Linux Kernel Support for Rust
Final Report

Tom Tase
Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
ttase97@vt.edu

Kashi Vishwanath Kolloju
Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
kkashivishwanath@vt.edu

Kevin Chahine
Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
kevinc21@vt.edu

***Abstract*—This report describes the work that has been completed, the challenges face and future work than can be done to support Linux Kernel Development in Rust. It will describe the background of Rust, reiterate the goals of the project and which have been achieved. It will evaluate Rust as a systems programming language. This paper describes how we implemented a kernel module in Rust and compare it to one written in C, analyze the performance against a traditional C based counterpart, and document the positive and negatives of each implementation.**

1. Introduction

For several decades C has been the language of choice in low-level kernel and driver development. While the Linux kernel has seen many improvements over that time, C has remained a constant as the only officially supported language for kernel development. And for many great reasons. C is very fast and efficient, it provides developers with necessary control over memory and hardware and is a overall great language for development of any program. C is a low-level language that provides no protections or safety guarantees for memory management, making it unsafe. As such code written in C is more prone to bugs than other high-level languages. While code in the Linux kernel is subject to rigorous testing before deployment, bugs still make their way into the OS. These issues create further vulnerabilities since most operating systems assume that all kernel code is trusted [1].

The risk of running unsafe kernel code is high. In 2017 the Common Vulnerabilities and Exposures database listed 217 vulnerabilities in the Linux kernel that can be attributed to unsafe code [2]. While not all the vulnerabilities were related to memory management, some were related to human errors related to low-level reasoning including bounds checking, lifetimes, etc [2]. Moreover, Linux does little to isolate errors caused by kernel code and extensions (modules). As a result, bugs may cause undesirable behavior such as kernel panics, memory leaks, kernel thread crashes, or deadlocks [3]. These errors are often beyond the scope of common software verification tools, making them harder to catch in development.

Attempts have been made in the past to create kernels and kernel extensions using languages that provide some form of memory safety. However, most of these solutions relied on a garbage collector or other expensive runtime checks [1, 2, 3]. The overhead associated with this process remains prohibitive for high performance applications [2]. Recent studies have looked at the possibility of using type and memory safe languages that do not perform garbage collection or any other runtime services for kernel development. A relatively new language, Rust, fits the description. Rust is a type and memory safe language that performs with runtime characteristics similar to C [1].

This project presents an implementation of a kernel extension (module) in Linux written in the Rust programming language. Section 2 presents the background research conduct and any related work. Section 3 covers the project setup and status of the module development. Section 3 also presents the remaining project schedule. Section 4 covers the implementation and status of the project to date. Section 5 outlines the process for testing and evaluating the performance of the created module as well as the currently recorded results. Section 6 focuses on possible future work related to the project as well as planned attempts to continue debugging issues encountered during implementation.

2. Background and Related Work

Most operating systems rely on the assumption that kernel code is trusted, unsafe or not. Traditionally, kernels have relied on hardware enforced memory protections on top of isolation provided by process abstractions [1]. C has been the primary language for Linux kernel and kernel module development. While powerful and efficient, C is considered an unsafe language as it provides no protections. This can often lead to the introduction of bugs and vulnerabilities into the Linux kernel. Some research has been done to answer the question - Is it possible to use a memory-safe, high-level programming language to write a kernel or module in place of hardware protections? Some examples include Spin [4], Singularity [5], and Biscuit [6]. These were high level programming languages that used a built-in garbage collector, which causes headaches in kernel code.

This project proposes to write a kernel module in Rust, a type-safe language that does not perform garbage collection, or any other runtime service.

2.1. Errors in the Linux Kernel

Many of the bugs present in the Linux kernel to date are the result of human error. Given C has no built-in static analysis for memory safety or concurrency, certain bugs are only noticeable during testing at the earliest. These errors can have a significant impact on the function of the kernel. Unlike user-space applications which are isolated, kernel space applications are treated as trusted code [2]. If unnoticed, these bugs can introduce vulnerabilities and exploits into the operating system or even cause the operating system to freeze. Some of the most common low-level errors that can occur are:

- **Memory:** incorrect memory usage, such as dereferencing a NULL pointer, double free, use-after-free, and memory leaks.
- **Concurrency:** data races and deadlocks.
- **Type:** mismatch between variable type and usage, usually when erroneous values are treated as valid.

Since the kernel is essentially a single monolithic C program, one small bug can cause system-wide failure [7]. Arguably, a full system crash is the desirable outcome as the bug would be noticed. More subtle issues would be difficult to detect and may cause undesirable behavior. Consequently, these bugs can also be exploited by malicious parties as identified by [8].

2.2. Current Approach to Kernel Extensions

Kernel level code depends heavily on callbacks and shared data structures. This generally requires multiple mutable references to the same memory location. C provides a powerful and efficient framework for this. In Linux, C remains the only officially supported language for kernel and kernel module development. Previous work has been done in the field of safe extensibility for kernel development. The aforementioned Spin, Singularity, and Biscuit projects increase the safety of kernel code through the checks and guarantees of the languages they are written in [3]. There are two major downsides to these projects. For starters they employ a garbage collector. In kernel space a garbage collector produces undesirable non-deterministic timing characteristics due to background locks that pause the entire OS. Furthermore, garbage collection creates challenges with memory placement and layout [1]. The other downside is these projects require creating an entirely new OS, and therefore are not compatible with the current Linux Kernel [3].

Another approach explored to ensure safe extensibility is Software Fault Isolation (SFI). SFI mechanisms enforce process-like boundaries around modules [2]. Additionally, SFI performs runtime checks to ensure modules cannot affect other systems [3]. These checks introduce overhead, which is not acceptable for modern systems [practical, sys]. Moreover, SFI mechanism do not automatically assume that a module is trusted [3]. In some cases, this restricts passing data by reference since the sender would maintain access to the data. Hence, a copy is required to maintain isolation [2]. While SFI provides a method of isolating and restricting the impact of module faults, it does not reduce the number of software bugs.

Rust, a high-level language with some low-level characteristics would allow for the implementation of extensions on the existing kernel. In fact, projects already exist exploring the implementation of drivers in rust [9]. Also, Rust does not perform any runtime checks, eliminating some of the issues seen with Spin, Singularity, and Biscuit. Furthermore, Rust's single ownership model enables SFI with minimal runtime overhead [2]. This makes it an ideal choice for safe module development in Linux kernel.

2.3. Rust Key Concepts

Rust is a type-safe language with fine grained control over memory management and concurrency. Rust employs a single ownership model that determines the lifetime of values at compile time, ensuring memory is freed when variables go out of scope. Some of the key concepts of safe Rust code are ownership, move, borrow, mutable borrow, and lifetimes [7].

- **Ownership:** An object in rust has a single owner. An object is bound to one owner, but can be copied if it implements the copy trait. Ownership allows for memory safety without runtime checks.
- **Move:** A variable or object type that does not implement the copy trait will transfer ownership to a new variable or object if assigned. The old variable becomes invalid and any attempt to access the old variable results in a compiler error.
- **Borrow:** Similar to references in other languages, borrow creates an immutable reference to a variable or object. Any attempt to modify the borrowed reference will result in a compiler error. References are dropped when a borrow is out of scope.
- **Mutable borrow:** A borrowed reference can be defined as mutable. The caveat is there can only be one mutable reference to a piece of data at a given time. As such, multiple mutable references will cause a compiler error.
- **Lifetime:** References in Rust are bound by lifetimes. These can be automatically inferred by the compiler or specified by the programmer.

These key principles help guarantee memory and type safety in the Rust language. However, the stringent single ownership characteristics creates a unique challenge for kernel code. As mentioned in section (current approach) kernel code generally requires multiple mutable references to the same memory. This is not possible since only one mutable reference can exist at a given time. An example of

```rust
// Rust
// This code will not compile
enum ValOrPointer {
    Val(u32),
    Pointer(&mut u32)
}

let external : &mut ValOrPointer;
match external {
    Pointer(internal) => {
    // Breaks safety rules by
    //writing to memory at 0x12345678
    *external = Val(0x12345678);
    *internal = 10000;
    }
}
```

Listing 1: Attempted Multiple Mutable Reference in Rust

Rust's ownership model can be represented when comparing the enum type in Rust to the similar union type in C. In Rust, the language ensures that it is impossible to access more than one parameter of the enum after it has been set. However, in C no such protections exist since having multiple mutable references is allowed. Levy et al provide an example of this issue depicted in Listings 1 and 2. In this scenario the C code would allow the construction of arbitrary pointers, while Rust would throw errors on compilation [1]. With these restrictions implementing data structures such as doubly linked lists would be impossible.

```c
// Equivalent C
// Compiles without any warnings
union ValOrPointer {
    uint32_t Val;
    uint32_t *Pointer;
}

union ValOrPointer *external;
uint32_t valptr = &external->Val;
*valptr = 0x12345678;
*external->Pointer = 10000;
```

Listing 2: Multiple Mutable Reference in C

2.4. Exploring Kernel Extensibility in Rust

Studies have demonstrated that Rust can be used for building high-performance kernels for embedded [10] and conventional [11] operating systems. This section will explore the possibility of using Rust to extend the functionality of the existing Linux kernel through modules and drivers. The process for developing kernel modules using Rust is covered in section 4.1.

By design, Rust is a safe language. Conversely, writing a kernel or kernel module sometimes requires writing unsafe code. In these instances, the unsafe code is generally hidden behind a safe interface. One example of this is the Rust library code [1]. One example of this is the Rust Cell abstraction. This abstraction allows multiple references to a mutable object [1]. While this seems to solve the main issue associated with kernel development in Rust, it comes at a computational cost. The Cell API requires copying of the data in order to access it [1]. Hence, multiple referrers (borrowers) operate on their own separate copy of the shared data.

There are several other challenges that arise from programming a kernel module in Rust. For starters, the rest of the kernel is written in C. The Rust module will need to interface with the C code. The bindgen tool is used to generate bindings that allow existing kernel functions to be call from Rust code [12]. However, concerns of unsafe code in the kernel still remains. The existing kernel interfaces were designed for extensibility, not type safety. As a result, passing data to and from the kernel becomes an issue. The existing interfaces often obscure the object type information passing regions of memory as raw pointers [3]. These raw pointers contain no guarantees about size of validity. Kernel extensions commonly involve passing resources across the OS/extension boundary. This poses a challenge to Rust's automated memory management. Finally, kernel code and extensions frequently use pointer arithmetic and type casting which are restricted in Rust.

3. Goals

The target is to implement a kernel module that interfaces with existing kernel data structures, and compare that performance to the equivalent C code. Here we reiterate our team goals from the project proposal and interim report.

- 75%: Check whether the kernel module written in Rust is supported with the kernel after installing required compilers (LLVM).
- 100%: A kernel module written in Rust that supports existing kernel data structures API's. Check for overhead that occurs with Rust against modules in C.
- 125%: As we faced a lot of difficulties using kernel data structures in previous projects here, we hope to implement a wrapper for kernel data structures APIs in Rust that can be easier interface with than the existing APIs written in C.

4. Implementation Status

4.1. Environment Setup

We expected built in Rust containers to crash when run in kernel space. Because of this, we plan to implement wrappers that use kernel APIs. Interacting with existing kernel APIs requires a tool called Bindgen. Bindgen generates bindings for kernel function calls that are accessible in Rust. Bindgen can be installed with cargo, which comes with the Rust installation. Rust's compiler is based on LLVM and provides configurations for many common targets [7]. Rust's source libraries take the form of crates (or packages). These crates provide the dependencies needed to perform basic actions and core functionality. They are similar to standard libraries in C and C++.

4.2. Project Setup

We have setup a centralized git repo on the campus' CS servers which we have access to on all our devices. That will speed up collaboration of our development. The git repo is currently separate from the kernel source code and is used to only implement user space code. An issue that will arise, is how to integrate code from our git repo into the Linux kernel's repo without having to fork the whole kernel repo into the CS servers. To solve this, we will to turn our repo into a git sub-module and add it in our local kernel repos. Because we need to compare the performance of Rust vs C, we will implement two projects, one in Rust and one in C. The programs will be functionally equivalent and help us to compare the performance of the two languages along with their developer friendliness.

4.3. Learning Rust

We began our project by learning the Rust programming language. The Rust community, provides a very well written book that explains everything from how to set up the Rust development environment to using almost every aspect of the language. We found this to be an important starting point to speed up our development and code quality in later work. From the book, we learned everything from how to install the compiler "rustc" and build manager "cargo" to using generics. We learned how to write, compile and run a simple "hello world" application and use comments. We learned how to install build dependencies and third party libraries using "cargo". Cargo is also used to create, build and run projects more automatically than rustc. The book goes on to explain the language. We learned how to declare variables and specify data types. We learned how rust deals with constant and mutable variables. We learned IO operations that stream standard input and standard output. We learned how to to stream data from standard in and out to /proc and the dmesg ring buffer. All this is done without making the module sleep, which can be problematic in kernel code. We then went on to learn control flow logic using ifs and loops. Later we learned the syntax of functions. Most of the syntax is very similar to C which made learning Rust easier. We have also learned how to use the built-in containers: Vec, LinkedList, HashMap and BTreeMap.

We then learned how to use generics which is called templates in other languages. In Rust, a generic is a function, class or method which is defined without explicitly specifying one or more data types. When generics are used elsewhere in code, the data type of their parameters are explicitly specified. When this happens the compiler automatically generates functions, classes and methods based on the generics definition and the specified data types. We believe generics give Rust a big advantage when working with data containers and algorithms. In Rust, a container is a type of data structure responsible for storing multiple elements of the same data type. For example, vectors, linked lists and hash tables. Generics give Rust a big advantage because they allow implementing a single function template that will work for multiple data types. In our case, we tried using generics to implement our evaluation functions to work with all the various Rust containers. The Rust book claims that generics to not add any runtime overhead when compared with normal functions. We found that to be true when comparing the execution time of a function written as a normal function verse the same function written as a generic.

4.4. Implementing the C module

We implemented our C module as a base line for evaluating Rust. We successfully implemented the C module to perform all the tests we chose in our interim report. The C module we implemented a fill, find, and for each operation for four data structures: linked lists, xarray, red black trees and hash tables. We implemented code to measure the execution time of each operation in units of clock cycles. And we printed the performance results to the proc file system and with the ring buffer. For testing purposes the c module accepts an input for the number of elements to add to each data structure and how many tries to attempt. The greater the number of tries, the less uncertainty there will be with the results due to a larger sample size.

4.5. Implementing the Rust module

Although writing code in Rust was slightly easier than C, implementing the Rust kernel module was much more challenging. We began by implementing a Rust program as a user space application. Doing this showed us that the majority of tools we needed are working and showed that we have a good enough understanding of the language syntax and build process.

We also implemented type agnostic test code using generics. This proved to be more difficult than expected. Rust generics require code to specify the traits that the templated types will have. For example, if a generic function is to assign a variable of a generic data type, then the copyable trait is to be explicitly specified in the generics header. This way, developers can't accidentally pass an object that cannot be copied. This is more type safe than macros, but it does make generics more challenging.

But all this was only implemented as a user space application. We then attempted to re-implement the Rust application as a kernel module. We successfully implemented, compile and ran the Rust code as a kernel module. Accomplishing this required modifying the build to use a nightly version of the rustc compiler. This version removes many of the error checking and safety features of the compiler but allows integration with unsafe kernel code. The Rust module, successfully interfaced with the proc file system and the kernel ring buffer. Passing data to the ring buffer was very straightforward. We simply needed to route data from print functions to the ring buffer instead of std out.

But we failed to integrate the kernel APIs into the module using bindgen. Because of the limited amount of research that has been published on the topic, we could

not find enough information to make bindgen work. Online forums did not contain solutions to this problem and we only found partially complete instructions on using bindgen. The internet seemed to have more questions than answers and that did not help us at all. This is to be expected because kernel development in rust is still a very new topic and is still in its early stages of development.

5. Testing and Evaluation

5.1. Evaluation

Our evaluation is intended to determine if Rust is a suitable programming language for Linux kernel development especially with existing kernel APIs. Our evaluation is focused on the performance of Rust and the code itself and how Rust helped us in the developmental of this project. Rust is not intended to be faster than C, but it is claimed that it isn't much slower. Rust and C have been shown to perform at nearly the same resource consumption. Because of this, the only thing Rust has to offer Kernel developers that C doesn't is faster and more efficient development. If Rust is easier to use and speeds up development, then maybe it can be used along side C. If it isn't, then there would be no reason to use Rust except to satisfy bias preference. We discovered that when compared to C, development using Rust is better in some ways and worse in others. This evaluation is in no way intended to shame the C language which is used to implement so much of the most used software in the world. Instead it is intended to show the advantages of the Rust language and how it can be used to aid developers in writing kernel code.

What stands out the most is that Rust makes it more difficult to create run-time errors. This is a great feature because run-time errors in the kernel can be very hard to find and debug. In the best case, run-time errors will appear quickly in simple testing. In the worst case, run-time errors will not occur until long after the code is deployed making the errors harder to reproduce and fix. Not only that, they risk creating vulnerabilities in industry applications that rely on a robust kernel. We noticed this when the compiler prevented us from using shared mutable references. Had the compiler not prevented the build, a run-time error could have occurred if the reference was deleted in one part of the code before another part finished using it. This would result in a dangling reference which could cause the kernel to hang and is a common bug in kernel development. Another thing we noticed is, the rustc compiler gives more useful compile time error messages. Along with that it even shows suggestions on how to solve the errors. It also provides links to find more information about it. The gcc compiler is knows to give helpful error messages, but we found that rustc is still more informative.

Another beneficial feature that rust provides developers is an alternative to namespaces. In C, each function must have its own unique name. As projects grow overtime, the chances of two functions having the same name increases and the need to prefix function names becomes inevitable. Some high-level languages solve this problem with namespaces and function overloading. Although rust does not support either of these, it does support structs and static methods. When used together, they are effectively the same as namespaces. And since structs in Rust are compiled into the equivalent of C structs and functions, structs do not incur the additional overhead that a high-level language might. By implementing functions as static methods, code can be organized into modules making code more maintainable and easier to read. These two things are important for the success of any project.

According to Marc Gregoire, the best measure of the quality of code is its re-usability. Re-use is an indication that the code was well written. Re-usability can be improved using encapsulation and object oriented programming. Because Rust supports structs, it allows developers to more naturally implement object oriented code and it does it without the overhead of constructors and destructors. This allows developers to encapsulate not only functionality but also data in order to hide complicated implementation details. This ultimately makes code easier to implement and more reusable. Encapsulation makes code reusable by preventing developers from accidentally using complex code the wrong way. It also means that they don't need to know as much to use it properly. We found this to be a very useful feature in our project. If all kernel APIs were written using encapsulation and object oriented programming, writing kernel code would be so much simpler and faster. Developers would spend less time debugging and more time creating new and more advanced features for the kernel.

Another luxury which Rust provides is generics. Generics make it possible to write code that works on multiple different data types. The main requirement is that each data type used must posses the traits that are needed in the generics definition. Doing this efficiently in C is a huge hassle for developers. If developers try implementing type agnostic code using pointers and casting, then programs will suffer from run-time performance and cache unfriendliness. Macros functions on the other hand must expose their implementation details to the code where they are called making them less encapsulated and more error prone. Generics made it easier for us to develop more functionality with less code when we implemented one function to test the performance of each of the four containers. In our C module, we needed to implement a different function for each container we tested. The code in each container was the same except for the data types used. Imagine if we needed to test hundreds of containers. With a single well composed generic function, a small snippet of code could be used for any number of data types.

A feature in Rust that creates memory safety and convenience to developers is references. Unlike C, Rust does supports passing variables to functions by reference. In C, this can only be accomplished using pointers. References in Rust provide two important benefits. One is that the parameters can be used like value-type variables without being dereferenced making code more readable. The other

benefit is that it prevents developers from accidentally passing a null pointer. Because references are guarenteed to point to something, function implementations don't need to check for nullptrs making them simpler and more safe by nature. With references, developers no longer need to be concerned with the possibility of null pointers being passed into functions. This makes run-time errors resulting from dangling pointers less common and removes the need for developers to wonder which pointers need to be deallocated. This did not significantly help us in our project, because we falsely assumed references to work exactly the same a they do in C++. Until we learned that passing a variable by reference requires the symbol to be used in both the function definition and function call, we had trouble working with this feature. Nevertheless, references were helpful once we understood how they are intended to be used.

The attributes of the project that evaluate are as follows. These were measured and compared for each language (C and Rust):

1) Fill Algorithm:
    a) Xarray Fill
    b) LinkedList Fill
    c) Hash Table Fill
    d) RBTree Search Tree Fill
2) Find Algorithm:
    a) Xarray Find
    b) LinkedList Find
    c) Hash Table Find
    d) RBTree Search Tree Find
3) Iteration Algorithm:
    a) Xarray For Each
    b) LinkedList For Each
    c) Hash Table For Each
    d) Binary Search Tree For Each

Note: This test will loop through each data structure and modify each element.

Each algorithm was tested using 500, 1000, 2000, and 4000 elements. This will give a good indication of how well each language scales with data size. Each was tested for execution time.

Execution time will be measured using timestamps. These timestamps will keep track of the CPU clock and be used to measure the execution time from before a function call to after its return. The duration was then divided by the number of elements in the container and measured either in terms of clock cycles per element. We expect the Rust program to be a little slower than C because of its garbage collection and run-time checks. Even though we aren't trying to prove Rust to be faster than C, verifying similar execution times is a indication that Rust can perform as an efficient language.

5.2. Results

Currently, only tests for the C module have been run due to issues with bindgen. Tables 1, 2, and 3 depicts the results of running the c modules on a fedora VM with a varying number of inputs. Total time was measured in processor cycles using the Linux rdtsc function.

	Number of Elements			
Data Structure	**500**	**1000**	**2000**	**4000**
Linked List	788988	1334624	2362336	4932472
Hash Table	588714	1091110	2238270	4840052
RB Tree	789398	1859368	3356200	6309910
XArray	125540	98268	183642	384668

TABLE 1. EXECUTION TIME IN UNITS OF CLOCK CYCLES TO INSERT ELEMENTS INTO DATA STRUCTURE FOR C MODULE

	Number of Elements			
Data Structure	**500**	**1000**	**2000**	**4000**
Linked List	115620	133458	266188	486098
Hash Table	21736	126974	462826	1049172
RB Tree	2030	2740	2914	3086
XArray	96396	24228	43588	95602

TABLE 2. EXECUTION TIME IN UNITS OF CLOCK CYCLES TO FIND ELEMENTS IN DATA STRUCTURE FOR C MODULE

	Number of Elements			
Data Structure	**500**	**1000**	**2000**	**4000**
Linked List	46242	107582	424888	878574
Hash Table	96426	229818	481782	1023300
RB Tree	79108	231568	442984	1019620
XArray	22654	42328	170310	184198

TABLE 3. EXECUTION TIME IN UNITS OF CLOCK CYCLES TO LOOP AND INCREMENT ELEMENTS IN DATA STRUCTURE

The key takeaways from these results is that certain data structures are better for certain tasks. The red-black tree was slowest to insert elements, but was easily the quickest in the find tests. The XArray seems to be the most powerful and efficient of all the other data structures. XArray also seems to be the safest data structure since it does not require any direct memory allocation or deallocation.

In C the data structures behaved as expected. However, without any memory safety features bugs during implementation did cause the kernel to panic and require a full reboot of the OS several times. This increased the development time for this kernel module. With Rust, many if not all of these issues would have been caught by the compiler. This would have reduced development and debug time in comparison to the C counterpart. But due to the novelty of Rust development in the Linux kernel, numerous other issues were encountered when trying to implement the counterpart in Rust. Several implementations, including linked list and red-black tree kernel data structures using Rust APIs were unfruitful because of the issues with bindgen. As such there are currently no results for the Rust implementations as we continue to debug issues with the bindgen tool.

The bindings required to integrate Rust into the kernel are currently not supported by the newer versions of Linux and hence the errors. Moreover, we attempted to use the forked Linux kernel developed for Rust development [13], but were unable to boot the kernel even after successful compilation.

6. Future Work

If we were to continue working on this project, we would focus on implementing a developer friendly API for each of the kernels containers. The APIs we develop, would be made to match the built-in containers that come with Rust. That way, developers who are familiar with Rust APIs will have no trouble, using them in kernel development. The difference in our APIs would be that they encapsulate kernel APIs using bindgen. With the use of generics this can be accomplished to work on any data type. This will significantly aid kernel developers who use Rust. It can make using kernel containers extremely safe and easy. Why use bindgen as opposed to re-writing the code purely in Rust? Using bindgen allows data to be stored and executed in the same way it would be in a C program. Without this, interfaces that rely on shared memory will not execute properly.

This paper has not considered the challenges associated with maintaining kernel code written in two very different programming languages. If the Kernel developers are to adopt Rust into normal development, it would require programmers to understand both C and Rust. Maintaining a code base in multiple languages makes it difficult to fix errors and make modifications because a programmer would need to thoroughly understand each language they are working with. Research on this problem is important to prevent complications for happening if Rust is to be accepted into Linux development.

Finally, kernel development in C is not likely to stop at the adoption of Rust. This paper has addressed integrating C code into Rust. But it hasn't addressed whether Rust code can be integrated into C. What challenges will have to be overcome to allow libraries written in Rust to be made compatible with code written in C. Because of this, we don't think Rust will truly be integrated into all parts of the Linux kernel. Especially not to implement kernel libraries. But Rust could still be a great language for implementing device drivers and kernel modules. For the most part, it will best suit parts of the kernel which will not become dependencies of C code.

7. Conclusion

Writing Kernel code in Rust would greatly reduce the quantity of errors in the Linux Operating System. Previous studies have attempted to look at other high-level languages for kernel and driver development. However, due to excessive run-time checks or incompatibility's with current kernel code, the projects never caught on. Rust could change this given it's C like performance and compile time static analysis.

This project has provided a kernel module programmed in Rust as a proof of concept. The module was assessed against a module written in C for performance and ease of implementation. Using the several tools to quantify performance that are built into the kernel, we produced a comprehensive analysis of the Rust module in comparison to the C baseline. By providing a proof of concept and performance analysis, this project has demonstrated that implementing a kernel module in Rust is possible and just as efficient as traditional C. But ultimately, Rust is still in its early developmental stages and is not yet ready to be used for Linux kernel development. There are still too many complications and pitfalls which will slow down integration of Rust into the kernel. Primarily due to bindgen, there is still much research and development that needs to be done before Rust can be considered for kernel development. For now, it is a great systems programming languages with lots of potential.

References

[1] Amit Levy et al. “The Case for Writing a Kernel in Rust”. In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys ’17. Mumbai, India: Association for Computing Machinery, 2017. ISBN: 9781450351973. DOI: 10.1145/3124680.3124717. URL: https://doi.org/10.1145/3124680.3124717.

[2] Abhiram Balasubramanian et al. “System Programming in Rust: Beyond Safety”. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 156–161. ISBN: 9781450350686. DOI: 10.1145/3102980.3103006. URL: https://doi.org/10.1145/3102980.3103006.

[3] Samantha Miller et al. “Practical Safe Linux Kernel Extensibility”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS ’19. Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 170–176. ISBN: 9781450367271. DOI: 10.1145/3317550.3321429. URL: https://doi.org/10.1145/3317550.3321429.

[4] B. N. Bershad et al. “Extensibility Safety and Performance in the SPIN Operating System”. In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 267–283. ISSN: 0163-5980. DOI: 10.1145/224057.224077. URL: https://doi.org/10.1145/224057.224077.

[5] Galen C. Hunt and James R. Larus. “Singularity: Rethinking the Software Stack”. In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424. URL: https://doi.org/10.1145/1243418.1243424.

[6] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. “The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 89–105. ISBN: 9781931971478.

[7] Johannes Lundberg. “Safe Kernel Programming with Rust”. MA thesis. KTH, Software and Computer systems, SCS, 2018, p. 56.

[8] *Linux Kernel vulnerability Statistics*. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[9] Simon Ellman. “Writing network drivers in rust”. PhD thesis. B. Sc. Thesis. Technical University of Munich, 2018.

[10] Amit Levy et al. “Ownership is Theft: Experiences Building an Embedded OS in Rust”. In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. PLOS ’15. Monterey, California: Association for Computing Machinery, 2015, pp. 21–26. ISBN: 9781450339421. DOI: 10.1145/2818302.2818306. URL: https://doi.org/10.1145/2818302.2818306.

[11] *Your Next(Gen) OS - redox - your next(gen) OS*. URL: https://www.redox-os.org/.

[12] Jonathan Corbet. *The Rust for Linux project*. Sept. 2021. URL: https://lwn.net/Articles/869145/.

[13] Miguel Ojeda. *Rust-for-Linux*. https://github.com/Rust-for-Linux/linux. 2021.