Creating Linux kernel modules in Rust

Rewriting a file system in a safe language

Clemens Tiedt[1] and Clara Granzow[2]

[1] Hasso Plattner Institute for Digital Engineering
{clemens.tiedt}@student.hpi.de
[2] Hasso Plattner Institute for Digital Engineering
{clara.granzow}@student.hpi.de

The `rsramfs` project explored the viability of Rust for the development of Linux kernel modules. We found that while the tooling and ecosystem for kernel development are not on par with C yet, Rust has the potential to make it easier to reason about code and increase trust in it.

1 Chances of Rust for kernel development

One of the biggest problems in kernel development are memory safety issues. For example, since 2006, approximately 70% of all security vulnerabilities Microsoft fixed are related to memory safety [1]. Additionally, C as a programming language does not prevent these kinds of vulnerabilities. Over 30% of security vulnerabilities in the most important open-source C projects are because of buffer errors, making this the most common type by a large margin [13].

A promising approach to tackle this problem is the use of a memory-safe language like Rust. Because of safe and unsafe blocks, tricky code is contained to a smaller block and thus easier to review. Run-time garbage collection is replaced by static checks. This means that it does not have to trade memory-safety for speed. [2]

Rust is currently used at a kernel level only in a very limited fashion. There is an operating system, RedoxOS, written entirely in Rust [10]. For Linux, there are currently no modules in Rust with an actual use-case. There are, however, a few experimental kernel modules. [12] [6] [4] [5] [7]

2 Contribution

Since there are currently very few Rust kernel module projects, we want to contribute our own project. Where other modules were mostly proof of concepts, we wanted to create one with actual functionality. In particular, our goal is to take an already existing module and translate it into Rust. This gives us the possibility to write a kernel module that is a little larger while still being able to do this in the span of one semester. The module we chose is the file system ramfs.

3 Theory

Before we go deeper into our implementation details, we want to give a short overview of the theory behind important concepts in our project.

3.1 Filesystems in Linux and ramfs

When we started the project, we only knew that we wanted to write a kernel module in Rust. But we were still unsure what type of kernel module would be the best. Since we agreed that it needed to be relatively self-contained, the choices were somewhat limited. After discarding the idea of a device driver due to the difficulty of interacting with hardware, we settled on a file system, specifically ext2. File systems are interesting and, more importantly, we would have a blueprint in form of the original C-code to work from. Also, ext2 is relatively small compared to more modern file systems.

However, it still proved to be too much for one semester of work. This became apparent the moment we started to actually try to implement it. Because of this realization, we decided to go with a simpler filesystem, ramfs, instead.

What is ramfs? Ramfs is a very simple Linux file system and can be seen as a barebones virtual file system module historically used to implement the `/tmp` directory. The code is pretty short, because most of the work can be done using the existing VFS caches and does not require any hardware interactions.

The Linux kernel provides an abstraction layer above the actual file system – the VFS. This allows the Linux user to choose between different file systems and gives a basic structure that a file system implementation may follow. As such, there are some important objects that an implementation of a file system like ramfs uses.

Firstly, there is the superblock. This is a struct in which the metadata about the mounted file system is stored. The superblock contains the general information about the file system like its type, mount flags and a reference to the superblock operations. These can, but do not have to be, customized by the file system. They include things like `show_options`, which displays the mount options specific to the file system.

Another important object is the index node or inode. This is also a struct and contains all the information about a specific file, but not its actual data. A reference to a list of all inodes can be found in the superblock. An inode contains for example the file's access rights, its size or the time it was last accessed. There are also customizable inode operations like `mknod`, which creates a new file by allocating an inode.

The file system treats a directory like a file that knows a list of files and other directories. When a directory or file is used, a dentry (directory entry) object is created. There exists one dentry object for each part of a path name. A dentry is associated with an inode. There is a dentry cache through which directory access becomes faster. [3]

Ramfs is a file system that is used, as the name implies, to save files on RAM. This works by using the existing caching infrastructure. However, the data is never put in storage. Because no access to the hard drive is required, ramfs is fast. But this also means that its data is temporary. A similar filesystem to ramfs that can be seen as its successor is tmpfs. As such, there is the option to limit tmpfs's filesystem size and memory use. Additionally, tmpfs allows for swapping out pages. Since none of this is possible with ramfs, ramfs poses the danger of running out of memory. [9]

3.2 A short primer on the Rust language

To be able to properly discuss Rust code and concepts, we will now provide a short introduction to some Rust features relevant to our project. An important concept is that of attributes. Rust uses attributes (which are written as `#[attr_name]`) for various purposes. They can be global (in which case they use a shebang instead of a hash symbol) to affect the entire crate (i.e. a Rust package) or attached to pieces of code like a struct or function. They are comparable to annotations or decorators in other languages. Rust offers a form of object-oriented programming using interfaces, here referred to as *Traits*. These make Rust's type system flexible and alow for polymorphism as they allow for restrictions on generic parameters. Some simple trait implementations such as `Copy` and `Clone` which handle moving data implicitly and explicitly can also be automatically generated using the `#[derive(Trait)]` attribute. Probably the most important feature to make writing kernel modules in Rust feasible is its foreign function interface (FFI). Using the FFI, functions from C libraries can be made available in Rust and Rust functions can be exposed with a C ABI. It should be noted here that C declarations allow some features not available in other places in Rust such as variadic functions. Rust also supports raw pointers which give up safety guarantees and behave like C pointers. When more complex data types such as structs are used, they need to be replicated in Rust and marked with the `#[repr(C)]` attribute which ensures that they are laid out in memory the same way they would be in C. Unsurprisingly, Rust-specific types such as enum variants with fields are not FFI-safe.

An important concept in Rust is safety. Generally, Rust can guarantee that any code that is not explicitly unsafe will behave correctly in terms of memory usage, i.e. not produce undefined behaviour (such as memory leaks or double frees). Rust furthermore comes with some features that complement these safety guarantees. Firstly, there are only two ways for a Rust value to contain a null pointer. Either it was explicitly constructed with the `core::ptr::null()` function or it was passed in through an external function via FFI. In fact, only raw pointers can be null. Rust differentiates between references and raw pointers. The compiler statically checks that references are valid and do not outlive the value they reference, whereas raw pointers make no such guarantees. Secondly, all variables are immutable by default. This makes it easy to see where values are changed as the relevant variables must be explicitly mutable. In general, only four kinds of operations are considered

unsafe by Rust. These are dereferencing raw pointers, calling unsafe functions (all functions called via FFI are considered unsafe), accessing static mutable variables, and implementing unsafe traits. Unsafe operations may be performed within an `unsafe {}` block. However, seemingly paradoxically, unsafe blocks are considered safe. The reason for this is that unsafe code is not strictly wrong, the compiler just cannot guarantee its safety - for example not every raw pointer is a null pointer. So, by writing an unsafe block, the programmer steps in to guarantee that the specific instance of a generally unsafe operation is safe in the context. This is important because (to borrow a mathematical term) safe code in Rust forms a closure, i.e. any combination of safe operations will be safe. With unsafe blocks meaning that the programmer guarantees the safety instead of the compiler, we can write safe code that builds on unsafe code.

3.3 Running rust code in kernel space

Getting code in languages other than C to run in the Linux kernel is generally not a trivial task. Firstly, not all languages are fit for this task. For example, the Go language is often named as a competitor to Rust. However, Go requires a runtime for memory management (e.g. garbage collection) making it extremely impractical for use in kernel mode. Even with languages that are better suited to kernel development usually two central issues come up. The first one is that the standard libraries of many languages (including Rust) link against libc which is strictly user mode only. The second is the lack of bindings to kernel interfaces. Thankfully, the first issue is comparatively simple to solve in Rust. Rust has a `#[no_std]` attribute at the crate level to disable linking against the Rust standard library. Without the standard library the core library, the dependency-free basis of the standard library is still usable. This way, many of Rust's language features and types are still available. When writing a crate without the standard library one needs to provide implementations for a few language features, e.g. the behaviour when panicking. Most of these can be implemented just as stubs. With these steps we could already create a library that could be linked into a kernel module. However, seeing as the core library does not provide any memory management, this library was not very useful yet. Specifically, we could not use any heap-allocated data types such as `Box<T>` or most importantly collection types such as Rust's list type `Vec<T>`. Fortunately, using the bundled crate *alloc* it is possible to define custom memory allocation which we use to allocate and deallocate heap memory with the `krealloc` and `kfree` functions respectively. We will discuss how we got access to these functions later, right now we just assume these were imported using an `extern "C"` block. Finally, we wanted the ability to print debug output since Rust's `print!` macro relies on libc. For this we used the implementation provided by souvik1997's example module *kernel-roulette* which uses `kprintf` to redefine Rust's `print` macro.

At this point, we had a Rust library that could theoretically be linked into a kernel module. Linking proved surprisingly simple. Linux kernel modules use `kbuild` which itself uses makefiles. So, we could just write a kernel module using

C that links against our Rust library in the kbuild process and provided an entry and exit point which would respectively hand over control to our Rust code. The Rust compiler can compile code for different platforms using target specifications. Our kernel module target requires a number of special settings such as aborting instead to attempting to unwind the stack in the event of a panic. The target specification from *kernel-roulette* gave us the ability to compile our module for an x86-64 architecture. However, without kernel bindings we could not hope to build any functionality. A naïve approach would be to write the necessary bindings ourselves as `extern "C"` function declarations and `#[repr(C)]` structs, but this way we could not have finished the project in one semester. Another idea would have been to use opaque representations which would have reduced the number of interfaces we needed to copy, but it would also have made checking any safety guarantees a lot harder. What we decided to do instead was to use the tool *bindgen* which can create Rust bindings for a C library. Since this process requires linking against kernel libraries, it is not trivial. Thankfully, *fishinabarrel* already solved this issue for their kernel module. Their solution extracts the required compilation flags from a mock module to then run bindgen using these flags. This causes a few issues mainly caused by Linux using *gcc* whereas bindgen relies on LLVM/clang instead. This restricted us to only being able to create bindings on Linux 4.x due to incompatibilities on kernel version 5.x. We also had to manually edit the generated bindings as bindgen failed to derive the `Copy` trait on some structs.

4 Porting C to unsafe Rust

By now, we had the C source code of ramfs and an empty Rust library capable of being linked into a kernel module. The next step for us was to port this C code to Rust. We did this in two steps. In the first one we rewrote the C code almost as a verbatim translation in Rust. Due to Rust's FFI, we could rewrite our code piece by piece and test that each translated function still behaved the same way. This is the main reason we chose to start with an unsafe implementation. We could have started with a more complex, but safe implementation, but then differentiating between errors caused by behavioural differences between C and Rust and actual defects caused by our implementation would have been significantly more difficult. As many functions from the original ramfs implementation simply call other functions with specific arguments or flags, we started with those. In almost all cases, porting the functions was a matter of translating C types and control structures to the equivalent Rust ones. There were some exceptions, notably `switch` in C behaves differently from Rust's `match` with `match` only matching against literals or patterns where `switch` allows for variables. Another issue that occurred was that on all systems except for Windows Subsystem for Linux 2 assigning a struct field that should contain a pointer to a `const` global instance of another struct caused a panic. While we were unable to determine the root cause, we were able to create a workaround by passing the struct over to the C side of our module and doing the assignment there. At this point we also did not port all functions. Notably, we initially left

out the parsing of mount options as we believed it would be easier to completely rewrite the relevant function using idiomatic Rust from the beginning. We also decided to only support systems with a memory management unit whereas the original ramfs implementation provides code for systems without one. Since we were already limited to x86-64 we considered this a sensible limitation.

4.1 Rewriting in safe Rust

Unfortunately, C-code that has only been blindly translated is not safe Rust code. So far, our code still contains things like dereferencing of raw pointers and directly calling C functions. But since Rust's properties of memory safety are one of the main reasons we want to use this language in the first place, we need to fix this – the unsafe code must become safe.

But before going into the details of how we did this, it is helpful to understand the overall structure our project had after completing this process. In the end, we had seven files of actual code; six Rust files and one C file. They can be arranged into two different groupings of three and two files, leaving two files separate. The

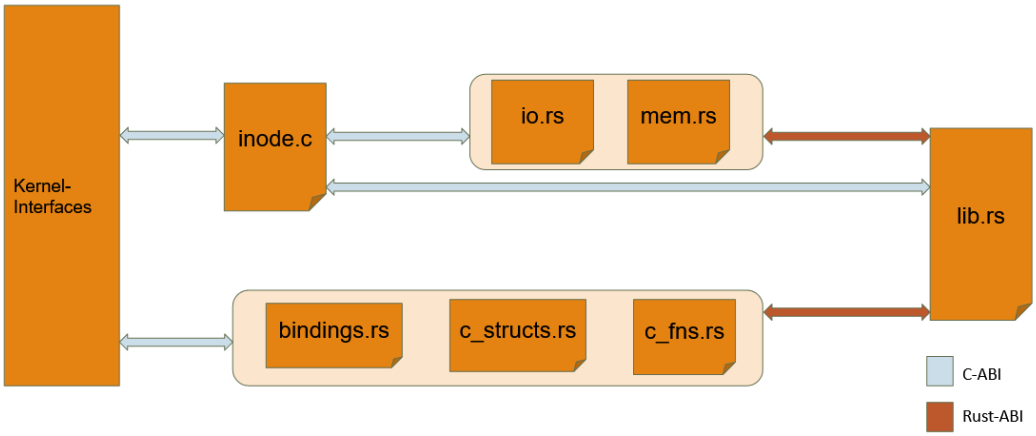

Figure 1: The architecture of rsramfs

first of those separate files is inode.c. This forms the starting point of the module. In essence, this file is just a copy of the inode.c-file from the original C-version of the file system, but with everything taken out that we moved elsewhere. What remains is what is needed to initialize the module. Additionally, the file contains some C-code that would be hard to call via FFI, like inline functions and macros.

Most of the actual functionality of the file system is contained in lib.rs, the second of the stand-alone files and the root of our Rust library. The implementations for the file system's functions can be found here. To be more specific, the file contains the implementations for `ramfs_get_inode`, `ramfs_fill_super` and `ramfs_mknod`, among others. Those functions can be called via the C-ABI, as if they were C-functions, by

the operating system. Aside from those functions, there are a few helper functions, mainly to provide Rust wrappers.

But that is not the only place where we can find Rust wrappers. The first grouping of several files, consisting of bindings.rs, c_structs.rs and c_fns.rs, can be summarized as providing Rust wrappers for kernel interfaces.

Bindings.rs is a file that is automatically generated by the tool bindgen, which generates Rust FFI bindings to C libraries [11]. This allows us the use of the C libraries used in the original implementation of ramfs without having to actually write the bindings by hand.

In c_fns.rs, Rust wrappers are put around the functions from bindings.rs for which this was necessary.

C_structs.rs contain the wrappers around certain C-structs, like inode. The process for this was slightly challenging and will be explained in more detail in a later paragraph.

The second file grouping contains mem.rs and io.rs, which, as is obvious from the name, are responsible for memory allocation and input-output. If we want to do things like printing or allocating pointers on the heap (which is required by certain data types) from our Rust functions, we need to write this functionality ourselves based on the kernel versions of these mechanics. Fortunately, we could take this from previous Rust kernel modules [4].

Now we can finally tackle the actual question – how do we make unsafe code safe?

It is not possible for us to get rid of every unsafe line of code. We have to do those unsafe operations *somewhere*. But what we can do is hide them.

Dealing with a function call of a C function is relatively straightforward. All we need to do is put a wrapper function around the unsafe foreign function call – provided we are sure the code we want to wrap is actually safe. Since Linux kernel code is written with a high code quality in mind, we do not have to worry about this.

```
pub fn rs_kill_litter_super(sb: SuperBlock) {
    unsafe { kill_litter_super(sb.get_ptr()) };
}
```

There are some naming conventions we kept to: Our wrapper functions are called 'rs_function_name', with 'function_name' being the original name of the C function. There are some places where it was beneficial to change the return type to one of the Rust-specific types Option and Result, which facilitate error handling. We use an Option when the original return type contains a pointer: It returns `Some(element)` if element is not null and `None` if it wasn't. This prevents us from creating wrappers containing null pointers.

```
pub fn from_ptr(sb: *mut super_block) -> Option<Self> {
    if sb == core::ptr::null_mut() {
        None
    } else {
        Some(Self { ptr: sb })
```

```
        }
    }
```

If the C-function may return an integer that represents an error code, we use a Result. Either `Ok(())`, which can be seen as equivalent to returning `0`, or `Err(error)` is returned.

```
fn rs_ramfs_mknod(
    dir: Inode,
    dentry: *mut dentry,
    mode: umode_t,
    dev: dev_t,
) -> Result<(), cty::c_int> {
    use bindings::ENOSPC;
    use c_fns::{rs_d_instantiate, rs_dget};

    match rs_ramfs_get_inode(dir.get_sb(), dir, mode, dev) {
        Some(inode) => {
            rs_d_instantiate(dentry, inode);
            rs_dget(dentry);
            dir.set_mctime_current();
            Ok(())
        }
        None => Err(-(ENOSPC as i32)),
    }
}
```

Thus, we have created a safe interface that we can now interact with like we would with any other safe rust code. If we want to use any of these functions, we will not have to worry about safety anymore.

This whole process is simple enough that we attempted to automate it. For that purpose, we started to write a tool, wrapgen [14], that would do this. It's still rudimentary and has a lot of issues. But because we only had the idea for wrapgen after we already finished creating most of the wrappers by hand, and because we prioritized our main project, we put its development aside for now.

There are, of course, a lot of places where we have to deal with more than just function call but need to use a struct from one of the C libraries instead. We mainly needed to access various fields from inode and superblock. Dealing with structs proved to be a little trickier than dealing with functions. The problem were the raw pointers, since accessing them is almost always unsafe.

Our first idea to solve this problem, defining a trait that would dereference a `*mut c_type` to a `c_type`, was unfortunately impossible. This solution would have violated Rust's ownership rules and we would have needed pointers for the C-interface anyway.

What we did instead was to create lightweight wrapper structs. The only field they have is for the pointer we want to wrap. The pointer is of the type of the struct we want to wrap, for example inode.

```
#[derive(Copy, Clone)]
pub struct Inode {
    ptr: *mut inode,
}
```

If we want to access this struct, we have to do it via an associated function, like `get_ptr` and `from_ptr`, which both Inode and SuperBlock have. In addition, Inode has two special constructors, `new` to create a new inode using the `new_inode` function and `null` to create one containing a null pointer. We need some associated functions to access certain fields of the struct we point to, like `set_ino`, and functions that have the struct as pointer as an argument need to be added as associated functions as well.

```
impl Inode {
    pub fn new(sb: SuperBlock) -> Option<Self> {
        Self::from_ptr(rs_new_inode(sb))
    }

    pub fn null() -> Self {
        Self {
            ptr: core::ptr::null_mut(),
        }
    }

    pub fn from_ptr(inode: *mut inode) -> Option<Self> {
        if inode == core::ptr::null_mut() {
            None
        } else {
            Some(Self { ptr: inode })
        }
    }

    pub fn get_ptr(self) -> *mut inode {
        self.ptr
    }


    pub fn set_ino(&self) {
        unsafe { (*self.ptr).i_ino = get_next_ino().into() }
    }
    .
    .
    .
}
```

Finally, there were a few things we added to the structs ourselves, like the RamfsSuperBlockOpts, that gives us the possibility to add a custom debug mode. They were added as an extension trait.

```
pub trait RamfsSuperBlockOpts {
    fn is_in_debug_mode(&self) -> bool;
}
```

```
impl RamfsSuperBlockOpts for SuperBlock {
    fn is_in_debug_mode(&self) -> bool {
        unsafe { (*((*self.ptr).s_fs_info as *mut RamfsFsInfo)).mount_opts.debug}
    }
}
```

After we finished the rewrite to safe Rust, our project ended.

5 Possible future work

5.1 Future work on rsramfs

At this point, rsramfs is feature-complete. There are however some remaining issues we would like to solve. We needed to split off a version that only ran in the Windows Subsystem for Linux 2 kernel because due to what we can only assume is a memory management bug assigning a reference to a global struct created in the Rust code causes a kernel panic. Other than that, we would like to separate the kernel bindings from our filesystem code. For a small project such as this one, it made sense to keep these in one crate, but the bindings and wrappers could be useful to other projects as well. By splitting them off into their own crate, they could be used by different projects and updated or expanded independently from our project. Beyond that, there are only some code style changes that might be useful. For example, the wrappers could be unified into a `Wrapper<T>` type that comes with functions to create one from a pointer of type `*mut T` and accessing that pointer when interacting with C code. We did not make this change in our project because the little code duplication was not an issue for us and other issues took priority.

Our project focussed mainly on making C interfaces easily accessible through Rust code. Future work could also consider making it easier to export Rust functions into C code. The ones we expose take raw pointers as arguments and need to construct wrappers from them manually. It would be easier for programmers to be able to write their function using Rust semantics and automatically generate an `extern "C"` function that takes raw pointers for each wrapped argument of the Rust function and converts them. This would have taken too much time during our project, but it is achievable using Rust's procedural macros. In fact, we built a prototype that given a function that takes some arguments of type `Wrapper<T>` can create a function that instead takes arguments of type `*mut T` and generates the required wrappers. However, we believe that this tool requires more development before it could actually be used productively.

Another concern here is performance. In general, we noticed no performance differences between the original ramfs implementation and rsramfs. However, our wrappers are not completely free of performance overhead. Almost every action is

wrapped in a function call and while this may not be an issue with smaller tasks (such as writing or reading short text files), the overhead might become noticable in larger or more complex tasks. There is some performance overhead we cannot get rid of, e.g. conditionals for null checking. We believe that by inlining function wrappers we could get rid of the bulk of the overhead. With this overhead gone, we believe that a larger kernel module using wrappers such as ours could actually be faster than a C one as e.g. null checks would only have to happen at the creation of a value instead of every time it is accessed.

5.2 Future work on other kernel modules

We see great potential in using Rust to create or rewrite more kernel drivers. Given the nature of ramfs our project had virtually no direct interactions with physical devices. Therefore we are very interested in seeing which challenges and opportunities arise when using Rust to create a proper device driver. Firstly, it could be interesting to build upon the abstractions we already wrote to create or recreate a file system that operates on an actual disk. Another project could be to write a driver for some sort of peripheral (e.g. PCIe) device. This should provide some insight into how well interacting with devices at a low level, e.g. having to use precise timings in communication protocols, works in Rust. Given that Rust is also used in embedded systems we have high hopes in this regard.

Another aspect our project did not explore is testing. Generally, testing kernel modules is not trivial and we resorted to a kind of simple integration test. However, kernel testing frameworks such as *KUnit* exist and it should be possible to create Rust wrappers around them. Rust's built-in testing support also works in a `#[no_std]` environment and we are eager to see how further work in this area can help make the functionality of Rust kernel modules more verifiable.

5.3 Possible official Rust support in the Linux kernel

Rust is still a relatively new programming language. Still, when we started the project, there were several other projects on GitHub for a Rust module for the Linux kernel [12] [6] [4] [5]. None of them are very big and all of them definitely experimental, but it is clear that there is some interest in the topic. Of particular note is a project that is attempting to create a framework for writing a Linux kernel module in Rust. It's authors are actively pushing the topic of using Rust in the Linux kernel into more official spaces: As a step in this direction, they presented their project on the Linux Security Summit North America 2019 [7].

But despite these valuable efforts, every developer that wants to delve into projects like this at the current moment in time will inevitably run into the problems that come with the lack of support.

This summer, however, offered a spark of hope that such a thing is not that far off into the future. Linus Torvalds himself stated in an interview that he could see Rust being used in the Linux kernel in the future – the kernel team was already looking

into having interfaces to write things like drivers in Rust. On an exchange on the Linux kernel mailing list, he sounded a little more cautious but still ultimately receptive to the idea. On the Rust side of things, Josh Triplett, Rust language team leader, seems open for collaborations and enhancing the Rust language itself in order to facilitate kernel integration. [8]

This culminated in a discussion about in-tree Rust support at this year's Linux Plumber's Conference at the LLVM microconference. All in all, there seems to be a lot of enthusiasm for the idea, but there are still many problems that need to be solved. For example, when creating bindings with bindgen, macros and inline functions are currently not supported very well – a problem that we encountered in our project as well. Related to this is the issue of needing to write a large amount of wrappers by hand, which we also noticed during our project. On a more abstract note, since the only currently mature Rust compiler, rustc, uses a LLVM backend, there are support questions with kernel architectures that don't have LLVM, as well as possible ABI compatibility issues with a kernel built with gcc. [2]

All in all, it does not seem like there is going to be any actual official support anytime soon. However, we are confident that Rust support in the Linux kernel (be it official or unofficial) will improve in the coming years.

5.4 Conclusion

Writing a Linux kernel module in Rust is, at the current moment in time, viable but challenging. In our project, we successfully created such a module by translating the existing module ramfs from C to Rust. This proved that while there is no official Rust support in the Linux kernel so far, Rust is already usable in this context and may become a viable option for productive use in the future.

There are, however, some unresolved problems, particularly the need to split off a version for Windows Subsystem for Linux 2 kernel, whose root cause remains uncertain.

There were also some issues relating to the lack of support of Rust in the Linux kernel. There exist no official bindings to kernel interfaces, and while bindgen is a very useful tool to circumvent this, it cannot help with inline functions and macros, and the resulting kernel interfaces need to be manually wrapped. This is what we spent most of our work time on, but could (and probably should) be at least partially automated to ensure that future projects can focus on the actual work to be done. The further development of the tool wrapgen we started for this purpose is a possible avenue to achieve this.

References

[1] *A proactive approach to more secure code*. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`. Online; accessed 29-September-2020.

[2] *Barriers to in-tree Rust*. https://www.youtube.com/watch?v=FFjV9f_Ub9o. Online; accessed 29-September-2020.

[3] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. Sebastopol: O'Reilly, 2006.

[4] *kernel roulette*. https://github.com/souvik1997/kernel-roulette. Online; accessed 29-September-2020.

[5] *kmod*. https://github.com/saschagrunert/kmod. Online; accessed 29-September-2020.

[6] *linux kernel module rust*. https://github.com/lizhuohua/linux-kernel-module-rust. Online; accessed 29-September-2020.

[7] *linux kernel module rust*. https://github.com/fishinabarrel/linux-kernel-module-rust. Online; accessed 29-September-2020.

[8] *Programming languages: Now Rust project looks for a way into the Linux kernel*. https://www.zdnet.com/article/programming-languages-now-rust-project-looks-for-a-way-into-the-linux-kernel/. Online; accessed 29-September-2020.

[9] *Ramfs, rootfs and initramfs*. https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html. Online; accessed 29-September-2020.

[10] *RedoxOS*. https://gitlab.redox-os.org/redox-os. Online; accessed 29-September-2020.

[11] *rust bindgen*. https://github.com/rust-lang/rust-bindgen. Online; accessed 29-September-2020.

[12] *rust.ko*. https://github.com/tsgates/rust.ko. Online; accessed 29-September-2020.

[13] *What are the most secure programming languages*. https://resources.whitesourcesoftware.com/research-reports/what-are-the-most-secure-programming-languages. Online; accessed 29-September-2020.

[14] *wrapgen*. https://github.com/ctiedt/wrapgen. Online; accessed 30-September-2020.