CS 5264 Project Final Report

Austin Minor
acminor@vt.edu

and

Connor Shugg
cwshugg@vt.edu

December 1, 2021

1 Introduction

RamFS Rust is the culmination of our work for Project 4. Rust and Linux has become a surprising match as more companies are starting to invest in Rust. Rust is seen as a better C. It is a systems programming language that can interact with low-level hardware while also giving programmers the ability to use high-level constructs such as iterators and functional programming idioms. Furthermore, unlike C++, it does this under two main constraints. First, it improves the security of the code in question. Rust accomplishes this through a complex, but compile-time, system of object lifetime and ownership. This system allows for Rust to assert that memory leaks are not occurring and that memory will not be double freed. It even allows for some guarantees on the sharing of an object across parallel threads. Second, it sticks close to C instead of going with a system like C++, namely object-oriented programming. Instead, Rust sticks with a trait oriented programming style.Instead, one defines traits, you can think of these as interfaces, then types implement said traits. Then, at compile time, the functions are monomorphized to use the proper functions for a given type, if using generics. This allows us to keep a similar style to C with types holding data while also giving types the ability to have interfaces and functions. The actual implementation details of such a system and the exact influence on programming style such systems have upon programming thought are complex and beyond the scope of our project.

With the security of Rust as a language and its unique high-level concepts, companies are very interested in applying it to critical infrastructure. We will delve more into the reasons for wanting more secure languages later. Some quick examples of Rust users are Microsoft[1], Google and Android[2][3], Pop OS[4], among others. Even Linus Torvalds, a quite picky man, has said that Rust in the kernel looks promising [5]. Needless to say, Rust and kernel programming seems to be gaining recognition of being a possibility. Current work has focused on the periphery, looking at kernel modules and device drivers[6]. This focus on the periphery versus internal kernel systems has been debated by some in the kernel development community[7].

For our project, we decided to address this concern and see what a kernel system written in Rust might pose. For this we chose RamFS. RamFS in the source code is self-described as a trivial file system and an example to base more complex file systems off of[8]. Furthermore, it is not a module. As mentioned in the documentation, it is too small to care about compiling out [9]. Thus, we have a minimal viable product (MVP) for implementing internal kernel functionality in Rust. Furthermore, RamFS might be used in the actual kernel boot sequence[9] and from our usage of kgdb appears to be used in our Linux build configuration. This gives weight to our implementation as it will be used for core kernel functionality. We did not choose an MVP on the periphery. Instead, we chose one on the inside, one that is used to accomplish real work while also being small enough to handle. Overall, we implemented RamFS in Rust and contributed some progress towards making the code more Rust-like ("rustification").

2 Background and Motivation

The C programming language has long been the top choice for developers of the Linux kernel. Thousands of raw memory accesses require careful programming practices. While most memory-related bugs are caught in development, some slip through the cracks. As such, the Rust programming language has increased in popularity for kernel developers over the past few years thanks to its memory safety guarantees provided by Rust's borrow checker and type system[10].

Because of the world's dependence on the Linux kernel, any memory-related vulnerabilities that lie within may result in exploits that bring down services millions of people depend on. Furthermore, memory safety bugs are the largest category of security vulnerabilities. A team of Microsoft security engineers have stressed that seventy percent of vulnerabilities found in Microsoft products were due to memory safety bugs[11].

In our previous reports, we cited some statistics on the number of CVEs filed in the Linux kernel in 2019, 2020, and 2021. In 2019, 265 were filed. In 2020, 117 were filed. Presently in 2021, the number sits at 151, up from the initial 136 at the time of our project proposal submission and the 143 at the time of our mid-point report submission. Several of these new CVEs involve memory corruption, a couple of which are marked as high-priority bugs[12]. To this day, memory safety is still an issue within the Linux kernel.

This is why Rust is so appealing. Rust's unique memory safety guarantees makes it an enticing language to write parts of the Linux kernel. It's also our motivation for this project. By implementing RamFS in Rust, we hope to forward the progress of the Rust for Linux project by providing a working implementation of a simple file system.

2.1 RamFS

RamFS is short for RAM File System (RAM standing for Random Access Memory). It's a simple file system used by the Linux kernel existing entirely in a computer's memory. It utilizes the pre-existing Linux disk-caching infrastructure to perform most operations, which is why the code base is small compared to other file systems[9].

RamFS plays a small role in the Linux boot process before the kernel's main file system is mounted when the `init` process takes over. Practical uses also exist after the kernel has booted. Creating a temporary RamFS mount and pointing a user-space program at it to perform its reads and writes allows for much faster access times due to RamFS's freedom from having to communicate with the hard drive.

2.2 The Rust Compiler

The Rust language has its own dedicated compiler based on LLVM and thus lacks the wide support of architectures provided by GCC[13]. Fortunately, the Rust compiler has an API that allows for custom code generators to be plugged in. The `rustc_codegen_gcc` project is one such custom code generator that utilizes GCC's support for several architectures *not* supported by LLVM[14]. While we did not make use of this during our project and it is currently marked as a work in progress[14], this is a promising route to fully support other architectures.

2.3 Related Work

This project builds upon the foundation of the Rust for Linux project[15]. It was created to utilize Rust as a language for kernel development. It consists of `make` rules to compile Rust source code properly, Rust binding generation from C source files, rusty interfaces to some kernel functionality, and other Rust specific functionality such as a Rust allocator based on `kmalloc` and macros for developing kernel modules[15].

Of course, RamFS is not a kernel module[9], and thus does not use the Rust kernel module interface provided by the Rust for Linux project. Because of this, we were in relatively untested waters. As such, we are relying heavily on the base that Rust for Linux has built their abstractions on. Mainly, Rust as the language itself and the Rust bindings generated from kernel source code.

As the kernel is not in a typical bootstrapped environment, we have some related work to lean on in terms of The Rustonomicon. This is a guide for writing unsafe Rust code, something that isn't often recommended but is quite necessary for kernel development, embedded systems, and other low-level software development[16]. Along with The Rustonomicon, we can also reference the official Rust documentation which goes into particular detail about the Rust language itself[17].

3 Goals and Milestones

The end goal at this project's inception was to complete a full, working port of the RamFS file system from C to Rust. In our project proposal, we introduced the following milestones:

%Goal	Deliverable
75%	25% of the functionality ported.
100%	50% of the functionality ported.
125%	100% of the functionality ported.
150%	Begin to make the code more idiomatic Rust.

Before embarking on the development process, we were unsure as to how easy or difficult this port might prove to be. Writing standard Rust code is one thing, but writing a kernel-level port of a file system is another beast entirely. Our 125% end goal was a complete port of all RamFS C code. An extra 150% goal was tacked onto the end in the event that porting the Rust code left us with lots of extra time.

Unfortunately our third group member had to drop the class for personal reasons early in the project's life, so these initial plans were modified slightly. To adjust for the lack of a third group member, we modified our agile-like development strategy to work without a scrum-master (our third group member's role) by using JetBrains Space's built-in issue boards, document storage areas, and pull-request code review support. We also adjusted our development goals to cover the 99% use-case of RamFS by *not* porting the RamFS code used when a system lacks an MMU (Memory Management Unit). We felt this change in development goals was acceptable, as most modern computers include an MMU and thus will never make use of the code found in RamFS's `file-nommu.c` source file.

4 Implementation Results

We are happy to report that porting RamFS to Rust has been successful. Excluding the non-MMU RamFS code, we have successfully ported every function and almost every C structure to Rust code, achieving our 75%, 100%, and 125% goals. We have also had the time to dip into our 150% milestone and fine-tune our initial port to use more rusty code.

4.1 RamFS Port

Overall, we achieved success in our RamFS port porting all the desired functionality with small stubs to call kernel function and one global object that was left in C. We had many ups and downs; however, the general process was less difficult than we initially thought. Rust is well-known for being a difficult language to work with. It's complex system of lifetimes and ownership can result in difficult to figure out errors and issues. However, as we are frequently interacting with C, we loose lifetimes across code boundaries. This results in our code not currently having to deal with lifetime

issues. This is compounded by Rust's continual improvement in solving lifetimes implicitly[18][19]. We will address our lifetime comment in the section on future work. Needless to say, our time porting, though met with issues, was not as perilous as initially thought and described in our initial proposal.

Before beginning a discussion of technical details concerning the port and things that we learned, we first must describe our porting process. We chose to do an incremental port. This is key for a few reasons. One, we could test the system as if it is working. If we started from scratch, we could only test parts of the system and even testing these parts would be difficult because the code is kernel-level code. Two, we could perform advanced integration tests as we ported each individual function. As mentioned, RamFS is used in the kernel boot process. Any major bugs might break such a process. Thus, each run and test of our kernel, not to mention our own test suite, further proved the correctness of our code. Three, we could provide a pathway and knowledge base to incrementally port other kernel systems.

The last point about a knowledge base is the most important part. Most developers might be skeptical of a completely new re-implementation of a system in Rust. However, by gradually porting, one could show that the system remained stable under a small set of changes between each release, relying on less and less of a C backbone as the port progressed in usage and porting progress. This is compounded by some kernel developers thinking that a subsystem of Linux is an interesting challenge for a Rust port to take on due to the interaction between the kernel's subsystems and any problems that may arise from this[20][7]. Our project shows that this incremental port process is possible and can be intermixed with C and Rust changes if need be.

In general our process hinged on two main things. First, a compilation of functions to be ported in the form of an issue list. This was handled using JetBrains Space and allowed us to better coordinated our work and see our progress. Second, additional Makefile functionality to handle generating C header files from Rust for use by kernel C. This process used CBindgen which will be discussed in further detail below. We accomplished this by creating a new implicit rule in `scripts/Makefile.build` to convert from Rust files to C header files. Luckily, there was plenty of prior work to show how to do this in the kernel code. Currently, these headers must be ignored separately in a `gitignore`; however, if future community consensus on a naming convention prevailed, we could potentially use a naming pattern to programmatically ignore these generated headers.

As mentioned, CBindgen is a program for generating C header files from Rust source code[21]. This formed a crucial part of our porting effort as it allowed us to port a function to Rust and then use that function in the original C source code. For the most part, this served more of a porting functionality. However, if only part of a system is ported to Rust or that system needs to be accessed from other subsystems, our work on generating headers could be very useful to these scenarios. Although we used it successfully, CBindgen is not without issues that we had to develop workarounds for. There were two main issues that we found in this process. First, CBindgen was made with the assumption that Rust projects used Cargo for building[22]. Linux does not use Cargo for building instead using custom logic built into KBuild leveraging Make[23]. As such, CBindgen had issues understanding the types of kernel structs we used in our code. To work around this issue, we leveraged a quirk with CBindgen that it does not currently understand Rust namespaces[24]. Thus, we can create a namespace within our source file and redeclare any types as their necessary overarching type such as a union or struct. Without this redeclaration, CBindgen used `C++` style structs which do not distinguish having or not having the prefixed struct keyword. This is necessary as in C `struct type` is different from `type` and same with `union`. A better future solution to this would be to extend CBindgen to handle different types of projects. Second, CBindgen requires certain tricks to export a struct that is not used externally. We handled this be creating a dummy function that exists only to export said struct types. Overall, these are issues that hopefully will get fixed in later releases of CBindgen. The Rust for Linux project does not have support for CBindgen; so, we had to add support. Hopefully with our support, the Rust for Linux community will help improve CBindgen over time.

With CBindgen and our list of issues, it was merely a process of selecting a function, porting it, and testing it. The nice thing about our process is that for the most part this could be done independent of other teammates as our ultimate interface was the C ABI not whether the functions or objects being compiled to the C ABI were in Rust or C. For the most part, this process of porting was uneventful. Most of it relied on the kernel bindings generated by Bindgen (a program for generating Rust bindings to C code[25]). This was already set up by the Rust for Linux community. From these bindings, most of our code consisted of using unsafe code on the pointers we received or converting pointers to Rust references. As we progressed, we learned more and more about how to use the least amount of unsafe code possible. For example, checking if a pointer is equal to the null pointer can be done in safe Rust code. We will touch on some of the areas of potential improvement to our port in the future work section. Besides this, some bits of code had to be left in C. There are multiple reasons for this. One reason is that a function is marked as inline and might not have a compiled symbol in the kernel binary. This was the case for our usage of `dget` which was marked both as inline and static. Other reasons we needed code to remain in C was unexported bindings and macros. The Rust for Linux project only exports a selection of bindings for the Linux kernel. This is controlled using the `rust/kernel/bindings_helper.h`. Some of our bindings such as `get_tree_nodev` were not in the default bindings. Given more time, more validation could be done on the benefit/cost of generating the rest of our required bindings. For now, except for `linux/seq_file.h` which we will mention later, we did not add to the bindings generated. In regard to macros, we have not ported the `fs_initcall` macro to Rust. This macro generates assembly language code for initializing a file system. Currently, we have left this in C as future work. If more users want to use Rust as a language for writing file systems, this macro could theoretically be ported as Rust has an assembly macro, `asm!`. This feature is not yet stabilized but should be possible to use[26][27].

On the whole, one of our most difficult issues in porting was porting global objects to Rust. There are a couple of reasons for this. One such issue is Rust's safety for multithreaded programming implying that certain types are not assumed to be thread-safe which is required by certain Rust static objects. This can be worked around by implementing an unsafe trait[28](we only needed the `Sync` trait); however, that pushes the goal of validation onto the programmer. Another issue is that Rust global object fields must be determined at compile-time. This led to different issues such as not being able to use `Default` trait constructed objects or having some strange compiling issues were certain unsafe linkages to C global objects would sometimes work and sometimes not (this might have been a compiler bug, we are unsure). The final issue we had is that unlike C, Rust does not assume that the rest of its fields are zeroed. This would go against a Rust guarantee that references are never zero. We are unsure why Rust does not implement a compile-time trait that would give zeroed defaults or compile-time error if a zeroed default is attempted to be used on a type that cannot have a default such as references. With these constraints, at first, we had to manually set all the remaining fields of a struct to their equivalent C zero values. This was a tedious process especially considering that certain parts of the object are KBuild configuration defined. To get around this, we wrote a macro that would default initialize a zeroed-struct (C-style default struct) at compile-time. We attempted to see if this could be made a function in the Rust standard library; however, it appears to be blocked by the unstable feature of `const-generics`[29]. The macro simply creates an array of zero-bytes the same size as the object being constructed. This is then cast using`core::mem::transmute` to the desired type. `core::mem::transmute` checks to make sure the two types are actually the same size and thus are castable to each other[30]. These operations can all be done at compile-time and allow us to use Rust syntactic sugar that fills the rest of a struct with the contents of another struct. An example of this can be seen in Listing 1.

During our initial porting to Rust, we noticed that one of our functions relied on sequence files to print output to a `proc` file. This originally was handled using a C-style `seq_printf` statement and a helper function in C. However, we desired to be able to use Rust-style format strings. An example of this is `println!("Hello, {}", "world")`. Initially, we tried to use Rust's own infrastructure only to find out certain parts were missing in the implementation for `core::fmt`. We believed this

```rust
#[no_mangle]
static mut ramfs_fs_type: file_system_type = file_system_type {
    name: c_str!("ramfs").as_char_ptr(),
    init_fs_context: Some(ramfs_init_fs_context),
    parameters: ramfs_fs_parameters.as_ptr(),
    kill_sb: Some(ramfs_kill_sb),
    fs_flags: RAMFS_RUST_FS_USERNS_MOUNT,
    ..c_default_struct!(file_system_type)
}
```

Listing 1: `c_default_struct!` usage

```rust
/* Original code */
unsafe {
    ramfs_rust_seq_puts_mode(m, c_str!(",mode=%o").as_char_ptr(), mode);
}
/* seq_printf! macro code */
seq_printf!(unsafe{ m.as_mut().unwrap() }, ",mode={:o}", mode);
```

Listing 2: Old-style and new-style sequence file printing

to be the end for this functionality because implementing this infrastructure would be out of the scope of what we could accomplish in our project timeline. However, we later discovered that Rust for Linux had implemented the ability to use Rust format strings for `printk`. With this knowledge, we were able to track down and determine that a new `printf`-style specifier was added by Gary Guo in commit `9e8bd679`[31] to handle Rust format strings. This specifier was `%pA`. With this, we were able to create a new macro based off of the Rust `printk` macros in `rust/kernel/print.rs` to handle printing to `seq_file` objects. An example of the transformation this can bring can be seen in Listing 2.

4.2 RamFS Testing

To test our incremental changes from C to Rust, we wrote a testing suite in Python and Bash to perform simple operations on a RamFS mount within a QEMU instance or virtual machine running the modified kernel. As we ported each function, a corresponding test was written, if possible, to ensure its high-level correctness. This testing suite allowed us to catch a few small bugs in our code during development.

In addition to writing our own testing suite, we found and utilized IOzone, an extensive and easy-to-use file system test suite[32]. While its main purpose is to benchmark the performance of Linux file systems, it also serves well as a way to stress-test our RamFS implementation. After running this testing suite several times, we found no issues and comparable performance to the original C implementation. Seeing as performance wasn't our main goal with this project, we did not spend much time comparing benchmarking results between the C and Rust implementations. This is something we leave as future work. Results concerning this benchmarking can be found in our submitted tarball.

Furthermore, some initial testing was done using the Linux Test Project test suite[33]. However, we determined after our mid-point review that this test suite did not seem to be well suited for our problem. Despite this, the results still stand that our implementation seemed to hold up under sustained load tallying multi-gigabytes of data.

```
S_IFREG => {
    inode.i_op = unsafe { &ramfs_file_inode_operations };
    inode.__bindgen_anon_3.i_fop = unsafe { &ramfs_file_operations };
}
```

Listing 3: Bindgen anonymous structs and unions example

5 Future Work

We will group our future work into two sections: completing the port and making it more "rusty". As we have completed most of the port, our future work mainly focuses on how to make our RamFS port more "rusty".

5.1 Completing the Work

Other than `file-nommu.c`, we have implemented most of our functionality in Rust. The remaining functionality is blocked by either a C macro remaining to be ported into Rust or inline/non-exported functions/types in the kernel bindings. Both of these points have already been discussed. In regard to the future work aspect of these points, the C macro translation is self-explanatory. The inline and non-exported functions are the less straightforward thing to handle. First, any inline function should be properly exported into Rust space. Second, bringing into scope non-exported functions by modifying `rust/kernel/bindings_helper.h` will bring many more items into scope than one may initially want. As such, thought needs to be placed into this. Obviously the Rust for Kernel developers decided not to export all functionality. Once these issues are addressed, we believe a complete Rust port of RamFS is possible. In addition, the main Linux kernel tree saw updates to RamFS *during* the duration of this project. In the future, it will be necessary to update Rust RamFS to account for these new changes[34].

5.2 Rustifying the Work

By far the more interesting remaining work is rustifying our base. Given the limited amount of time, we were only able to rustify part of our port. One key example of this process is the `seq_printf!` macro for printing to a sequence file. We will attempt to order our list of future rustifying work from low-hanging fruit to more difficult tasks.

First, currently Bindgen will auto-generate names for types and variables names for anonymous structs and unions in C code bases (see Listing 3). This is because Rust does not have a concept of anonymous structs and unions in the same fashion as C. However, when compiled, these anonymous structs and unions behave exactly like a union and struct in layout. Thus, Bindgen can generate bindings for them. However, it must do so with programmatically generated names. One simple solution to this problem that we have prototyped, but did not implement, is to add a conditional name to the structs based on a C macro definition. Since Bindgen uses libclang[35] to do it's work, we can define this macro when generating bindings to provide variable names to the anonymous struct only at Rust binding generation time. The negative to this is that now all anonymous structs and unions in C that we desire to apply this to must be annotated. Thus, this fix interferes with kernel C developers as well. However, we do not know of any other simple ways of getting around this. This is an issue because if the kernel changes its binary interface for a struct by reordering the fields, our code could break as the auto generated names are generated from top-to-bottom of a struct with numbers denoting the anonymous type/objects.

Second, we have a two pronged future work. One, we should make everything that is not expected to be exported private in Rust. This ensures the exact same interface that we had in C that was defined using the `static` keyword or private implementations inside of `.c` files instead of `.h` files.

Rust for Linux	Commit Hash: `3976bd9d29fd5b576671fc8ef72dbccf9509c027`
Rust Compiler	Version: 1.56.0
Rust Source	Installed via: `rustup component add rust-src`
`cbindgen`	Installed via `cargo`. Version: 0.20.0
Rust `bindgen`	Installed via `cargo`. Version: 0.56.0
`git apply` Command	`git apply /path/to/patchfile.patch`
Kernel Build Command	`make -j ${NUM_CORES_DESIRED} LLVM=1`

Table 1: These dependencies and commands all come from the Rust for Linux quick-start guide[15].

Currently, we have not guaranteed that our implementation adheres to this. Two, once we know what is exported, we should work on generating a Rust "safe" implementation with C-bindings on top of it. Currently, we are relying on Rust to use a C-style interface for all of the ported functions. It would be better to rely on a Rust-style interface only dropping down to a C-style interface when needed to interact with C code. This could be done by putting either the Rust and/or C-style interface into separate modules/files and doing the necessary conversion from references to pointers and vice-versa. It is even possible to add additional safety guarantees than the original code that only run in debug builds for double-checking pointer non-null facts. This has the added benefit that we can start to build up internal lifetime chains that will bring us some additional safety. It would also get rid of some of the unsafe blocks in our code pushing the safety checks onto the compiler instead of future maintainers.

Third, and finally, we could create traits for some of the different C vtables such as `inode_operations`. One would implement these traits for a struct instead of creating the vtable objects from kernel bindings. This will make the code more idiomatic Rust code. However, a lot of thought must go into the exact interface for this as we must interact with C and Rust. A nice interface would be to generate compile-time failures in Rust if an unimplemented function was called. While, in C, we would want to generate an object with null pointers for the unimplemented functionality. Furthermore, as this is a commonly used coding tactic in kernel C, it would be nice to have a macro to automatically generate the trait for a given kernel C vtable.

6 Building the Kernel

Because our source code is in-tree in the Linux kernel, and our changes touch multiple parts of the Rust for Linux kernel (not just the RamFS source code), we cannot package our source as a compressed tarball for submission. Rather, we have submitted our modifications to the Linux kernel as a patch file generated by `git`. Upon applying this patch and installing the needed dependencies, it is possible to build the kernel with Rust RamFS.

To achieve this, one must simply set up the proper build environment[15], checkout the correct base commit in the Rust for Linux commit history, and run `git apply` with our patch file. To set up the correct build environment, see the Rust for Linux quick-start guide (Documentation/rust/quick-start.rst)[15]. In addition to the dependencies listed here, `cbindgen` version 0.20.0 must also be installed, in a similar fashion to how `bindgen` is installed in the Rust for Linux quick-start guide.

In our experience, it is also vital to have a modern version of LLVM installed and use it. The Rust for Linux quick-start guide mentioned that in some situations it might be possible to mix a GCC compiled kernel with LLVM compiled Rust code[15]; however, we have ran into some issues with this. When building the kernel, enable LLVM by running `make LLVM=1` rather than just `make`.

7 Conclusions

Overall, porting code from kernel C to Rust was not as difficult as first imagined. However, there are still many hills to climb. We hope that our work can form a base to build other porting projects off of. Our addition of CBindgen as a general framework for incremental porting will hopefully inspire other people to do likewise with other Rust for Linux work. Our porting of RamFS to the Rust language shows that porting internal kernel code is indeed possible and feasible. As mentioned in our future work, there is still much work left to be done in making our code more idiomatic Rust and in exploring how the Rust memory model meshes with C. Despite this, our code serves as a base that hopefully others can build on to explore this relationship.

With this in mind, we hope, with the professors blessing, to offer our code and findings up to the Rust for Linux kernel as a contribution for future study and potential future use. We do not expect this to be mainlined into the kernel any time soon. However, with some Makefile tweaking, we believe that our code can be included as a KBuild option, allow future Linux kernel builders to build our RamFS code or the C RamFS code.

References

[1] C. Cimpanu, "Microsoft to explore using rust," 2019. https://www.zdnet.com/article/microsoft-to-explore-using-rust/.

[2] "Android rust introduction," 10 2021. https://source.android.com/setup/build/rust/building-rust-modules/overview.

[3] "Android rust binder source code." https://android.googlesource.com/platform/frameworks/native/+/refs/heads/master/libs/binder/rust/.

[4] "Popos github repo showing rust projects." https://github.com/pop-os.

[5] J. Salter, "Linus torvalds weighs in on rust language in the linux kernel," 2021. https://arstechnica.com/gadgets/2021/03/linus-torvalds-weighs-in-on-rust-language-in-the-linux-kernel/.

[6] L. Tung, "Rust in the linux kernel: Why it matters and what's happening next," 2021. https://www.zdnet.com/article/rust-in-the-linux-kernel-why-it-matters-and-whats-happening-next/.

[7] J. Edge, "Rust for linux redux," 7 2021. https://lwn.net/Articles/862018/.

[8] L. Torvalds, "Ramfs original source code," 2000. https://github.com/torvalds/linux/blob/master/fs/ramfs/inode.c.

[9] R. Landley, "Ramfs, rootfs, and initramfs," 2005. https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html.

[10] N. Elhage, "Supporting linux kernel development in rust," 2020. https://lwn.net/Articles/829858/.

[11] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues," 2019. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[12] stack.watch, "Recent linux kernel security vulnerabilities," 2021. https://stack.watch/product/linux/linux-kernel/.

[13] J. Corbet, "Rust and gcc, two different ways," 2021. https://lwn.net/Articles/871283/.

[14] antoyo, "rustc codegen gcc github repository," 2021. https://github.com/antoyo/rustc_codegen_gcc.

[15] R. for Linux, "Rust-for-linux/linux github repository." https://github.com/Rust-for-Linux/linux.

[16] "The rustonomicon." https://doc.rust-lang.org/nomicon/.

[17] "The rust programming language." https://doc.rust-lang.org/book/.

[18] "Rfc-0141: Lifetime elision." https://github.com/rust-lang/rfcs/blob/master/text/0141-lifetime-elision.md.

[19] T. R. C. Team, "Announcing rust 1.31 and rust 2018." https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html#non-lexical-lifetimes.

[20] J. Corbet, "Using rust for kernel development," 2021. https://lwn.net/Articles/870555/.

[21] "Cbindgen github repository." https://github.com/eqrion/cbindgen/.

[22] "Cbindgen source code: src/main.rs." https://github.com/eqrion/cbindgen/blob/11228f6b6f7c9f458a0d8c23e1647e53964a246b/src/main.rs#L98.

[23] "Rust for linux linux kernel source code: scripts/makefile.build." https://github.com/Rust-for-Linux/linux/blob/4349c441fcac85eedf464354c4180663b4e31887/scripts/Makefile.build#L345.

[24] "cbindgen user guide: Writing your c api." https://github.com/eqrion/cbindgen/blob/master/docs.md#writing-your-c-api.

[25] "Rust bindgen github repository." https://github.com/rust-lang/rust-bindgen.

[26] "The unstable rust book - asm macro." https://doc.rust-lang.org/beta/unstable-book/library-features/asm.html.

[27] "Rust language inline assembly github issue." https://github.com/rust-lang/rust/issues/72016.

[28] "The rustonomicon: Send and sync." https://doc.rust-lang.org/nomicon/send-and-sync.html.

[29] "The rust rfc book - const generics." https://rust-lang.github.io/rfcs/2000-const-generics.html.

[30] "Rust core documentation - transmute." https://doc.rust-lang.org/core/mem/fn.transmute.html.

[31] "Rust for linux github repository - commit 9e8bd67." https://github.com/Rust-for-Linux/linux/pull/280/commits/9e8bd679ecf29e8d776de322e1685e0db1d5acc0.

[32] "Iozone filesystem benchmark." http://iozone.org/.

[33] Linux-Test-Project, "Linux-test-project/ltp github repository." https://github.com/linux-test-project/ltp.

[34] "Git history for ramfs." https://github.com/torvalds/linux/commits/master/fs/ramfs.

[35] "Bindgen user guide - requirements." https://rust-lang.github.io/rust-bindgen/requirements.html.