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

Amélie Gonzalez, Djob Mvondo, Yérom-David Bromberg

University of Rennes — IRISA — Inria — CNRS
Brittany, France

Workshop on Programming Language and Operating Systems (PLOS'23)
October 23rd 2023

Context: The Linux Network Stack

In Linux, network handling/the stack is located in the `net` subsystem.

- We know there are other faster means of networking [1]
- Written in C, doing its best to be fast AND memory safe
- Drivers still cause most errors bugs [2,3]
- A memory error in the data path can have *catastrophic* consequences
- Slow code in the data path can also have *catastrophic* consequences

Networking $\implies$ processing data extremely fast and without any faults

[1] Høiland-Jørgensen et al., "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel" (CoNEXT '18)

[2] Chou et al., "An Empirical Study of Operating Systems Errors" (SOSP '01)

[3] Palix et al., "Faults in Linux: Ten Years Later" (ASPLOS XVI)

Context: Rust for Linux

Rust:

- Strong memory safety verifications at compile time
- Flagship AOT compiler (`rustc`) based on LLVM
- Great efforts to interface with C/C++, notably with `bindgen`[1]
- Advertises zero-cost abstractions

The Rust for Linux (RFL) project:

- Officially started around 2020
- Great efforts to build a Rust ecosystem in the kernel
- Still very early in the experimental phase

So why not do networking in Rust?

[1] See `rust-lang/rust-bindgen` on GitHub

Leading Questions

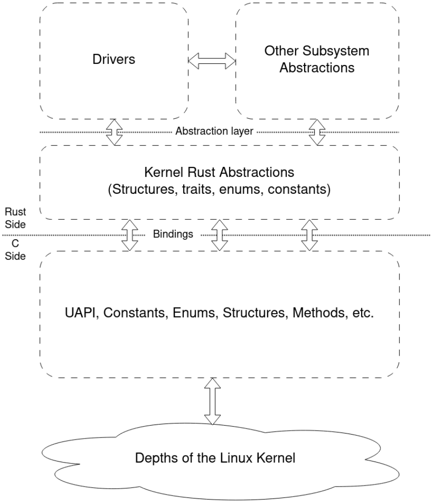
- In order to study the impact of Rust, we focused on the network stack (adjacent to some of our other work)
  - Is there a latency impact? How significant?
  - Is there a throughput impact? How significant?

  Previously other supports have been studied, like NVMe support[4]
- **Contribution**: an evaluation of a Rust network driver VS a driver that performs the exact same function in C, with an unmodified build system of Rust for Linux

  Other contribution: discussions within the RFL project about how we should handle abstraction development

[4] Hindborg, Linux Rust NVMe Driver Status Update (Linux Plumbers Conference '22)

The General Design Plan



- C kernel code remains unchanged
- Bindings to methods are created by `bindgen`
- C code is **not verified** by `rustc`, so C calls are `unsafe`
- Rust code wraps `unsafe` code in safe data structures
- Drivers build atop the safe abstractions, and as little `unsafe` code as possible

A Challenging Task

I will present **three challenges** we faced while developing the driver

They involve the driver code as much as the abstractions

Challenge 1: Idiomatic yet familiar

Old habits die hard, memory bugs (hopefully) die harder

At the same time:

- Our Rust code needs to **remain idiomatic**
- Our Rust code should **act similarly** to the C code it's interfacing with

`module_init`, `module_exit`, ⋮ } ↦ `trait kernel::module::Module`

C function descriptors ↦ Rust traits

C constants ↦ Rust constants or enums

`(const)? struct my_c_type*` ↦ `&'a (mut)? MyRustType`

Takeaway: Stratagems

General stratagems for creating abstractions (but those are *not* rigid rules)

Challenge 2: Oddities of Net & C Kernel Programming

The network stack API is designed to be used in C:

- Descriptors of function pointers
- (Checked) direct access to areas of memory
- Typecasts of memory areas at the discretion of the drivers

Those are *hard* to transfer to a memory-safe language

Regarding the network device's private data area:

- **First solution**: `unsafe` methods, cast the private data area to a `Sized` type
- Next idea: associated Types in Traits + strong use of typing in drivers

Takeaway: Problem-Solving Approaches

Many approaches to wrap `unsafe` outside of inner-unsafe:

- sometimes granting drivers `unsafe` is simpler
- Crafting typing rules may not be that easy

Challenge 3: Socket Buffers

Socket buffers have a complex API...

`struct sk_buff` ≡ packet(s) + metadata

Packets have to be handled in the *data path*, dropped, and data inspected.

~~Dirty method: drop through a `&'a mut`, then return~~

Requirements:

1. **Ownership of the abstraction** (`SkBuff` ↔ `struct sk_buff*`)
   Combined with a custom `Drop`, and a field that stores the skb drop reason
2. Safe wrappers to return regions of the buffer as `&[u8]`
3. Safe wrappers to force-cast buffer data to headers
4. All trimming/pushing/setting/getting functions in the abstraction

Takeaway: Typing is here to help

Use Rust's type system to your advantage, stray away from stratagems when it's more convenient

In the end

WgRS/RustyPipe:

- Structure based on `wireguard`
- Point-to-Point UDP Tunneling
- No cryptography
- NAPI-enabled
- Managed with `ip(8)`
- Peers hard-coded

a C version doing the same thing:

- based on `wireguard`
- Point-to-Point UDP Tunneling
- No cryptography
- NAPI-enabled
- Managed with `ip(8)`
- Peers hard-coded

Both drivers follow the same steps, use the same API
Only one of them uses FFI, wrappers and Rust's core code

Evaluation Setup

- Run latency + throughput benchmark between two machines with a Rust-enabled kernel and our tunnel modules deployed
- Tool: `netperf`, `TCP_RR` and `TCP_STREAM` tests
- Setup: Intel NUCs, model NUC7i7BNH, 4-core Intel Core i7–7567U CPUs
  1 Gbps duplex link on a Cysco Catalyst 2960-S switch
  It was our best bare-metal setup available
- One run = 60 seconds of run + 10 seconds of cooldown
  4000 runs for baseline, C and Rust (12000 runs total)

Results

Latency: $p = 1.464e{-15}$

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

Throughput: $p = 6.004e{-5}$

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	$1.03e{-1}$

What we learned

On unmodified RFL as of July 2023 with no build optimization:

1. There is a **measurable** impact on throughput and latency
2. Making a Rust network driver is daunting but **very much doable**
3. A **non-trivial amount of work** is necessary ahead of driver development to even make it possible
4. Once our abstractions were deemed (but not proven) **sound**, we never encountered memory errors

Going Forward

1. Improvements to the driver
2. Digging into precise reasons for the performance loss, notably the lack of LTO
   LTO was important in the NVMe driver experiment[4]
3. Working on improving abstractions for more sound foundations

[4] Hindborg, Linux Rust NVMe Driver Status Update (Linux Plumbers Conference '22)