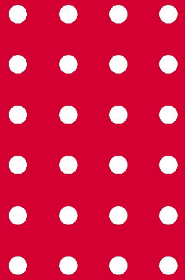


(Re-)implement a linux kernel driver in Rust

Riccardo Strina, Robert Hernandez, Simone Mannarino

CS 594 Professor Xiaoguang Wang

Spring 2024

From C to Rust: Unveiling the Motivations

1. Memory safety: Rust's ownership system prevents common memory errors like null pointer dereferencing and buffer overflows, which are frequent sources of bugs and security vulnerabilities in C.
2. Concurrency safety: Rust's ownership model and built-in concurrency primitives enable safer and easier concurrent programming, reducing the likelihood of data races and deadlocks.
3. Developer productivity: Rust's expressive type system and powerful compiler provide better error messages and static analysis, leading to faster development cycles and fewer bugs.
4. Security: Rust's safety guarantees make it harder for attackers to exploit vulnerabilities, enhancing the overall security of the kernel.
5. Ecosystem: Rust's growing ecosystem of libraries and tools can enhance code reuse and enable faster development of kernel features.

Functionality of the C driver

The pre-existing driver came from HW9

Driver for a virtual device written in C that exposes an ioctl interface through /dev/lkp_enc

The interface supports reading and writing a single integer seed and reading and writing a null terminated character string of 4092 or less characters

Building a working kernel with Rust enabled

Using the Rust-For-Linux fork of the kernel found at
https://github.com/Rust-for-Linux/linux/tree/rust

Or any version of the kernel after v6.1

Install Rust and set the required version by the kernel

Install *clang, llvm, lld* and the crate *bindgen*, for Rust to C bindings

Compile the Kernel using LLVM with the Clippy linter enabled "*$ make LLVM=1 -j4 CLIPPY=1"*

Every kernel comes with scripts that will do most of these steps for you

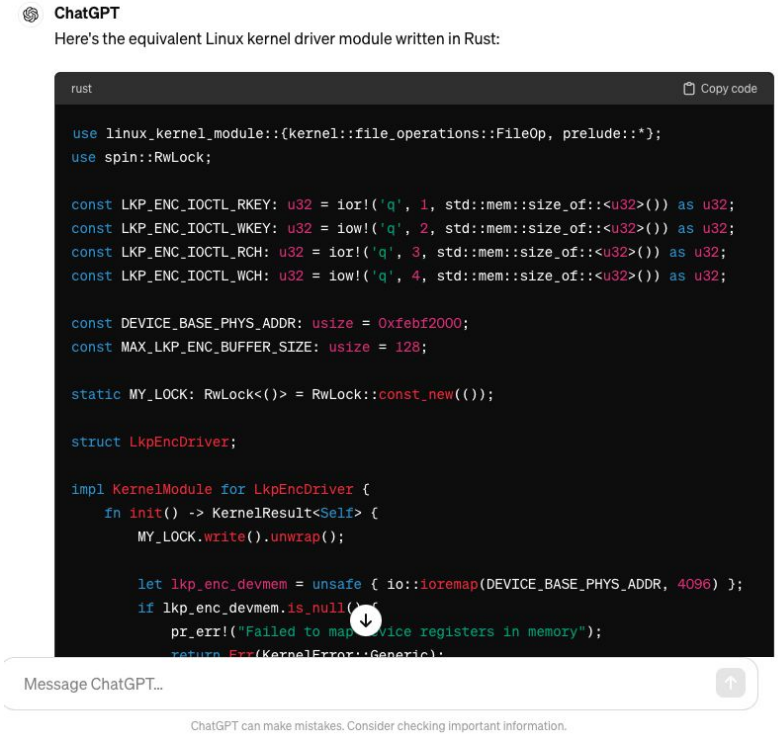
UIC

Leveraging Large Language Models

Proper prompt engineering is important to getting workable output from LLMs. Given we had more than one potential C driver candidates, after extensive discussion and testing of each by each group mate, we selected a driver as a base and as input into ChatGPT 3.5 which was used to generate the first draft of the Rust Driver with the following prompt, followed by the full text source of a C version of the driver:

"Read the following linux kernel driver module written in C, convert it to a module written in Rust:"

ChatGPT 3.5 then output early draft Rust code that was used as inspiration when drafting the second version of the driver. The ChatGPT output was as expected not compilable without proper setup and Bindings.



C Concepts into Rust

Differences:

1. Memory management:
   - C uses manual memory management with malloc and free, which can lead to memory leaks and segmentation faults if not handled properly.
   - Rust uses a system of ownership, borrowing, and lifetimes, enforced at compile time, to ensure memory safety without sacrificing performance.
2. Concurrency:
   - C lacks built-in support for safe concurrency, leading to potential data races and undefined behavior in multi-threaded programs.
   - Rust provides safe concurrency through its ownership model and language features like threads, channels, and the `async`/`await` syntax, ensuring thread safety at compile time.
3. Safety:
   - C does not have built-in safety features, allowing for undefined behavior, memory corruption, and security vulnerabilities if not carefully managed.
   - Rust guarantees memory safety and thread safety through its ownership system and strict compile-time checks, reducing the risk of bugs and security vulnerabilities.
4. Syntax and Features:
   - C has a simpler syntax and fewer language features compared to Rust.
   - Rust has a more modern syntax and includes features like pattern matching, algebraic data types, iterators, and generics, which can lead to more expressive and concise code.

C Concepts into Rust (cont.)

Similarities:

1. Low-level capabilities:
   - Both C and Rust are low-level programming languages that allow direct manipulation of memory and hardware, making them suitable for systems programming tasks.
2. Performance:
   - Both languages are designed for high-performance computing, allowing close control over system resources and efficient execution of code.
3. Tooling:
   - Both C and Rust have mature toolchains, including compilers, debuggers, and profilers, making them well-suited for developing and debugging system-level software.
4. Compatibility:
   - Rust has interoperability with C, allowing Rust code to call C functions and vice versa, making it easier to integrate Rust into existing C codebases or use C libraries from Rust.

Is it all Rust?

In the current state of the kernel, Rust is used mainly as a wrapper for C functions, few things have been ported over.

Are the benefits of Rust lost doing so?

NO! The way wrappers are done allows for a better and safer development when compared to C

Rust is based on *Struct*s, for every *Struct* one or more *Trait*s can be implemented.

A *Trait* defines a behavior that a *Struct* must have

Implementation of Drop

```
impl<const SIZE: usize> Drop for IoMem<SIZE> {
    fn drop(&mut self) {
        // SAFETY: By the type invariant, `self.ptr` is a value returned by a previous successful
        // call to `ioremap`.
        unsafe { bindings::iounmap(self.ptr as _) };
    }
}
```

Many Rust wrappers implement the *Drop* trait, when the *Struct* is no longer referenced an action will be executed to clean the memory footprint of the struct, avoiding memory leaks.

How many times have we forgotten to destroy a rbtree, a linked list or to simply call kfree?

A well implemented wrapper will do that for us!

What is wrapped then?

The latest kernel (v6.9) comes with a lot of wrappers already implemented but few things can be done.

A simple module can be defined, the *task_struct* of the current scheduled process can be retrieved but not much more.

How was possible to (re)-implement a driver then?

Rust-for-linux branch "*rust"* was the branch to show the world how good Rust is for kernel development.

In that branch many more wrappers are available and are not bound by the strict rules of mainstream kernel.

Wrappers for *file_operations*, *copy_to/from_user* exist and we leveraged those to write our driver.

UIC

Integration Testing

```
ubuntu@ubuntu ~
$ sudo ./user_space_app 3 "Hello, World"
Original string Hello, World
Encrypted string Khoor, Zruog
ubuntu@ubuntu ~
$ neofetch
                    .-/+oossssoo+/-.               ubuntu@ubuntu
                `:+ssssssssssssssssss+:`           -------------
            -+ssssssssssssssssssyyssss+-           OS: Ubuntu 22.04.3 LTS x86_64
          .ossssssssssssssssssdMMMNysssso.         Host: KVM/QEMU (Standard PC (i440FX + PIIX, 1996) pc-i440fx-8.2)
        /ssssssssssshdmmNNmmyNMMMMhssssss/         Kernel: 6.3.0-g18b749148002-dirty
      +ssssssssshmydMMMMMMMNddddyssssssss+         Uptime: 7 mins
    /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/         Packages: 759 (dpkg), 4 (snap)
   .ssssssssdMMMNhsssssssssshNMMMdssssssss.        Shell: zsh 5.8.1
   +ssssshhhyNMMNysssssssssssyNMMMysssssss+        Terminal: /dev/ttyS0
   ossyNMMMNyMMhsssssssssssssshmmmhssssssso        CPU: QEMU Virtual version 2.5+ (4) @ 2.027GHz
   ossyNMMMNyMMhsssssssssssssshmmmhssssssso        GPU: 00:02.0 Vendor 1234 Device 1111
   +ssssshhhyNMMNysssssssssssyNMMMysssssss+        Memory: 89MiB / 3925MiB
   .ssssssssdMMMNhsssssssssshNMMMdssssssss.
    /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/
     +sssssssssdmydMMMMMMMMddddyssssssss+
      /ssssssssssshdmNNNNmyNMMMMhssssss/
       .ossssssssssssssssssdMMMNysssso.
         -+sssssssssssssssssyyyssss+-
           `:+ssssssssssssssssss+:`
               .-/+oossssoo+/-.

ubuntu@ubuntu ~
$ uname -a
Linux ubuntu 6.3.0-g18b749148002-dirty #5 SMP PREEMPT_DYNAMIC Mon Apr 22 00:52:05 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
```

LIVE DEMO

RECORDED DEMO

https://www.youtube.com/watch?v=0X8t5_0mo70

The Road Ahead: Advantages, Considerations, and Future of Rust Drivers

Whatever can’t be directly ported from C to Rust requires a wrapper.

A wrapper is a good abstraction from the C code and it requires a single implementation for then enjoying it throughout all the code.

Porting Kernel code in C to Rust is roughly functional. An LLM, like Chat GPT, can help you get started porting a driver from C to Rust but LLMs require proper context to be proficient. Tooling like rust-analyzer and clippy can be useful in providing a developer context for prompt engineering, development, and debugging.

Known Issues

UserSlicePtr (wrapper for copy_to|from_user)

- The function is sometimes not able to retrieve the values from user space into kernel space

io-mem.rs (wrapper for ioremap and iounmap)

- Even if it exists it wasn’t possible to use it because of privacy settings in the module

ioctl.rs (port for ioctl.h)

- Only available in newer kernel versions, we had to include it in the “*rust*” branch


Thank you for your attention!
Q&A
UIC