Integrating Tyche client libraries as a Rust-based Linux kernel module

Bachelor semester project

Noé Terrier

École Polytechnique Fédérale de Lausanne

EPFL

Integrating Tyche client libraries as a Rust-based Linux kernel module

Bachelor semester project

by

Noé Terrier

Student Name	Student Number
Noé Terrier	310041

Instructor:	Prof. Edouard Bugnion
Teaching Assistants:	Charly Castes & Yuchen Qian
Laboratory:	Data Center Systems Lab at EPFL
Date:	Spring semester 2023
Faculty:	School of Computer and Communication Sciences, EPFL

Cover:	The Rolex Learning Center at EPFL (Modified)
Style:	EPFL Report Style, with modifications by Batuhan Faik Derinbay

EPFL

Contents

1 Introduction **1**

2 Rust language support in the kernel **2**
- 2.1 Dependencies . . . 2
- 2.2 How to activate Rust support within the kernel . . . 2

3 Write a first module in Rust **3**
- 3.1 Writing the module . . . 3
- 3.2 Building . . . 4
- 3.3 Limitations of the Rust support inside the Linux kernel . . . 5

4 File operations and IOCTL **6**
- 4.1 File operations . . . 6
- 4.2 Input / Output control (IOCTL) . . . 7

5 Memory allocation and mapping **9**
- 5.1 Page allocation . . . 9
- 5.2 Memory mapping . . . 9

6 Using library crates inside a module **11**
- 6.1 Crates specifications . . . 11
- 6.2 Use a library crate inside a module . . . 12

7 Comparison with the C language **14**

8 Conclusion **15**

References **16**

A Rust and C comparison programs **17**

1

Introduction

This bachelor project is based on the development of the Tyche project: a project that aims to improve confidentiality and trust inside a computer system. The following text is a description of Tyche drawn from the paper *"Creating Trust by Abolishing Hierarchies"*: [3]

We present Tyche, a prototype implementation of an isolation monitor, designed to be formally verifiable, with a unified isolation API that allows to create, combine, and nest various isolation abstractions, including sandboxes, enclaves, confidential virtual machines, and more.

One of Tyche's features is its ability to create enclaves: memory regions that are opaque to the kernel and in which the desired confidential and autonomous processes run. To create and manage these different enclaves, Tyche needs to communicate with the Linux kernel via modules (or drivers): customized pieces of software inserted into the kernel to perform various operations (allocate memory, manage hardware components, implement network protocols, etc.).
One of the main technologies used in this project is the Rust programming language [13]. This language offers many advantages: it's fast, memory-safe, thread-safe, greatly documented [10] and comes with a powerful compiler catching a lot of errors and preventing a lot of crashes or memory leaking.

In order to unify the project and to be able to use the same Rust libraries (or *crates*) in the monitor and in the kernel modules, it would be interesting to know to what extent it is possible to develop modules written in Rust for a Linux kernel originally written in C. The aim of this project is therefore to find out about the use of Rust in the Linux kernel (v6.3.0), to see if it is possible to write a Linux module in Rust and to compare the advantages and disadvantages with the use of C.

What we will see in the first section of this report describes how to write a module in Rust and what are the limitations of the Rust support in the Linux Kernel. The second section explains how to interact with a module in order to control its behaviors from user space. The fifth section is about allocation and mapping of memory in order to create enclaves. The last implementation aspect we're going to go over is how to use external library crates in a Rust module to reuse the same Rust code in many places of the project (section 6). Finally we will compare what are the advantages and drawbacks of using Rust instead of C.

In the remainder of this report, we consider the use of Tyche as a monitor coupled with a Linux kernel [7]. And the version of the Linux kernel will be 6.3.

2

Rust language support in the kernel

First of all, we need to configure the Linux kernel in order to be compatible with Rust utilisation, and set up all the required tools to develop and build Rust programs. Rust has been supported by the kernel since the version 6.1, released in early 2023. The idea of adding Rust support in the kernel is to make possible Rust code addition into the kernel, to benefit from all Rust language advantages. Rust support is available in the Linux kernel under certain conditions. Everything is detailed on the official Linux kernel site [14] and covers the following points:

2.1. Dependencies

- The kernel depends on a specific Rust toolchain. A toolchain is the set of compiler tools used to write and build Rust source files. They generally contain a Rust compiler such as rustc, the dependency manager and cargo construction tool, as well as development tools such as code formatters and documentation generators. For example, the toolchain used for kernel v6.3.0 is the version 1.66.0.
- The Rust source code is necessary because the compiler will cross-compile the alloc and core crates, which is required to use certain functions implemented in their code.
- bindgen [2] is required to generate the bindings needed to call kernel functions. These bindings, generated at build time, are entry points to kernel functions that can generally be called from a C module, but wrapped in a Rust interface so that they can be called from a Rust module.
- ibclang [4] is used by bindgen to understand the C code in the kernel. The kernel must therefore be compiled with clang (or LLVM) to be compatible with bindgen.

2.2. How to activate Rust support within the kernel

In order to make Rust available in the kernel, the build option CONFIG_RUST needs to be enabled. This will enable other options in the General setup menu and add more features for using Rust in the kernel but which are not required in the scope of this project. Once the previous requirements have been validated and the kernel has been built with the right compiler, you can now use Rust in the kernel and start writing modules. Some examples of Rust modules are already present in the kernel in the samples/rust folder.

The following section explains how to write a module in Rust and shows the limitations of Rust support in the Linux kernel.

3

Write a first module in Rust

In this section we provide a simple implementation of a first Linux module written in Rust and show the limitations of the Rust support inside the Linux kernel (v6.3).

3.1. Writing the module

Now that Rust support is enabled, we can start writing and compiling a Rust module.
The following code is a minimal implementation of a "Hello World!" Rust module.

```
// SPDX-License-Identifier: GPL-2.0

//! Rust out-of-tree sample

use kernel::prelude::*;

module! {
    type: HelloWorld,
    name: "hello_world",
    author: "Noe Terrier",
    description: "Rust hello_world module",
    license: "GPL",
}

struct HelloWorld;

impl kernel::Module for HelloWorld {
    fn init(_name: &'static CStr, _module: &'static ThisModule) -> Result<Self> {
        pr_info!("Hello world! (init)\n");

        Ok(HelloWorld {})
    }
}

impl Drop for HelloWorld {
    fn drop(&mut self) {
        pr_info!("Goodbye! (exit)\n");
    }
}
```

The structure of the code is quite similar to a C module.
There is a macro `module!` specifying all the attributes of the driver such as its name, the author, a description, etc. Then there is a struct HelloWorld in which we can put data to be used everywhere in the module. Here it is empty as we don't need to store anything.
The following is an implementation of the `kernel::Module` trait for the struct HelloWorld. To be clear, a trait in Rust is a collection of functions defined for an unknown data type. These functions may already be defined in the trait, and implementing a trait for a type means providing an implementation of all the functions that the trait defines. In this way, the feature `kernel::Module` defines a function

`init(...)`, an implementation of which is provided between lines 18 and 21 of the preceding code. The `init` function is called when the module is inserted into the kernel.

Finally, an implementation of the `drop(...)` function of the `Drop` feature is given. This function will be called when the module is deleted.

3.2. Building

We need two more files: a KBuild file and a Makefile. Makefile are general building files used by Make [8] that can be used for many purposes. Generally, Makefile are used to specify instructions and rules to build big projects or to simplify the building chain, optimizing the build by ignoring the already done building steps. KBuild files are kernel specific build files that are used by the build system of the Linux kernel, which is based on Make, to build kernel objects.

Here, the Kbuild and the Makefile are pretty simple:

Kbuild

```
obj-m := hello_world.o
```

Makefile

```
KDIR ?= /path/to/linux/build

all:
        $(MAKE) -C $(KDIR) M=$(PWD) modules CC=clang
```

The path to the Linux kernel build needs to be specified. The rust code provided inside the kernel and the C code will be bound with the rust module. Using the `make` command will generate a `hello_world.ko` file that can be inserted into the kernel.

Here are some interesting commands that help to manipulate and inspect kernel modules (table 3.1), and an example of use with the previous "Hello World" module (figure 3.1).

`sudo insmod hello_world.ko`	insert the module in the kernel
`sudo rmmod hello_world`	delete module
`sudo modinfo hello_world.ko`	print info about a module file
`sudo lsmod`	list all installed modules
`dmesg`	print the kernel logs

Table 3.1: Useful module commands

```
noe@debian:~/rust-drivers/hello-world$ sudo insmod hello_world.ko
noe@debian:~/rust-drivers/hello-world$ lsmod
Module                  Size  Used by
hello_world            16384  0
noe@debian:~/rust-drivers/hello-world$ dmesg | tail -n 1
[  222.984051] hello_world: Hello world! (init)
noe@debian:~/rust-drivers/hello-world$ sudo rmmod hello_world
noe@debian:~/rust-drivers/hello-world$ dmesg | tail -n 1
[  243.240269] hello_world: Goodbye! (exit)
noe@debian:~/rust-drivers/hello-world$
```

Figure 3.1: Example of module insertion

3.3. Limitations of the Rust support inside the Linux kernel

Writing and building a module works pretty simply. But in fact, what we can do inside the Linux kernel is very limited. For now, the only things that can be implemented are printing into the logs of the kernel and manipulate data or strings. There is no support for implementing more complex and interesting features such as IOCTL, memory allocation, network operations, or other features that Tyche requires. The Linux project wants to improve rust support in the following years. This effort took form into a secondary project called Rust for Linux: a Linux kernel with more Rust features, and the intention to merge the work done in the project into the Linux kernel mainline gradually [12].

The Rust code support provided inside the Rust for Linux project enable the implementation of more interesting functionalities like file operations including IOCTL and memory mapping, semaphores, random number generation, etc.

From now on, the kernel used in this report will refer to the one from the Rust for Linux project, as the one from the mainline doesn't provide enough Rust support, and it is expected that the work done on the Rust for Linux kernel will be moved into the mainline.

4

File operations and IOCTL

One of the most useful features that modules implement are the file operations interface. In Tyche, we need to interact with the driver space to allocate memory and manage enclaves and we can do that through file operations such as `mmap` or `ioctl`. This section lists common used file operations in driver development, explains how to implement them and details how to implement IOCTL.

4.1. File operations

Once a driver is inserted in the kernel, it is associated with a device, which can be a hardware device or not, and this device is identified by a file structure which can be found in the filesystem as a regular file (in /dev/). File operations are used to implement system calls on the device file, like `open`, `read` or `mmap`, which are handled by the driver. The following operations, defined in the Operations trait in rust/kernel/file.rs, are available and equivalent to the file operations defined in the `struct file_operations` structure in a C module.

Rust operations	C operations
`open`	`open`
`release`	`release`
`read`	`read` and `read_iter`
`write`	`write` and `write_iter`
`seek`	`llseek`
`ioctl`	`unlocked_ioctl`
`compat_ioctl`	`compat_ioctl`
`fsync`	`fsync`
`mmap`	`mmap`
`poll`	`poll`

All file operations can be implemented with the Operation trait for the module.

Example:

```
// ...
// begin of the module file
// ...

struct RustFile;

#[vtable]
impl file::Operations for RustFile {
    type Data = Box<Self>;

    fn open(shared: &(), file: &File) -> Result<Box<Self>> {
```

```
        // ...
    }

    fn read(
        data: <Self::Data as ForeignOwnable>::Borrowed<'_>,
        file: &File,
        writer: &mut impl IoBufferWriter,
        offset: u64,
    ) -> Result<usize> {
        // ...
    }

    fn ioctl(
        data: <Self::Data as ForeignOwnable>::Borrowed<'_>,
        file: &File,
        cmd: &mut IoctlCommand,
    ) -> Result<i32> {
        // ...
    }

    fn mmap(this: &Self, file: &File, vma: &mut Area) -> Result {
        // ...
    }

    // ...
    // others file operation implementations
    // ...
}
```

4.2. Input / Output control (IOCTL)

Input/Output control [5] is a syscall usually used for controlling specific operation of a device. One can define a IOCTL more or less when no other syscall fits the scope of the operation we need to define, even if it's not controlling a hardware device. It is used to define specific operations that will be executed on a specific file descriptor given as argument. In Linux, this syscall can be launched with `int ioctl(int fd, unsigned long request, ...)` which takes as argument a file descriptor `fd` of the module file and a numerical code `request` identifying the operation. The functions managing IOCTLs are designed to be able to execute different operations depending on the code they receive as an argument.

In this context, IOCTL management can be defined in the module's file operations, as shown in the code in the previous section. A simple implementation of the ioctl method could be:

```
#[vtable]
impl file::Operations for RustFile {
    type Data = Box<Self>;

    fn ioctl(
        data: <Self::Data as ForeignOwnable>::Borrowed<'_>,
        file: &File,
        cmd: &mut IoctlCommand,
    ) -> Result<i32> {
        cmd.dispatch::<Self>(this, file)
    }
}
```

The code `cmd.dispatch::<Self>(this, file)` transfers the call to a IOCTL handler which will execute the corresponding operation for the given numeric code, present in the IoctlCommand. A handler can be implemented in the same file as the module using an implementation of the trait `IoctlHandler` (defined in `file.rs`). There is an example of such handler:

IoctlHandler example:

```
const IOCTL_PURE_PRINT_VALUE: u32 = 1;

impl IoctlHandler for RustFile {
type Target<'a> = &'a Self;

fn pure(_this: Self::Target<'_>, _file: &File, cmd: u32, arg: usize) -> Result<i32> {
    match cmd {
        IOCTL_PURE_PRINT_VALUE => {
            pr_info!("driver received value {}\n", arg as u32);
            Ok(0)
        }
        _ => Err(EINVAL),
    }
}
```

This handler receives the command code from argument `cmd` and optional arguments in `arg`. Then, if match on `cmd` to find the operation corresponding to the code. Only one code is available here and it use the numerical value of `IOCTL_PURE_PRINT_VALUE` (=1 here), which is a magic number that needs to be known both from the driver and from the user space calling the ioctl syscall.

Another kind of dispatching in this handler is on the *direction bits* of the IOCTL command it receives. Direction bits tell whether the user read or wrote and if the kernel read or wrote and are contained in the `request` parameter and follow modern conventions. Using the macros _IO, _IOR, _IOW or _IOWR, one can create the correct IOCTL request code with correct direction bits. For example, the IOCTL call defined with the _IO macro will be redirected to the `pure` function in the IOCTL handler. But there are other functions available as table 4.2 shows.

IOCTL macro	direction bits set	handler function
`_IO`	_IOC_NONE	`pure`
`_IOR`	_IOC_READ	`read`
`_IOW`	_IOC_WRITE	`write`
`_IOWR`	_IOC_READ|_IOC_WRITE	`read_write`

Table 4.2: Direction bits and corresponding handler functions

IOCTL support is very important as Tyche needs to control its own drivers to triggers custom kernel operations from userspace.

5

Memory allocation and mapping

Memory allocation and memory mapping is needed by Tyche to create enclaves: portions of memory that are opaque to the kernel and belong to a process. This section shows how to allocate memory using the Pages structure and how to map this memory into user space.

5.1. Page allocation

The Rust for linux project implements a simple way to allocate memory using the `Pages` structure and its associated functions defined in pages.rs.
Pages can be allocated by calling the `new` function on `Pages` as in: `Pages::<0>::new()`. The number inside the brackets is called the *order* and it is used to define the size of the allocation. The allocation will have a size of 2^{order} system pages. Then, to allocate only one page, one can call `Pages::<0>::new()` as $2^0 = 1$.

Allocation is done with `alloc_pages` function of the kernel. According to the code and documentation of `alloc_pages`, the allocated pages are continuous and naturally aligned on the order (eg an order-3 allocation will be aligned to a multiple of $8 \times PAGE_SIZE$ bytes) [1]. For now, used page flags are `GFP_KERNEL`, `__GFP_ZERO` and `__GFP_HIGHMEM`. That means that the allocation will be internal to the kernel, zeroed after allocation and be part of the high memory. We can't specify allocation flags when allocating with the `Pages::new` function. These behaviors are compatible with Tyche, which requires pages to be aligned and continuous.

Moreover, the `Pages` structure offers functions for manipulating data of the page in a easy way, as `read` or `write` functions.

5.2. Memory mapping

Memory mapping is a really important concept of system development. Mapping memory means to select portions of virtual memory and associate them to a physical memory region, in a way that operations on the virtual address space apply to physical memory and vice-versa (in practice this mapping is done using translation of the virtual memory addresses into the associated physical addresses so it's the very same physical region).

Memory mapping is used to open files in memory, for inter-process communication (IPC), memory allocation or even memory protection control. Tyche needs mapping in order to give back allocated memory regions to userspace.

In UNIX system, memory mapping is usually done by the use of the `mmap` syscall:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

With, in the order of arguments: Start address in virtual memory, size of the projection, protection flags, file descriptor (of the module file in our case) and the offset in this file. A call to `mmap` will be redirected to the mmap implementation of the module. This `mmap` function is implemented inside the file operations implementation of the module (cf in the code example of section 4.1).

Here is an example of what a `mmap` implementation could be:

```
fn mmap(
        _this: &Self,
        _file: &File,
        vma: &mut Area,
    ) -> Result {
        const PAGE_SIZE : usize = 4096;

        pr_info!("Begin of mmap");
        let size = vma.end() - vma.start();

        // size must be positive
        if size <= 0 {
            pr_err!("End is smaller than start");
            return Err(EINVAL);
        }

        if vma.start() % PAGE_SIZE != 0 || vma.end() % PAGE_SIZE != 0 {
            pr_err!("End or/Start is/are not page-aligned.");
            return Err(EINVAL);
        }

        let page = match Pages::<0>::new() {
            Ok(p) => p,
            Err(e) => return Err(e),
        };

        vma.insert_page(vma.start(), &page)
    }
```

This implementation only allocates one page of memory. The `Area` structure parameter is analogous to (and in fact binded to) the `vm_area_struct` C structure used in C modules. This represents an area of the virtual memory of the process space and can be used to insert allocated pages in it with the function `insert_page`.
After the mapping, all operations done on the mapped memory and all its data will be reflected on all virtual spaces mapped to this region.

As memory allocation is available and can be mapped to userspace to belong to a process, Tyche may be able to manage memory and create enclaves.

6

Using library crates inside a module

The main motivation to use Rust for implementing drivers for Tyche is to reuse code between modules at kernel-level and application at user-level. This can be done using libraries. We will see how Rust provides a way to use libraries, called library crates, in programs and how the Rust-for-Linux project allows use of such crates inside a Rust module.

6.1. Crates specifications

In Rust, crates are defined as compilation units [9]. There are two types of crates: binary crates and library crates. Binary crates are rust files compiled into a binary executable. Library crates are not executable and don't have a `main` function: they only provide code to be used in many other and different programs, with the same intention as the other programming languages using libraries.

Usually, the folder structure of such a library crate is very simple: a `src` folder containing a `lib.rs` rust file containing the code we want to export, and a `Cargo.toml` file specifying the dependencies and how to build the crate.

Figure 6.1: Simple library crate folder structure

By default, Cargo (the build tool used to build crates and manage dependencies) builds this crate using default configuration and output library compiled files that can be used in other crates. But here we are building a module that will be inserted in the kernel and then we need to override the configuration to specify the target, the path to the rust crates in the kernel, the type of object that rustc needs to output and more. This can be done by adding a `config.toml` file in a newly created `.cargo` folder. Cargo will then use the configuration of that file instead of using its default behavior. An example of such a config file is provided in figure 6.2, followed by explanation of the content of this file.

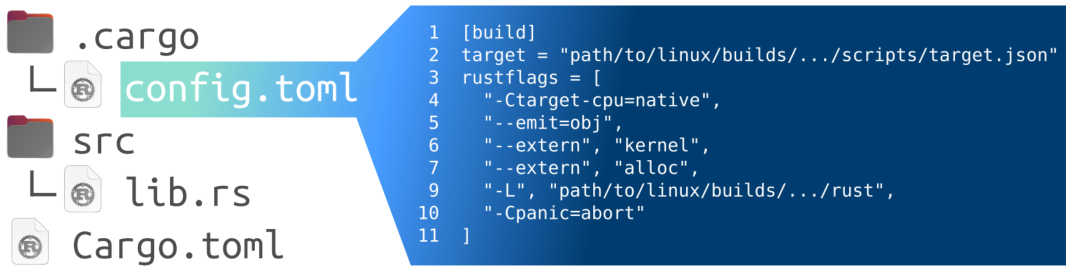


Figure 6.2: config.toml configuration file

The second line of this example file specifies where to find the target.json file inside the kernel. This target.json file specifies the target architecture, llvm-target triple, and other characteristics of the kernel. Next lines are defining rust flags in order for the rust compiler (rustc) to find the rust code in the kernel, what kind of object to output at the end of the compilation (here it's an object file .o), etc. All the rust flag meanings can be found on the official rustc documentation [16].

A last thing that needs to be done before we can be ready to use this crate for a kernel usage is to put the `#![no_std]` crate-level attribute at the beginning of the `lib.rs` file, to avoid linking of the std-crate that is not available inside the kernel, but to link to the core-crate instead.
From the documentation of the embedded rust language [11], one can find the following table 6.3 listing the feature available when std is used or not.

feature	no_std	std
heap (dynamic memory)	*	✓
collections (Vec, BTreeMap, etc)	**	✓
stack overflow protection	✘	✓
runs init code before main	✘	✓
libstd available	✘	✓
libcore available	✓	✓
writing firmware, kernel, or bootloader code	✓	✘

* Only if you use the `alloc` crate and use a suitable allocator like alloc-cortex-m.

** Only if you use the `collections` crate and configure a global default allocator.

** HashMap and HashSet are not available due to a lack of a secure random number generator.

Figure 6.3: features available with no_std or std use

Once the library crate is ready, we can build it by calling the `cargo build` command at the root of the crate folder. This will generate a .o file in `/target/target/debug/deps`.

6.2. Use a library crate inside a module

Now that our crate is built and ready to be used, we can use it in our driver with the following clause, as a regular library crate: `use <library-name>`

But there are still two steps to do before being able to build a .ko file ready to be inserted in the kernel. First we need to update the `Kbuild` file to link with the generated crate build. We specify parameters in the `rustflags-y` variable which will be added to compilation parameter so that rustc can find the compiled target file. And we specify the `module-y` variable to depend on our module and our library crate object files. Note that before, this wasn't required to specify this as the module was constituted of only one file and now it depends on many files. So instead of specifying the object file directly in `obj-m`, we create a `<module_name>-y` variable depending on all other object files, and put `<module_name>.o` in `obj-m`. [6]

Kbuild

```
obj-m := <module_name>.o
rustflags-y := --extern <lib-crate-name> -L $(src)/<lib-crate-folder>/target/target/debug/deps
<module_name>-y := <module_main_file_name>.o <lib-crate-name>.o
```

The second thing to modify is the `Makefile` file. In the following example, line 4 just builds the crate as we did in the previous part. The next line copy-paste the built .o file as a `.o_shipped` file at the root of the module folder tree. `.o_shipped` files tells Kbuild that the content of this object file is external to the kernel. Finally it builds the module as we did before.

Makefile

```
KDIR ?= path/to/linux/build/

build-ko:
	cd <lib-crate-folder> && cargo build
	cp <lib-crate-folder>/target/target/debug/deps/<lib-crate-name>-*.o <lib-crate-name>.o_shipped
	$(MAKE) -C $(KDIR) M=$$PWD CC=clang
```

This results in a .ko module file that can be inserted into the kernel.

This approach allows the reuse of the code of the library in many different Rust modules and userspace applications. It also benefits from the power of cargo to manage, build and optimize code of shared libraries.

7

Comparison with the C language

This section is an overview of the benefits and drawbacks of using Rust instead of C when developing kernel modules.

First of all, Rust and C performances are pretty similar in terms of execution time and are frequently compared[15]. But the size of the produced module in C or Rust can still be relevant to compare.
We compare here the size of the .ko file that a module in C produces with its equivalent written in Rust. The module simply prints a "Hello world" message in the kernel logs. The annex A presents both implementations of the C version and its Rust equivalent which are similar in size too.

The two versions were built using this version of clang and the following toolchain, without debug info:

Clang version:	*Rust toolchain:*
Debian clang version 11.0.1-2	1.66.0-x86_64-unknown-linux-gnu (default)
Target: x86_64-pc-linux-gnu	rustc 1.66.0 (69f9c33d7 2022-12-12)

And here are the results of the size of the produces .ko files:

4448 bytes for the C ko file
4832 bytes for the Rust ko file

The Rust file is a bit heavier than the C one, and we might expect that it will always be the case for more complex modules.

Comparing both languages on the building scope, both use Kbuild and Make to build and in the example of annex A, both module versions can be built with the same Makefile and Kbuild files. But Rust requires the kernel to be built with Clang where usage of C can be more flexible on the compiler used. Rust can also benefit from Cargo when using library crates inside modules for managing dependencies and building complex crates. Rust prevents memory safety violations and provides concurrency safety [13]. It is coupled with a powerful compiler and analyzer that help programmers to prevent crashes or memory errors. This is a really good advantage of Rust over C when developing a project based on trust and confidentiality like Tyche.

8

Conclusion

Currently, it is possible to write Rust Linux modules, but this is very limited by the Rust support of the Linux kernel v6.3. What is available for now is actually a proof of work that Rust can be used inside the kernel. Instead of using the mainline version of the kernel, we can take a look at the efforts to integrate Rust into the kernel and use the Rust for Linux kernel to experiment with Rust integration. The Rust for Linux project centralises progress made in this area and allows us to anticipate the future steps of Rust integration. This project provides a lot of implementations of usual mechanisms like memory allocation, memory mapping, Input-Output control and allows the use of external Rust crates. All these mechanisms are essential to the Tyche project in order to write Rust drivers that can create enclaves and manage them. In the future we can expect the Linux mainline to integrate more and more progress made in Rust for Linux. Then, we will be able to use Rust to implement functionalities of Tyche and benefit from Rust's safety mechanisms, performance, compiler, build tools and reusability of code between the user space application and the kernel module side.

References

[1] *alloc_pages source code and documentation*. URL: https://elixir.bootlin.com/linux/latest/source/mm/mempolicy.c#L2247.

[2] *Bindgen documentation*. URL: https://rust-lang.github.io/rust-bindgen/.

[3] Castes, Charly, Ghosn, Adrien, Kalani, Neelu S., Qian, Yuchen, Kogias, Marios, Payer, Mathias, and Bugnion, Edouard. *2023. Creating Trust by Abolishing Hierarchies. In Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA. ACM, New York, NY, USA, 9 pages.* URL: https://doi.org/10.1145/3593856.3595900.

[4] *Clang project*. URL: https://clang.llvm.org/.

[5] *IOCTL linux kernel documentation*. URL: https://www.kernel.org/doc/html/latest/driver-api/ioctl.html.

[6] *Kernel Kbuild documentation*. URL: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html.

[7] *Linux github repository*. URL: https://github.com/torvalds/linux.

[8] *Make wikipedia page*. URL: https://fr.wikipedia.org/wiki/Make.

[9] *Rust Crates documentation*. URL: https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html.

[10] *Rust documentation*. URL: https://doc.rust-lang.org/book/.

[11] *Rust embedded documentation*. URL: https://docs.rust-embedded.org/book.

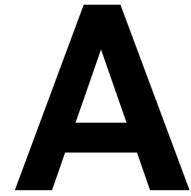
[12] *Rust for Linux wikipedia page*. URL: https://en.wikipedia.org/wiki/Rust%5C_for%5C_Linux.

[13] *Rust programming language*. URL: https://www.rust-lang.org/.

[14] *Rust Quick Start*. URL: https://www.kernel.org/doc/html/latest/rust/quick-start.html.

[15] *Rust versus C clang*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-clang.html.

[16] *Rustc command line arguments*. URL: https://doc.rust-lang.org/rustc/command-line-arguments.html.



Rust and C comparison programs

The following Makefile and Kbuild files can be used to build both of the two programs described below:

Makefile

```
KDIR ?= path/to/linux/build

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules CC=clang

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Kbuild

```
obj-m := hello_world.o
```

The following programs are the ones used as example in this report to compare C and Rust languages in kernel module development. The first is a C version of a "Hello World" program, and the second is its Rust equivalent

C version hello_world.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>


MODULE_LICENSE("GPL");
MODULE_AUTHOR("Noé Terrier");
MODULE_DESCRIPTION("C hello_world module");


static int my_init(void)
{
    printk( KERN_NOTICE "Hello world from C module! (init)\n" );
    return  0;
}

static void my_exit(void)
{
    printk( KERN_NOTICE "Goodbye from C module! (exit)\n" );
    return;
}

module_init(my_init);
module_exit(my_exit);
```

Rust version hello_world.rs

```rust
// SPDX-License-Identifier: GPL-2.0

//! Rust out-of-tree sample

use kernel::prelude::*;

module! {
    type: HelloWorld,
    name: "hello_world",
    author: "Noe Terrier",
    description: "Rust hello_world module",
    license: "GPL",
}

struct HelloWorld;

impl kernel::Module for HelloWorld {
    fn init(_name: &'static CStr, _module: &'static ThisModule) -> Result<Self> {
        pr_info!("Hello world from Rust module! (init)\n");

        Ok(HelloWorld {})
    }
}

impl Drop for HelloWorld {
    fn drop(&mut self) {
        pr_info!("Goodbye from Rust module! (exit)\n");
    }
}
```