray. https://lore.kernel.org/rust-for-linux/20230224-rust-xarray-v3-1-04305b1173a5@asahilina.net (Message to the Linux Kernel Mailing List).

[25] Ariel Miculas, Miguel Ojeda, and Wedson Almeida Filho. 2023. [RFC PATCH 00/80] Rust PuzzleFS filesystem driver. https://lore.kernel.org/rust-for-linux/20230609063118.24852-1-amiculas@cisco.com (Message to the Linux Kernel Mailing List).

[26] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[27] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 305–318. https://doi.org/10.1145/1950365.1950401

[28] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[29] Redox. 2023. The Redox Operating System. https://www.redox-os.org/

[30] Redox. 2023. *redox-os/netstack*. Redox OS. https://gitlab.redox-os.org/redox-os/netstack

[31] smolTCP. 2023. *smoltcp-rs/smoltcp: a smol tcp/ip stack*. GitHub. https://github.com/smoltcp-rs/smoltcp