Linux Device Drivers in Rust

Zach Schuermann
Columbia University

Kundan Guha
Columbia University

Abstract

Rust is a relatively new systems programming language focused on safety, performance, and concurrency [4]. The Linux kernel is currently implemented in C, an "unsafe" language which relies on the developer to manage memory, perform error checking, and reason about race conditions (as opposed to the compiler enforcing correct behavior). Rust, on the other hand, enforces semantics at compile-time which guarantee many unsafe behaviors are not possible. In an effort to bring many of Rust's safety guarantees to the realm of the Linux kernel, we attempt to build device drivers for Linux in Rust. Considering the majority of the kernel source is device drivers, and many are maintained out-of tree, any increase of safety in device drivers could have far-reaching implications for boosting the safety of a substantial portion of code running in kernel mode. Our contributions include: refining a system for building device drivers in Rust, implementing a handful of device drivers leveraging the system, and beginning a re-implementation of the Wireguard Linux device driver in Rust. The current ecosystem is very young and presently, developer productivity is hindered by the lack of support. We present our experiences working with Rust near the kernel and experiments with various device drivers and frameworks. In our preliminary experiments we found Rust has negligible runtime performance impact although increases driver sizes by an order of magnitude.

1 Introduction

The main project goal is to investigate the ability to increase Linux device driver safety via Rust implementations instead of the usual C implementations. The project's motivation stems from our interests in the applicability of Rust, a type-safe language which provides strong isolation, concurrency, and memory guarantees to operating systems and their surrounding infrastructure [11]. While many other languages provide similar guarantees and abstractions via garbage collection, Rust can provide memory safety (among many other safety guarantees) without garbage collection; and, in a performance-sensitive context such as the kernel, we hypothesize that Rust can provide similar performance to C while strictly increasing safety. This is possible via so-called "zero-cost abstractions": many of Rust's abstractions are effectively compiled away without incurring any overhead during run time; many other languages which provide similar safety expend run time resources (on things like garbage collection).

Since the majority of the Linux kernel source (as of 5.6) is in-tree device drivers, any success in increasing safety of device drivers will have significant implications for safety within the kernel.

Although secondary to the goal of increasing safety within the kernel, using Rust means developers have access to modern tooling and modern language features. Being able to support newer languages can help attract new developers, and Linux kernel module development has always been one of the primary gateways for people new to the Linux kernel to approach developing for it. Using Rust ensures that although new and less skilled developers can work with (or near) the Linux kernel, the developers can rely on the safety of Rust as a means for safer/easier development.

1.1 Contributions

- We **integrate prior frameworks and update** them for the latest Linux kernel. Prior to our work, there existed two efforts for writing kernel modules in Rust, which shared portions of their implementation. In an effort to improve the preexisting frameworks, we merged some of their functionality including the `sync` implementation, which is discussed more below.

- We **tested character and network device driver implementations** in the context of our refined framework. In order to validate early improvements and upgrade the frameworks for Linux 5.3.18 we implemented a simple character device which behaves similarly to `/dev/rand`, along with a simple loopback networking device. Simple analysis validated performance and exposed large binary sizes in Rust implementation.

- Additional **usability improvements** were included for writing kernel modules in Rust and clearly defining interfaces between Rust and kernel code. The remaining work to unify the unsafe interface and develop a means to easily modify the interface is left for future work.

- A **Wireguard implementation** was attempted for evaluating the effectiveness of device drivers written in Rust in order to measure relative performance, safety, and ease of development. In lieu of performance metrics, we provide a qualitative study on the feasibility and usefulness of Rust in such applications.

The remainder of this paper describes Rust and our choice of the language, the design of our system to develop device drivers in Rust, improvements on previous frameworks, our experiences developing a large-scale device in Rust, and an evaluation of both the framework and its usage for real devices.

2 The Rust Programming Language

The basic outline of *Rust* as a language is best given by Matsakis and Klock [11] as "a programming language for developing reliable and efficient systems... designed to support concurrency and parallelism". Importantly, Rust has a strong static type system which provides strong guarantees about isolation, concurrency, and memory safety in what is called "safe Rust". Specifically, Rust guarantees that safe Rust code will be free of data races, buffer overflows, stack overflows, and access to uninitialised or deallocated memory [11].

2.1 Safety

When operating under guarantees provided by the compiler, you have [10]:

- No segmentation faults.
- No buffer overflows.
- No null pointers.
- No data races.

For example, Rust provides types to encapsulate error checking which can then be enforced by the compiler. The code that follows shows two such types `Option` and `Result`. The former prevents the need for `NULL` constructs and the latter forces developers to perform error checks on the encapsulated type `T` through Rust's default behaviour of enforcing exhaustive branch checks.

```
enum Option<T> {
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

In a toy example, consider the following code in which we are searching through a vector for elements that satisfy a predicate. The nature of this computation requires the ability to return either the element found or something to represent the "nothing" case. Rust handles this cleanly with the `Option` type, which is returned by the `find()` method.

```
let v = my_vec.find(|t| t >= 45);
```

Again, the result of the parsing below relies on being either the correct parsing result or an error when parsing. This behavior is perfectly captured by the `Result` type which can then be "unwrapped" to yield the inner value.

```
// Result<i32, ParseIntError>
let n = "45".parse();
// i32
let n = "45".parse().unwrap();
```

The wonderful promises of safety are not always possible, though. Through use of the `unsafe` keyword, bits of code can effectively forgo most of the compiler's checking for enforcing the safety discussed above. In this case, it is up to the developer (as in C) to perform necessary checks for guaranteeing memory safety. That being said, the inclusion of `unsafe` code need not compromise the entire Rust source. In our case, (and in many cases like `std::sync::atomic` types), we rely on a safe interface to an underlying unsafe implementation. Though this may sound like a step in the wrong direction, leveraging unsafe Rust is necessary to make foreign function calls to C. Only then can we build safety on top of the inherently unsafe C which the Rust compiler has no way to reason about.

2.2 Performance

Preliminary work by Li et al. found Rust kernel modules to be two to three times larger in binary size, which is still on the order of *kilobytes* [8]. Since Rust has no garbage collector and a similar run time overhead to C/C++, its performance, in general, is comparable to C and C++ [2].

Our implementations are mainly concerned with lines of code (LOC) of a Rust implementation as well as the number of *unsafe* lines of code / unsafe blocks. In theory, access to higher-level abstractions allow for fewer lines of idiomatic Rust code for similar tasks compared to C, and minimising unsafe code allows for stronger guarantees to be made on the memory safety of a kernel module, with fewer areas requiring full manual auditing.

2.3 Usability

Many facilities exist in Rust that allow for smoother developer experience including: String manipulation, a dynamic `Vector` type, etc., which means one can forgo many of the functions and data structures defined in the kernel and instead rely on often easier to use and safer Rust mechanisms. Aside from language features, the surrounding tooling has improved via `cargo`, Rust's package manager, and well-developed programming tools like *Rust Analyzer* [13]. Combined with an intelligent compiler with non-cryptic error messages, the aforementioned improvements have led to a rather pleasant developer experience.

3 Design

The design of the frameworks used to develop Linux kernel modules rely on a well-defined interface from Rust to kernel C code. For normal kernel development, such interfaces are specified via C header files; however, Rust does not understand C header files directly. Normally, a data compatibility layer would be written in Rust that specifies Rust native types conforming to the same Application Binary Interface (ABI) as the given C interface. Once written, development would continue through the use of Rust's Foreign Function Interface (FFI) to execute C functions and pass data between the two interfaces. Unfortunately, there are two main issues in the case of the Linux kernel. First of all, the kernel is *large* and manually rewriting all pertinent interfaces, while potentially more code efficient, would be cost-prohibitive (in terms of developer time). More importantly, the Linux kernel does not guarantee ABI compatibility between releases, or even for the same release (as structures may be changed via kernel compilation flags). Therefore, instead of writing a manual interface, an automatic process leveraging LLVM compilation steps to translate C headers to Rust structures, called Rust Bindgen [14], is used to parse the required parts of the Linux kernel. Additionally, appropriate compilation flags are automatically found through the `kernel-cflags-finder` system [5] to ensure ABI compatibility.

However, bindgen is not a perfect translation. As it relies upon LLVM compilation, it cannot be used on code bases that `clang` cannot compile, which the Linux Kernel sometimes is, and it operates upon LLVM data generated *after* preprocessing steps and code inlining is performed. What this means, is all macros and inline functions cannot have automatic Rust bindings generated for them, or any sort of FFI operation. The standard workaround, when applicable[1], is to "wrap" these macros or inline functions in a C function which is then processed via bindgen and may be used in Rust via the FFI like any usual function. These wrapper functions may then be further wrapped in a pure Rust interface that guarantees safety requirements are met, as FFI calls are inherently `unsafe` operations in Rust, and used as required in module code. The complete library interface generation process is illustrated in Figure 1.

```rust
extern "C" {
    fn hello();
}

fn main() {
    unsafe { hello(); }
}
```

As an example, bindgen may be used to convert the following C code to the proper Rust calling convention (note the datatypes):

```c
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c,
 ↪  CoolStruct* cs);
```

becomes

```rust
#[repr(C)]
pub struct CoolStruct {
    pub x: ::std::os::raw::c_int,
    pub y: ::std::os::raw::c_int,
}
```

[1] Certain macros, such as the `container_of` macro, almost certainly would not be exactly wrappable as they rely on text-level metaprogramming to function

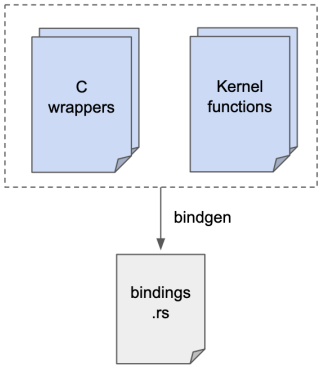

Figure 1: Kernel headers and associated C wrapper functions are translated to Rust bindings

```
extern "C" {
    pub fn cool_function(
        i: ::std::os::raw::c_int,
        c: ::std::os::raw::c_char,
        cs: *mut CoolStruct
    );
}
```

In the context of kernel code, the interface allows for using the generated bindings (types, functions, etc.) as normal (albeit unsafe) Rust via inclusion from the generated Rust source "bindings::". Similar to above, callbacks can be defined in Rust and invoked from C.

```
// fictional network callback
unsafe extern "C" fn net_xmit(
    skb: *mut bindings::sk_buff,
    dev: *mut bindings::net_device,
) -> bindings::netdev_tx_t {
    bindings::consume_skb(skb);
    return 0; // NETDEV_TX_OK
}
```

Furthermore, structs, such as those defining network device operations (`netdev_ops`), can be instantiated and modified in Rust. In summary, Rust can leverage **all** functionality afforded to C when in an unsafe context. The crux of this design is minimizing this unsafe intersection of C and Rust in order to provide a small, well-defined, and easily audited layer.

4 Linux Kernel Module Framework

In recent months, two separate[2] "Linux Kernel Modules in Rust" (LKMR) framework implementations have been released [5, 9], with accompanying conference talks or papers respectively [6, 8]. Both are implementations of "a framework to help developers to quickly build device drivers in Rust", which provide the majority of the functionality discussed above. Both frameworks had varying support for kernels above 5.0 and individually cover different subsets of kernel primitives in Rust implementation. It is worth noting one significant implementation shared between both: the redefinition of the `Global Allocator`. In order to request storage on the kernel's heap via `kmalloc()` and `kfree()`, Rust's usual memory allocator was swapped for a custom implementation which effectively wrapped calls to `kmalloc()` and `kfree()`, giving a safe Rust interface to the underlying implementation. In terms of basic functionality, in order to build a module from Rust both systems used a `build.rs` file to specify how to generate the required C-bindings via bindgen (discussed above), and `cargo` was configured to compile to a static object file, which could then be used in the standard Kbuild system. Of course, just like in C, the entire Rust standard library is *not* included and instead source files are compiled with solely the `core` and `alloc` (implemented through `kmalloc()` as mentioned) modules, along with a limited subset of modules re-implemented for kernel use.

In the end, the "fishinabarrel" implementation was deemed most mature, and as part of our contributions their interface was updated to support kernel version 5.3, and support for synchronisation primitives, such as `spin_lock` and `mutex`, were merged in from the other implementation. As discussed, these modifications mainly entailed adding proper C wrappers for any involved macros and defining the proper Rust interfaces and semantics. For example, locks were implemented using standard Rust semantics, where Rust encourages locking *data* rather than code sections. Specifically, a "lock" is a type which, upon locking, *returns* a different "locked" type from which the held data may be accessed. Unlocking may subsequently be implemented as part of that type's *Drop* framework, ensuring the lock is freed only after the data has stopped being used. In this manner, the compiler's type and borrow checker may be leveraged to make any improper accesses of the data (i.e. without having acquired the lock) a *compile time* error, rather than an unfortunate bug discovered at run time.

5 Network Devices

A simple loopback-like device was implemented in a attempt at a simpler target than Wireguard. While this proved possible, it solidified the necessity to standardize a means of creating device drivers with Rust. While Linux supports four classes of devices (character, block, network, and USB), an implementation of a Rust frame-

[2] Some files are shared between the two implementations but they are otherwise separate.

work could provide a means to effectively 'instantiate' each such case. Below, in the Results section, we will outline the design of a standardization which would reflect the Linux device driver interface and effectively use Rust's language features.

6 Wireguard

The Wireguard implementation is meant to evaluate whether the safety afforded by Rust can be leveraged in a real-world implementation, or if the reliance on unsafe functions and kernel API's is too great and the usage of Rust is not worth the developer's time. The following measurements are necessary for comparison:

- **Performance**: Performance metrics, including packet processing, throughput, and latency, must be collected and compared via benchmarking.
- **Safety**: The Rust compiler can count unsafe lines of Rust code to quantify the size of the unsafe interface.
- **Ease of Development**: The overall lines of code, along with anecdotal experiences as we develop the device driver, will be used to compare the relative ease of development.

Wireguard is a "fast, modern, secure VPN tunnel" operating at layer 3, implemented as a Linux virtual network interface [3]. Wireguard is a modern solution meant to replace IPSec and OpenVPN (as well as other userspace VPN's). We selected Wireguard for three reasons: (1) It is a real-world project with a seemingly large userbase. (2) It has a rather small footprint: only about 6000 lines of C. (3) It is relevant and actively developed; upstreamed in Linux 5.6.

While the initial project's goal was to create a full implementation of Wireguard in Rust, only a subset of its original functionality has been implemented in Rust.

Currently, Wireguard has proven a valid target for reimplementation due to its heavy usage of kernel functions (such as cryptographic facilities, etc.), as well as its own usage of data structures which can be replaced by Rust data structures: lists, structs, etc. However, the difficulty of an incremental rewrite means the current implementation is unfinished.

While Rust is often lauded for its ability to integrate with existing codebases via its strong FFI support, this did not apply well to the case of Wireguard. Since Wireguard defined many of its own data structures to be used throughout the system, there was hardly a possibility of an incremental rewrite. In order to reimplement even the smallest function, since it likely leverages some data structure which then depends on many, many more data structures (including their rust implementations), a small incremental rewrite is infeasible even with access to bindgen bindings.

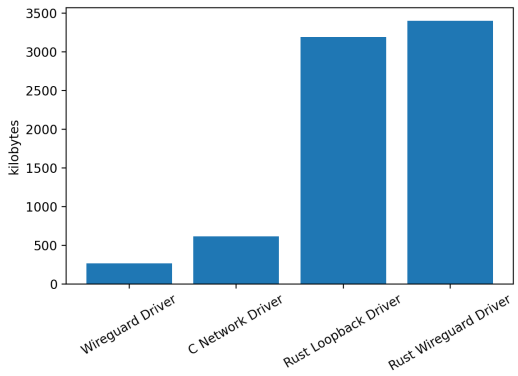

Figure 2: Device driver size comparison.

7 Results

The performance of the implementations of the character device and the network device were similar to that of C. Due to the simplicity of each implementation, rigorous analysis is omitted; furthermore, this is expected due to the architecture of the language and its ability to leverage zero-overhead FFI. More notably, the Rust modules were an order of magnitude larger than their C counterparts. This was observed in every one of the implementations. As of right now most of this is Rust's own overheads, due to including `core`, as even without its full standard library Rust still takes a very conservative style and includes more code than is strictly necessary to *all* compilations. Some further reduction in size may be had through more specific reductions on this side, but a certain amount of overhead is unavoidable at Rust's current stage of development, with future plans in the language's own development allowing for further reduction in code overhead.

Preliminary results encourage the use of Rust, though at a high developer cost: In order to leverage the necessary functions/structures exposed by the kernel, one must wrap via bindgen or (less often) manual wrapper implementations. In order to make this developer overhead worthwhile, a semi-stable Rust interface is much more desirable than each module developing their own wrappers themselves.

Only a small portion of the Wireguard codebase was rewritten in Rust. This is due to the highly-dependent nature of the Wireguard implementation, where despite the implementation itself being fairly "simple", and indeed not consisting of nearly as many lines of code per

file than the Linux kernel itself, much of the code of individual files depends on large parts of the kernel, other wireguard files, or both. While for any project such interdependencies are to be expected, in this specific case it meant that rewriting even smaller functions, let alone entire files, requires significant legwork in rewriting data structure frameworks and their implementations in Rust, as well as the requisite C-bindings. This requirement increases the complexity of the rewrite to the point where it is much harder to do a "simple" incremental rewrite, as each piece ends up requiring understanding significant portions of the total project.

The Wireguard implementation hoped to validate performance and analyze the amount of unsafe code utilized. More importantly, a measurement on whether the safety (i.e. how often things like Rust's `Result` can be effectively utilized) has enough of an impact to justify the use of Rust at all. So far, much of the implementation for Wireguard has relied heavily on kernel facilities though there has been some room to benefit from the many of Rust's advantages like the Vector type.

7.1 Limitations

Despite the sweet promise of safety within the context of Rust, there no doubt must exist an **unsafe** interface to the C kernel code it interacts with. No matter the safety afforded in "safe Rust", there will always be an unsafe interface. It seems rather unlikely for the Linux code base to transition away from C, however promising Rust may seem [7]. Nonetheless, there has been some talk of including a Rust device-driver interface in-tree [1]. Though more robust than before, the current framework to develop a Rust kernel module still leaves much to be desired. Hopefully, as it is developed, a safe Rust API can be stabilized around the Linux kernel API, allowing many device drivers to benefit from a single, vetted, interface with the kernel.

8 Future Work

8.1 Interface Design

In order to successfully leverage Rust, there must exist a well-defined interface between device driver implementations and the kernel API they must leverage. While this is a lofty goal due to the kernel's ABI instability, the chances of keeping a library up-to-date for the current kernel is much more likely if the development framework is maintained in-tree [1], or otherwise requires a significant community effort.

Currently, our implementation relies on a rather "hacky" method of implementing wrappers ourselves. While we built some facilities within the context of the kernel to attempt utilizing some of Rust's features (like locking data instead of code), we were reduced to manual wrapping for a number of other API's (Figure 3). In order to use Rust's strengths and promote developer productivity, it is much more desirable to provide a safe wrapper for the kernel API's such that device driver developers can lean on Rust's compiler to enforce its safety (Figure 4).

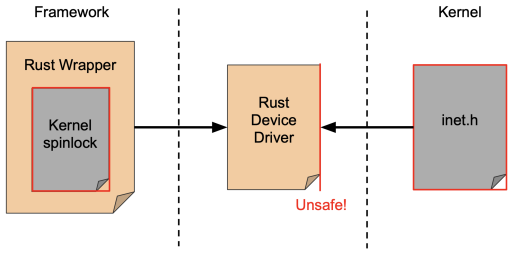

Figure 3: Current unsafe interface reliant on developer wrapping certain kernel API's manually.

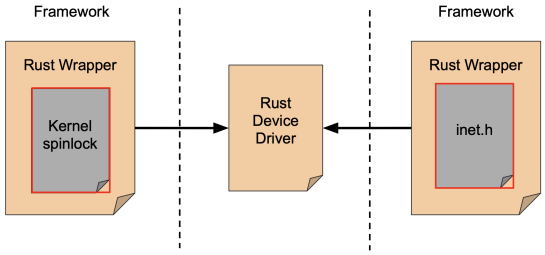

Figure 4: Ideal scenario in which a single framework can encapsulate kernel API's.

The interface design, we hypothesize would provide a means to effectively use Rust, is one which uses Rust's `traits` feature to implement different drivers entirely (or mostly) from safe Rust. Take, for example, the Nix [12] crate: it wraps Libc and provides a safe, idiomatic Rust interface to the inherently unsafe C functions. The framework we hope to implement would function similarly: it would provide not only safe versions of kernel API's, but also a stable and idiomatic framework within which developers can implement device drivers.

Rust's traits allow for defining *shared behavior* (much like interfaces in other languages). A framework could map each of the four device driver kinds to a specific type and so long as that type implements certain trait for the driver kind, it is easily loaded all within the context of

the framework. This ease of development relies on two major parts: (1) an implementation of the Linux device driver API as traits/types and (2) a safe wrapper for the Linux API such that it can be utilized within the driver implementation.

The development of the aforementioned two implementations is no doubt a large undertaking, but would most likely result in greater accessibility to Rust device driver development. Despite the difficulty of Rust as a language, the formalization and (hopeful) documentation of the Linux device driver interface could prove to be easier and more approachable than the current requirements for device driver development.

8.2 Limiting Module Size

The current framework results in rather large binaries. With the current implementation, a number of dependencies are included which should be stripped from the final module. There are two means we leave for future work. Either (1) attempt at refactoring dependencies such that only a select few which are truly required by the module are included, while others are so-called 'dev-dependencies' or (2) integrate into the build system a means of cleaning/stripping the binary of unused artifacts. These have not yet been explored.

9 Lessons Learned

Despite the lack of a large Rust implementation in a project that originally targeted such an outcome, we found the project to be an eye-opening experience through learning the Rust ecosystem and exploring the possibility of bringing safer interfaces to Linux device drivers.

Firstly, the Wireguard implementation began as an ambitious idea. Without the humanpower to develop an extensive network device, this was infeasible without the support of an effective framework. Although the implementation has not come to fruition, the outcome is hopeful and has shown the requirements of using Rust for device drivers. This points to the value of developing pieces of the framework itself prior to undergoing a grand project without the support of a framework. Though, attempting the rewrite was possibly the best way of reaching these conclusions.

Secondly, Rust achieves some amazing goals. The promise of speed and safety is really astounding. That being said, when Rust seems to provide speed and safety without compromise, the developer is really the one that must compromise by climbing a steep learning curve. That is, Rust speed and safety comes at a cost of learning a rather difficult language. While this is an investment which we believe is paid many times over in the future, in time-constrained projects, (like a semester!) learning Rust and teaching new developers can significantly impede forward progress. However, now that we *have* put in that effort, we hope to continue to work on related projects in our future personal or academic pursuits, if possible.

References

[1] multiple authors. *Rust is the future of systems programming, C is the new Assembly*. https://lwn.net/Articles/797828/. 2020.

[2] Doug Bagley. *The Computer Language Benchmarks Game*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html. 2020.

[3] Jason A. DonenFeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2017*. Network and Distributed System Security Symposium, 2018. URL: https://wireguard.com/papers/wireguard.pdf.

[4] *Frequently Asked Questions - The Rust Programming Language*. https://prev.rust-lang.org/en-US/faq.html. 2020.

[5] fishinabarrel. *Linux Kernel Module Rust*. https://github.com/fishinabarrel/linux-kernel-module-rust. 2019.

[6] Alex Gaynor and Geoffrey Thomas. "Linux Kernel Modules in Rust". https://ldpreload.com/p/kernel-modules-in-rust-lssna2019.pdf. Linux Security Summit North America 2019. 2019.

[7] Amit Levy et al. "The Case for Writing a Kernel in Rust". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems - APSys 17* (Sept. 2017). DOI: 10.1145/3124680.3124717.

[8] Zhuohua Li et al. "Securing the Device Drivers of Your Embedded Systems: Framework and Prototype". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. ARES '19. Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019. ISBN: 9781450371643. DOI: 10.1145/3339252.3340506. URL: https://doi.org/10.1145/3339252.3340506.

[9] lizhuohua. *Linux Kernel Module Rust*. https://github.com/lizhuohua/linux-kernel-module-rust. 2019.

[10] Nicholas D. Matsakis and Felix S. Klock. “The Rust Language”. In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: https://doi.org/10.1145/2692956.2663188.

[11] Nicholas D. Matsakis and Felix S. Klock. “The rust language”. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT 14* (Oct. 2014). DOI: 10.1145/2663171.2663188.

[12] nix-rust. *nix*. https://github.com/nix-rust/nix. 2020.

[13] rust-analyzer. *Rust Analyzer*. https://github.com/rust-analyzer/rust-analyzer. 2020.

[14] rust-lang. *Bindgen*. https://github.com/rust-lang/rust-bindgen. 2020.