Developing Device Drivers in Rust

Kyle Christie
B00415210

School of Computing, Engineering
and Physical Sciences

BSc (Honours) Computing Science
University of the West of Scotland

Supervisor: Paul Keir
Moderator: Stephen Devine

Table of Contents

Table of Figures

Acknowledgements

This report is dedicated in memory of my beloved Gran and Great Aunt Helen who I sadly lost while undertaking this project, they always supported me in my work and will be forever missed. I'd like to thank my partner for always supporting and encouraging me in all I do. Thank you to my closest friend, Joshua, for his time, support and games of chess during this project. To my family for supporting me throughout this project. Thank you to my supervisor, Paul, for his endless support throughout this project and to my moderator, Stephen, who previously lectured me for systems programming concepts (the very class in which my success pushed me to research this topic and undertake an Honours project). Finally, thank you to the following industry professionals; Miguel Ojeda, Alex Gaynor, Jonathan Blow & Dave Plummer for giving their individual time and advice for this project.

Abstract

1. Background

1.1 Device Drivers

Device drivers are a vital component of Operating Systems which allow for the control of peripheral devices while interacting with underlying hardware. Drivers also provide facilities which can be used to extend an Operating System via file systems, network protocol, anti-virus capability and more (Ball et al, 2006). Described as the "software layer that lies between applications and physical devices" (Corbet et al, 2005), drivers are clearly a necessity within an Operating System however they suffer from a range of issues with dangerous consequences.

Drivers continue to be programmed with the C programming language which was first developed at Bell Labs between 1969 and 1973, alongside early development of Unix (Ritchie, M.D, 1993). It was designed as a "system implementation language for the nascent Unix operating system" (Ritchie, M. D, 1993). Languages such as C, C++ and Assembly have the potential to be memory unsafe (Gaynor, 2019) which can then lead to critical vulnerabilities as observed by several organisations over the years (Thomas and Gaynor, 2019).

Memory safety is an attribute found within various programming languages with the aim of preventing the developer from introducing certain bugs which strongly relate to memory management (Prossimo, 2022). Memory safety issues usually lead to security problems with common vulnerabilities being out-of-bounds reads, out-of-bounds writes and use-after-frees (Gaynor, 2019).

Furthermore, Linux drivers have seen little to no change within the last two decades. Evidence pointing to this can be found in Linux Device Drivers 3, a book written for Linux Kernel 2.6 (Corbet et al, 2005), where its code examples can compile and successfully run on more recent kernel versions with little to no change. Further evidence supports this point as even online tutorials from 2014 (Karthik M, 2014) continue to compile and run on recent kernel versions. Such examples were built and executed on a small collection of Linux distributions that utilise more recent kernel versions, specifically 4.19.0-17-amd64, 5.15 .0-52-generic and 5.15.67-v7l+. It is therefore clear that device drivers have not evolved in any significant way. Code which targets Linux kernel version from over a decade ago continues to run on more recent versions.

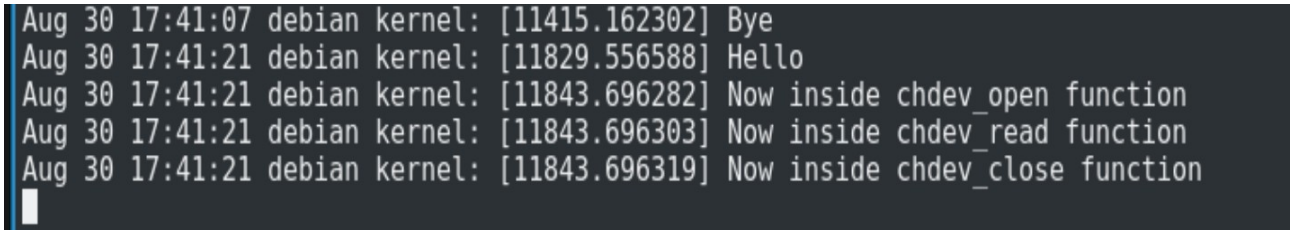

Figure 1: Debian Virtual Machine (Linux kernel 4.19.0-17-amd64 running character driver from Karthik M tutorials.)

1.2 Input from Industry

During the project, prominent or relevant figures within Game Development and Software Engineering were contacted. These include Jonathan Blow (developer of Braid, The Witness and the Jai programming language) Dave Plummer (former Microsoft engineer, now entrepreneur, best known for his work in creating Windows Task Manager among other projects), Alex Gaynor (Software Resilience Engineer working on the Rust-For-Linux project), Miguel Ojeda (Software Engineer working on the Rust-For-Linux project) and Asahi Lina (Software Engineer working on a Linux GPU driver in Rust for the Apple M1).

Blow fed back that drivers are a good problem to consider though felt that the project was not ambitious enough. He spoke of a concept called an 'Exo-kernel model' which he feels would fix a lot of device driver problems as drivers are simply considered 'normal' code. He also feels that the field is held back as the basic idea of what an operating system should be has already been figured out, Blow seems to think there is little innovation in the field. Alongside his feedback, he also provided me with a USENIX ATC Keynote; 'It's Time for Operating Systems to Rediscover Hardware'.

Plummer explained that he had mostly worked on cache software and filesystems but had never really worked on drivers unless it was necessary. He recommended a textbook, 'Windows 7 Device Driver' by Ronald D. Reeves Ph.D. and gave his best wishes.

Gaynor praised the project, saying "Building a USB mouse driver with Rust for Linux sounds like a great project", and outlined the approach used by the RustForLinux community when writing Drivers which is as follows;

1. Check if RustForLinux already has existing APIs for the relevant kernel subsystem.
2. If not, they design a safe Rust API that exposes the original kernel APIs.
3. It is then possible to use these new abstractiosn to write the original driver.

Gaynor mentioned a pull request for the Rust-For-Linux repository which aims to add support for USB device drivers. After a short correspondence, Ojeda provided some advice and useful resources. He recommended getting used to reading C code within kernel. With regards to the Rust

driver, his recommendation was to write a C version which can then be referenced for the Rust driver, this will also help in learning kernel APIs and verify whether issues are a result of Rust support or otherwise.

1.3 Project goal

The aim of this project is to try and overcome the previously highlighted issues by developing a Linux device driver in Rust. Not only will it replace C, Rust and its features should prevent issues with memory safety. Rust is a relatively young language with several benefits and features that aim to improve memory safety. It continues to spread through industry as it was recently incorporated into the Linux Kernel from version 6.1 (Vaughan-Nichols, 2022) and there have been public calls from developers for Rust to be utilised more. An example of this being Microsoft Azure CTO, Mark Russinovich, urging the industry (regarding to C and C++) 'For the sake of security and reliability, the industry should declare those languages as deprecated.' (Claburn, 2022).

1.4 General Concepts

1.4.1 Kernel

A kernel is the primary interface between hardware and computer processes, ensuring resources are used as effectively as possible (Baeldung, 2022). The kernel runs within the operating system and controls the function of hardware alongside managing memory and computer peripherals.

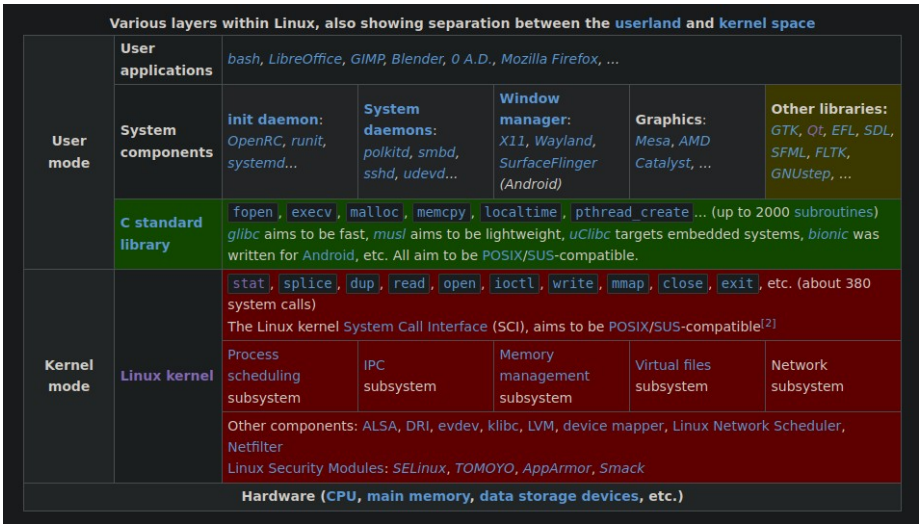

Figure 2: System layer break down of Linux (Wikipedia, 2022)

1.4.2 User space and Kernel space

Kernel space is an area of memory used exclusively by the kernel and encapsulates device drivers (Wikipedia, 2022). User space is a separate area where user applications run and file systems can be managed (Baeldung, 2022). User applications communicate with the kernel via system calls. Kernel and user space is separated to protect memory and protect the hardware layer of the system as

showcased in Figure x. The term 'space' is interchangeable with the term 'mode' as this concept is also related to processors.

1.4.3 Device node system

Device nodes are special file types (especially in Unix-based systems) which represent a resource allocated by the kernel. These resources are identified by a major number and minor number which are both stored within the structure of the node. Typically, the major number identifies the device driver while the minor number identifies a specific device (or collection of devices) that the driver can control with these numbers being passed to the driver.

1.4.4 Build system

Building refers to the process of converting (or translating) source code into executable binary files. Thus, a build system is "a collection of software tools used to facilitate the build process" (Zhang, 2020). Build systems have been used for over 30 years and have not seen major change. There are several build systems that can be used in the present including: Make, GNU Make, CMake, QMake and Ninja.

1.4.5 Makefile

A makefile controls the build system, 'make', and "describes the relationships among files in your program and provides commands for updating each file." (2022, Free Software Foundation, Inc.). The makefile contains a pre-written set of rules which control how make compiles and links a program. It is also possible to run various external commands via make, a clean-up operation as an example. When the makefile is run, it typically builds a piece of software, even updating specific files when additions are made.

1.4.6 Development

Driver development can occur across 2 different machines. As is the case in Windows and, previously, Apple systems. In such scenarios, the debugger and driver are run on separate computers (Microsoft, 2022). The computer running the debugger can be known as the 'host' machine while the computer running the driver can be known as the 'target' or 'test' machine.

2. Literature Review

In this section, various technologies, technical topics and concepts are discussed ranging from the core focus of this project to alternative concepts that could be employed to achieve the same goals in a more efficient manner. Within these discussions, various points are highlighted in terms of the relevant advantages and disadvantages of a given topic.

2.1 Operating System Drivers

2.1.1 Linux

Linux utilises 'kernel modules' to implement most device drivers, these kernel modules are primarily written in C (Corbet et al, 2005). These kernel modules make use of kernel space, which is separated from user space though the kernel space itself is not compartmentalised. Kernel modules are typically compiled with makefiles. These convert the code into an object file by calling the kernel build system to link code and object files to '.ko' executables. Linux makes several commands available to add, remove and manage kernel modules which is one of the primary methods of testing custom built device drivers.

Kernel space is separated from user space by the CPU, the Linux kernel runs in the highest CPU level so there is little restriction on what a driver can do this also means that faults that occur within kernel space have a high likelihood of affecting the overall system. Kernel modules can be run in user space but the associated performance issues mean doing this is worthless.

2.1.2 Apple

Apple device drivers encapsulate iPad, iMac and iPhone. Apple recently re-structured their device driver technology, 'Kernel extensions' to implement additional driver types known as 'Driver extensions' and 'System extensions' (Auricchio et al, 2019). The reason behind the change being that Apple developers found several issues with kernel extensions; They are difficult to develop and debug and also pose a risk to data security, privacy and overall reliability. System and driver extensions improve on kernel extensions and thus are easier to develop while improving security and reliability.

A system extension is similar to a kernel extension but is instead a component of an application. They are intended to implement features that need kernel level co-operation, such as custom network behaviour (Apple Developer Documentation, 2022). These extensions run in user space and contain 3 types: Network; Endpoint Security (including Anti-virus and Data loss protection); Driver (USB, Serial, NIC and HID). System extensions can use any framework within the macOS SDK as well as any programming language. System extensions are said to be much easier to debug as they do not pause the kernel, there is no need to restart the machine and it is possible to build, test and debug on a single machine with full debugger support.

Developers also found that the Kernel is an unforgiving environment (Auricchio et al, 2019). That writing and debugging kernel code is difficult, kernel extensions need 2 machines in order to debug which introduces overhead and only has limited debugger support and that kernel extensions are a great risk to security as successful attackers can gain free reign in the kernel while any bug in a kernel extension could also be a critical reliability problem. Kernel extensions only support the C and C++ programming languages.

2.1.3 Windows

Windows also implements the concept of user mode and kernel mode. Microsoft documentation lists various driver types including: Function driver (which communicates directly with the device); Filter driver (which performs auxiliary processing) and Software driver (used when desktop software needs to access something in kernel space). Windows also provides different frameworks which can be used when writing device drivers through Visual Studio such as the User Mode Driver Framework (UMDF) and Kernel Mode Driver Framework (KMDF). Device drivers are still written in C but are usually categorised under C++.

Windows drivers seem to work similarly to those of Linux. Drivers are built into a '.dll' file with accompanying files including '.inf' (which stores driver information). Depending on the driver type, it may be built into a '.sys' file with a '.cat' file which is used by during installation to verify the drivers signature.

2.1.4 FreeBSD

FreeBSD drivers are similar to that of Linux and makes use of similar concepts. Drivers can be statically compiled into the Operating System or loaded via the dynamic kernel linker facility. FreeBSD makes use of the Unix device node system, known as '`/dev`' which is also found in Linux. FreeBSD makes use of KLD to dynamically manage and extend the kernel without rebooting, which is very similar to Linux kernel module commands. '`kldload`', '`kldunload`' and '`kldstat`' are examples of commands used in managing drivers. FreeBSD makes use of 'pseudo-devices' where a driver emulates the behaviour of a device in software without hardware. Drivers on FreeBSD are split into two categories: Character and Network. Character devices are typically used to directly transfer data between a user process and other processes, they are the most common type of driver.

2.2 Rust

Rust is a "compiled, concurrent, safe, systems programming language" (Klabnik, 2016) which was released in 2015. It was originally invented by Graydon Hoare, an employee at Mozilla, who started the project in 2006 which was then adopted by Mozilla in 2010. Rust has several features which are highly attractive especially with regards to drivers and memory safety.

Rust is accompanied by a powerful compiler that makes use of a strong type system and enforces good practices in code. It checks code at compile time so errors can be detected before code is deployed (Li et al, 2019) thus highlighting errors with clear feedback and potential solutions which prevents developers from making common mistakes (Klabnik, 2016). This feature is critically important to drivers, it was previously established that writing device drivers is no easy task as developers previously struggled to work with the Windows XP driver API (Ball et al, 2006). It has also been highlighted that writing C code for the kernel is difficult (Renzelmann and Swift, 2009).

The compiler also disallows unused variables and enforces correct concurrency (Oatman, 2022). If a variable is sent to be owned by a thread or channel, it can no longer be read, and a compiler error occurs if an attempt to read is made. The compiler also forces the developer to handle errors.

Rust is reliable as code is backwards compatible; ensuring old code is always able to compile with new versions of the compiler (Oatman, 2022). Another benefit of this is that old code will continue to benefit from optimisations made to the rust toolchain. Code of all ages will improve and speed up alongside the language itself. A further benefit is a small revolution in code maintenance, some of the most popular crates can be considered 'complete'. In some cases, they have not been updated in a long time, as the code has no issues and is less likely to rot.

As shown in Figure 4, Rust has no defined memory model, utilising simple memory structures comparable to that of JVM, Go and C++11. As there is no garbage collection there is no generational memory or complex substructures. Memory is managed as part of execution, applying the Ownership model during runtime (Sasidharan, 2020).

Rust, of course, implements a stack and dynamic heap within programs. Typically all variables are placed on the stack with the following exceptions: a manually created box; and when a variable size is unknown or grows over time. In these cases, the variable is then allocated to the heap with a pointer to the data placed on the stack. A box is an abstraction that represents a heap-allocated value. In order to manage memory, Rust uses a system of Ownership upheld by three rules which are applied to both the stack and heap:

1. Each value must be owned by a variable
2. There must always be a single owner for a variable at any time
3. When the owner goes out of scope, the value is dropped

These rules are checked at compile-time. Memory management is conducted at runtime with execution which means there is no cost to performance or other miscellaneous overhead. Ownership can be changed with the `move` function. This is performed automatically when a variable is passed to a function or when the variable is re-assigned. The `copy` function is instead used for static primitives.

Rust utilises RAIL - Resource Acquisition is Initialisation - which is enforced when a value is initialised. Under RAIL, the variable own its related resources with its destructor called when the variable goes out of scope, which reduces the need for manual memory management. This concept is borrowed from C++. Rust also implements a system of borrowing where a resource can only ever have one owner at a time, variables can be passed by value or by reference and the Rust borrow-checker enforces ownership rules and ensures references make use of valid objects. (Sasidharan, 2020).

Variables have lifetimes, a concept which is important for the functionality of the ownership system. A variable's lifetime begins at initialisation and ends when it is closed or destroyed. This should not be considered variable scope. The borrow-checker uses this concept at compile time to ensure that all references to an object are valid. It is clear that Rusts implementation of memory management will no doubt help in ensuring memory safety, an important factor for the application of Rust within drivers.

2.2.1 Rust for Linux

Rust for Linux is a project, originally started in 2019 by Miguel Ojeda with the aim of introducing a new system programming language into Linux kernel. Rust would be chosen as it "guarantees no undefined behaviour takes place (as long as unsafe code is sound), particularly in terms of memory management." (Ojeda, 2022) which would eliminate issues such as use-after-free, double free and data races. The project was created as there had long been desire to write Linux kernel code in Rust. Several attempts were made, the earliest being in 2013, though none of these projects provided Rust support from within the kernel. Rust for Linux carries on initial work that was undertaken by Gaynor and Thomas (2019) in an attempt to introduce Rust into the Linux kernel.

There has since been various technical achievements within the project with several organisations from industry approaching Ojeda with interest including Google, Arm, Microsoft and Red Hat as well as private companies. Alongside these companies, academics have also reached out such as researchers at the University of Washington. Work carried out by the Rust for Linux project was recently integrated into the Linux kernel, starting from version 6.1, marking the first time a new programming language has successfully been introduced into the kernel.

2.2.2 Criticisms

Creator of C++, Bjarne Stroustrup, has previously criticised Rust and similar memory safe languages as "every safe language, including Rust, has loopholes allowing unsafe code" (Claburn, 2022) however in Googles security blog, the use of unsafe is described as an "... escape hatch which allows interacting with system resources and non-Rust code." (Vander Stoep, 2022). Klabnik also

previously noted that "fundamentally the machine definitely is not immutable by default" therefore unsafe is important to the language (Klabnik, 2016).

The comparison could be likened to how I/O is implemented in Haskell, a purely functional language where no functions allow for the use of traditional I/O and there are no global variables. Haskell, similarly to Rust and 'unsafe', implements several backdoors for the sake of functionality within the language yet Haskell continues to be used doesn't seem to have met such criticism as Rust has.

It would seem that while the inclusion of 'unsafe' could potentially lead to issues within a safe programming language, it is ultimately deemed a necessity which allows such languages to be used within system environments and permit interactions with other languages while also considering how the hardware itself works. There are other languages, namely Haskell, that have similarly implemented necessary backdoors. However, it should be considered that the use of unsafe requires the developer to be responsible for safety. Stroustrups concerns may be valid although the implementation of unsafe seems to be a necessary trade-off for the language to function as intended.

2.2.3 Google & Android 13

Android 13 has recently seen a significant drop in memory safety vulnerabilities and an associated drop in vulnerability severity with the annual number of memory safety vulnerabilities dropping from 223 to 85 between 2019 and 2022 (Vander Stoep, J. 2022). Memory safety vulnerabilities now account for 35% of Androids total vulnerabilities (previously 76%) with 2022 being the first year where the majority of vulnerabilities are not related to memory safety. This drop coincides with a move away from memory unsafe programming languages with Android 13 being "the first Android release where a majority of new code added to the release is in a memory safe language".

Rust was announced in Android 12 as an alternative to C and C++ with the goal being to shift development of new code to memory safe languages over time. Now, in Android 13, 21% of all new native code is written Rust with approximately 1.5 million total lines of Rust found within Android Open Source Projects across a handful of new features. Google found that "To date, there have been 0 memory safety vulnerabilities discovered in Androids Rust code." it is not expected for this number to remain 0 but is a significant result which suggests that Rust is fulfilling is intended purpose in preventing Androids most common source of vulnerabilities. It's believed that 'it's likely that using Rust has already prevented hundreds of vulnerabilities from reaching production'.

Google also found that the use of Rust allows optimisation of both security and system health with fewer compromises as safety measures typically slow memory-unsafe languages. This usually means developers must make trade-offs between security and performance in adding sandboxing, sanitizers, runtime mitigations, hardware protections which negatively impact code size, memory and performance. It was also found that when compared to other vulnerabilities (which have a well defined scope of impact) Memory safety vulnerabilities are much more versatile. If code execution is obtained in a process, not only is access granted to the specific resource but to everything that the

process can access which provides an attack surface to other processes. "Memory safety vulnerabilities are often flexible enough to allow chaining multiple vulnerabilities together", it was found that the majority of exploit chains abused in Google products use one or more safety vulnerability. Due to the decrease in severe vulnerabilities, there has been an increase in less severe types with around 15% of 2022 vulnerabilities being Denial of Service vulnerabilities which represents a drop in security risk.

2.3 Memory Safety

Memory unsafe languages allow programmers to potentially access memory which is supposed to be outside the bounds of a given data structure (Gaynor, 2019). This is even more detrimental as memory safety vulnerabilities consistently account for the highest percentage of vulnerabilities within large codebases as showcased in the figure below.

Android	>65% of High & Critical security bugs
Android (bluetooth & media components)	90% of vulnerabilities
IOS 12	66.3% of all vulnerabilities
MacOS Mojave	71.5% of all vulnerabilities
Chrome	~70% of serious security bugs
Microsoft	~70% of CVE vulnerabilities
Firefox (CSS subsystem)	73.9% of bugs
Ubuntu kernel	65% of CVEs (In security updates between November and May 2020)

Figure 3: Memory safety error statistics within major codebases (Gaynor, 2020)

These statistics were observed and reproduced across several large code bases (containing millions of lines of code). Each code base was built by a different company, started development at various points in time and applies a different development methodology. The single common property that unites these codebases is that they are written in memory-unsafe programming language such as C or C++. Gaynor concludes that the magnitude of memory-unsafe vulnerabilities is higher than memory-safe vulnerabilities and that the research supports the notion that the use of memory-safe languages would critically reduce the total number of vulnerabilities.

In the case of data structures, memory unsafe languages allow programmers to access memory which is supposed to outside the bounds of a given data structure. For instance, an array is able to access an element that doesn't exist. This means that the program fetches whatever happens to be at that position in memory. When this is the case in a memory safe language, an error is thrown which forces the program to crash.

As an example, we can consider a program that manages to-do lists for several users. If implemented in a memory unsafe language, it is possible for the programs data structure to both

access negative elements and positive elements that don't exist thus the data structure can access data which is outside of its bounds. This can lead to users having the ability to read each others lists which would then be a security vulnerability in the program, this is known as an 'out-of-bounds read'. If users were able to change elements in other users lists, this is known as an 'out-of-bounds write'. If a to-do list is deleted and later requested then a memory unsafe language has the ability to fetch the memory that it was previously finished with. Within the program, this space might now contain another users list, this is known as a 'user-after-free' vulnerability.

2.3.1 Garbage Collection

Garbage collection refers to automatic memory management which is carried out by what is known as a garbage collector. It can also be described as a "memory recovery feature" which is "built into programming languages" (Sheldon, 2022). A programming language which uses a garbage collector may utilise many collectors with the aim of freeing memory allocated to objects that are no longer in use or required by the program thus the free memory can be re-used for future object allocations. Garbage collection is utilised in several programming languages including: Java, C# and D.

D is a systems programming language that utilises garbage collection. The developer allocates memory as needed and from time to time the garbage collector will free unused memory, making such memory freely available once again. D garbage collection is carried out as follows;

1. All other threads are stopped and the current thread is hijacked for garbage collection.
2. Root memory ranges are scanned for pointers to allocated memory, this memory itself is recursively scanned for more pointers.
3. All memory that holds no active pointers is freed with unreachable memory requiring destructors queued.
4. All other threads are resumed and destructors run for all queued memory, any remaining unreachable memory is freed.
5. Current thread is returned to previous work.

Garbage collection has several benefits such as ensuring a program doesn't exceed allocated memory, ensuring continued functionality and taking responsibility from developers who would otherwise need to manually manage such memory thus reducing the likelihood of memory-related bugs. Specifically in the case of D, it was found that garbage collected programs are often faster, can't suffer from memory leaks (thus have more long term stability) and are faster to develop and debug (explicit de-allocation code is not developed, debugged, tested or maintained) (D Language Foundation, 2022).

2.3.2 Reference Counting

Reference counting is a mechanism applied in garbage collection. This mechanism works by counting the number of references to a block of memory or object from other blocks. A reference count holds the number of references. The count is increased when memory or a reference to the object is created and decreased when a pointer to the memory is de-allocated or destroyed. Upon the count reaching 0, it is clear that there are no pointer references thus the memory is considered unreachable and should be reclaimed as garbage.

Reference counting has several advantages including being easy to implement, reclaiming objects as soon as they become garbage, quick return of system resources (especially if objects support destructors) and ensuring that garbage collection is distributed throughout all the execution period (these means no system freezes, especially in interactive systems). Although there are several disadvantages, namely the addition of significant bloat within code as each assignment will see a call which updates the reference count. In multithreaded systems, the reference count becomes a potential problem as locks must be used to update the reference count. It should also be noted that re-used of optimised atomic operations can still be costly when used repeatedly and the header space used within reference counting has a high cost, when causes significant overhead especially when used with small objects.

2.4 The Exo-kernel

The exo-kernel is a concept originally developed at MIT that attempts to return management of hardware resources to the application itself. This kernel is designed in a way that separates resource protection from resource management in order to allow applications to customise how they interact with underlying resources thus the application is completely in charge of its own paging, scheduling, context switching and handling of page faults.

In present designs, the Operating System is positioned between applications and the physical hardware which impacts performance as well as functionality and scope of applications. The exo-kernel philosophy looks to force as little abstractions as possible, exposing hardware where possible. The kernel itself is small, all hardware abstractions are moved into untrusted OS libraries in order to ensure there is no forced abstraction though components such as POSIX are still available if required by a given application.

Such kernel would provide several features including improved support to application control (as security is separated from management) and the availability of a low-level interface. Abstractions are securely converted intro libraries, and offer high portability and compatibility. This design and its underlying features are complimented by several benefits which include improved performance in applications, more efficiency when using hardware (due to precise resource management), development and testing is simplified alongside each application having the ability to apply its own optimised memory management. It should be noted that the main drawbacks of the exo-kernel are less consistency and a more complex design in the kernels interfaces.

The exo-kernel is an example of how changes in operating system design & implementation (OSDI) may be leveraged in order to improve device drivers. It was previously found that OSDI has stagnated and, similarly to device drivers, does not see much work or research. It was previously described as 'hugely rich design space' (Roscoe, 2021) that has 'very little published work'. Roscoe, in his USENIX keynote, declared that OSDI doesn't have the priority that it should and that the design of operating systems is in fact, affecting how hardware is designed to the detriment of both the operating system and hardware (almost like a feedback loop). It may be that some of the problems we observe with device drivers could be related to the aforementioned stagnations in OSDI and that new research in the field (or making use of new concepts such as the exo-kernel) could lead to an improvement within drivers.

2.6 Summary

To summarise, Rust clearly reduces problems related to memory safety. This can be observed in its application in Android 13 as its use has reduced the total number of vulnerabilities from 76% to 35%, marking the first year where the majority of vulnerabilities are not memory safety. Alongside this improvement, the Rust code found within Android has, to date, not suffered from any memory safety vulnerabilities proving that Rust is able to reduce and prevent issues related to memory safety within a codebase. With these results, the various benefits (and main attractions) of Rust are further consolidated including its strong type system, compiler (and its enforcement of good programming practice), ownership model, memory management, borrow system and lifetime system.

Memory safety is a clear problem, especially in large codebases that make use of unsafe languages (such as C, C++ and Assembly). Memory safety vulnerabilities typically account for the highest percentage of vulnerabilities within large codebases across Apple, Microsoft, Google and Linux. It was found by Gaynor that the magnitude of memory-unsafe vulnerabilities are higher than that of memory-safe vulnerabilities and that the use of memory-safe languages would reduce the total number of such vulnerabilities (as proven in findings from Android 13). There are various vulnerabilities that can stem from issues with memory safety including out-of-bounds read/write, use-after-free and double free. Garbage collection is another system that may help with memory safety issues though it is not perfect and may even negatively impact performance within software.

Finally, the exo-kernel is an alternative solution that may be explored to solve issues with device drivers. Rather than approach the problem from a language or tool, this approach considers the wider system and its overall design with the main philosophy being to remove as many abstractions as possible and allow applications themselves to implement their own memory management (from paging to fault handling). Alongside discussing the concept of the exo-kernel, it is also necessary to discuss stagnations in operating system design & implementation. OSDI, was found to be a rich design space which has little research work, it seems to be of little priority (similarly to drivers) and OS design may in fact be affecting how the underlying hardware is designed. Nevertheless, I mprovements to drivers may stem from improvements made to overall operating system design.

3. Development

Within this section lies a discussion of the development undertaken during this project. From C drivers to Rust software and drivers, these technologies are each discussed both individually and comparatively, highlighting the difference and even similarities between Rust and C. Alongside this are more technical discussions with regards to compilation and related information.

3.1 Writing C drivers

Before working on Rust drivers, it is necessary to consider their predecessor, C drivers. C being the primary language employed in the Linux kernel and, as previously discussed, was created between 1969 and 1974 alongside Unix. C accounts for 98.5% of the code written for Linux (Torvalds, 2023). It should be noted that Linux drivers can often be referred to as 'kernel modules', though within this context they will be referred to as drivers for simplicity.

It should be noted that kernel modules also implement extensions such as filesystems which is why they are not strictly labelled as 'drivers'. As such, filesystem modules are not considered to be a driver in the traditional sense and can rather be considered to be a software driver as it maps low-level data structures to higher-level data structures. While a filesystem extension indeed implements lower level system calls for file access and mapping functionality, the interface which it utilises is separate from that used to facilitate the physical transfer to the disk which is, instead, carried out by a block device driver. Thus kernel modules can also be used to facilitate core but broad functionality such as providing the low-level functions which are required for the functionality of standard filesystems.

3.1.1 Driver types

There are different types of drivers available for implementation within the Linux kernel, such types are flexible but are categorised into the following classes: Character, Block and Network. Character typically works with a stream of bytes by using file functions such as open, close, read and write. and by working with file system nodes. Unlike a regular file, character devices can only be accessed sequentially. Examples of a character device can be found in text consoles and serial ports.

A block driver typically works with any device that can host a filesystem. Linux itself reads and writes block devices as if they were a character device thus the only difference between block and character devices is the way that data is managed and they utilise radically different kernel interfaces.

A network driver controls a network interface (whether hardware or software) which itself conducts network transactions between different hosts. The driver itself mostly uses the network subsystem within the kernel and is typically in charge of sending and receiving network packets, the driver is typically designed with a focus on packet transmission rather than focusing on individual connections. Unlike the aforementioned driver classes, the network does not utilise a stream and is

not mapped to a filesystem node (though a unique name is still assigned), thus function calls for packet transmission are made rather than calls to read and write.

3.1.2 Build steps

C is the main language used within Linux drivers with several, if not all, driver subsystems written in C. Driver source code is stored in '`.c`' files, similarly to traditional programs. They can either be represented in a single file or across multiple interconnecting files which contribute to make a single driver. These files can be found alongside a 'makefile' which is used to create the executable format of the driver, in way of the '`.ko`' file type, which is the object code of the driver. If a recompiled or customised kernel is not available on the system, it is then necessary to download and install kernel headers for the specific kernel version being used this can be accomplished by running the following command '`sudo apt install linux-headers-$(uname -r)`' in '`apt-get`' which will automatically handle installation and make the kernel header files available, typically found in '`/usr/src/`' under '`linux-headers-`*`kernel-version`*' (thus an example could be '`/usr/src/linux-headers-5.8.0-63-generic`'). With this, it should be noted that the driver is specifically compiled for the given kernel version thus it should recompiled for other kernel versions as necessary.

After running make, the executable driver is produced alongside several other files. With this, it is now possible for the driver to be used within the kernel. Several commands are available to use and interact with a driver, '`lsmod`' is an example which lists all drivers and their statuses within the kernel. '`insmod`' and '`rmmod`' can both be utilised to link and unlink '`.ko`' files to the current kernel. '`dmesg`' can be used to display messages from within the kernel and can act as a form of debugging when loading and running a driver. '`modprobe`' can also be used to check and automatically debug code before insertion. Such commands form the basis of introducing, using and testing drivers within a Linux kernel and are commonly used during development.

3.1.3 'Hello, World.' driver

'Hello, World.' is the common introduction to programming in any language, this also applies to drivers. A simple 'Hello, World' driver can easily be created and serves to showcase key concepts of Linux drivers and how they differ to traditional programs. As expected in C, the first 'include' lines represent headers and libraries and are commonly used to call various subsystems that can be used within the driver. In this example, libraries ('`linux/init`' and '`linux/module`') can be observed that provide core components for drivers.

Following this is a macro function that declares the license utilised by the driver. This function is usually a necessity and without it, issues may be encountered regarding compilation and loading dependant on whether compilation occurs via headers or a recompiled kernel.

Next, are the functions utilised by the driver, in this case we simply have '`hello_init`' and '`hello_exit`'. Drivers typically make use of initialiser and exit functions to carry out pre-configurations before the driver runs and complete all de-allocations before unloading the driver. Such functions are present within all drivers and are executed upon inserting and unloading. Finally, we have macro functions in the form of '`module_init`' and '`module_exit`' which declare the initialiser and exit functions so these can be suitably called and used.

3.1.4 Character driver

The character driver, as found in Appendix APPEND can be used to further introduce driver programming concepts. In this case, the `struct file_operations` is introduced which controls how the driver interacts with files. Such structures are typically used to store and act as a format to declare key functions throughout several kernel subsystems. Alongside this new data structure is the introduction of driver registration. Driver registration refers to the process of driver software being assigned a major and minor number as well as a physical device file. It can also be observed that the driver utilises the filesystem - '`fs`'- subsystem.

Drivers are registered to a device entry file by running the following command: `sudo mknod -m 666 /dev/"DRIVER_NAME" c 240 0` . The first parameter '666' sets the permissions of the given file. Next is the intended name and location of the entry. The 'c' indicates that the driver is of the character class and finally, the major and minor number that should be used for both the device and driver. The '`/dev/`' directory is typically used to hold all device entries to be used by drivers. Major and minor numbers are used to associate drivers to devices. The major number, specifically, is used to associate entries to the driver with the minor number being used to represent the number of instances of that device.

`file_operations` assigns and stores new functions, written by the developer, to the typical file functions: read, write, open, close (in this case, known as 'release') allowing for customised behaviour to be assigned to such functions. The structure also declares the owner of these new functions. Within this example, new replacements have been created in the form of '`chdev_open`', '`chdev_read`', '`chdev_write`' and '`chdev_close`' which simply print a message alongside the function name when that function is called on the device file.

In order to test the functionality of this driver, various commands can be executed on the device file including '`cat`', '`less`', '`more`' and so on which utilise the previously mentioned file functions. After interacting with the device file, the '`dmesg`' command can be run to examine results and verify whether the driver itself is functioning as expected.

3.3 Building Linux with Rust support

3.3.1 Initial Work

Early research and development resulted in the creation of a virtual machine used to test Rust for Linux and Rust integration into the Linux kernel. This machine was built via make with BusyBox used to generate configuration files. Running via the QEMU hypervisor, the system served to provide insight into core concepts for the project such as building via make, enabling and testing Rust support in the kernel.

Upon rust being enabled and restarting the machine, various samples were compiled and available for testing. With this, it was possible to add a new sample entry in the way of a simple echo server. This server simply prints out whatever input it receives to its device entry. After writing a new entry into the necessary kernel configuration files and makefile, the echo server was then compiled and loaded as part of the Rust samples on boot.

3.3.2 Results

Further research resulted in the development being focused on virtual Linux systems created via the VirtualBox hypervisor. With this, the available system was much closer to that of a physical machine and was ultimately more capable when compared to the initial QEMU machine.

A first attempt was made using the Debian distribution however introducing Rust support was largely unsuccessful due to issues with introducing Rust support into a newly compiled kernel, the process was also found to be time-consuming due to long kernel build times on the virtual machine, though this could also be attributed to the availability of less resources. An issue was encountered where a special file, '`target.json`', for the '`rust-analyzer`' component was unable to be produced, this file is used within code editors to generate documentation which is embedded into the Rust code for kernel subsystems thus the file is important for making documentation available to the user throughout development.

Later, a second attempt saw full success. This new machine utilised Ubuntu which was much found to be much easier to recompile on, eventually becoming the primary system for development and testing.

3.3.3 Build Steps

In order to begin building a new Linux kernel with support for Rust, it is necessary to download Linux kernel source code. While these sources are widely available, in the context of this work, it is best to make use of the Rust for Linux GitHub repository. This repository will provide the latest updates and work regarding Rust in the Linux kernel where the official Linux kernel repository does not. The repository can be downloaded (or cloned) with the following command; ’`git clone –depth=1 https://github.com/Rust-for-Linux/linux.git`’. After downloading and unpacking the Rust for Linux kernel sources in a preferred location, it is then

necessary to download and install several packages to the system which are required for building and installing a new kernel. These dependency packages can be installed via ‘`apt`’ or ‘`apt-get`’ and are listed in the following table.

`Apt` Package title	**Dependency name**
make	Make build system
flex	Flex
bison	Bison
lld	LLD LLVM Linker
libelf-dev	ELF access library
libssl-dev	OpenSSL
libclang-dev clang	Clang Compiler
llvm-dev	LLVM

Figure 4: List of dependencies required for building Linux kernel

Make, as previously discussed, is the build system to be used when constructing the kernel. Flex can also be known as ‘fast lexical analyser generator’ and is a tool used to generate programs which conduct text pattern-matching (Ubuntu manuals, 2023). Bison is a “general-purpose parser generator” (Free Software Foundation Inc, 2014) that creates parsers tables for programming languages . LLD is a linker from LLVM which is used to replace typical system linkers to improve performance. The ELF access library is a shared library that facilitates reading and writing of Executable and Linkable Format (ELF) files at a high level, with this specific version is aimed towards development. Clang (and related development files) is used in place of the gcc compiler as it utilises LLVM which is required for Rust to be enabled and compiled within the kernel thus LLVM itself is required. LLVM being a “collection of modular and reusable compiler and toolchain technologies” (LLVM Administration team, 2023).

With dependencies installed, it is then possible to create core configuration files and begin compiling the new kernel source code. It should be noted that it may be necessary to run such commands as either the root user or via ‘`sudo`’. The first command that should be executed is ‘`make LLVM=1 -j4 menuconfig`’ which generates configuration files and utilises the ‘`ncurses`’ library to provide a graphical user interface allowing easy modification of the kernel configuration and related files. To expand on the semantics of the command, ‘`LLVM=1`’ indicates to make that LLVM and Clang should be used. If this parameter were not included then make would use ‘`gcc`’ by default. ‘`-j`’ is used to denote how many CPUs should be utilised when carrying out the command. In this case, make is provided with 4 CPUs though this value can be higher or lower depending on available resources which will then impact the efficiency of the build system.

After modifying the configuration as required, it is then possible to begin building the kernel source code. This can be carried out by running '`make LLVM=1 -j4`' which begins the build process. Depending on the allocated CPUs, such process may require a large amount of time (possibly over an hour). Make utilises this time to traverse all necessary source files, compiling each into an 'object' file which is then linked into the final kernel executable. Upon building the sources, it is then necessary to build all kernel modules and drivers; '`make LLVM -j4 modules`'. This is very similar to the previous command in that make traverses all module sources, compiling them into their executable versions to then be loaded into the kernel. It is here that all module and driver samples are built. It is then necessary to install the previously constructed driver executables, this can be performed with '`make LLVM=1 -j4 modules_install`' which loads all builtin '.ko' files into the kernel.

Finally, it is then possible to install the kernel with '`make LLVM=1 -j4 install`'. In this step, all necessary boot entries are created to load the kernel. If '`grub`' is installed as a bootloader then it will be used by make to generate all boot files and install the new kernel alongside those previous. Great caution should be taken when undertaking these build steps as missing even one will lead to the creation of a new boot entry which is incapable of loading due to missing files or dependencies. In such cases where a step has been missed and a non-functional entry has been generated, it is necessary to revert to an older kernel version and once again build the new kernel beginning from the very first step. If the kernel was successfully built, then Linux will load as expected. Within Linux, it is possible to check and verify the current kernel version by running '`uname -r`' which should now display the new kernel version (for Rust support, typically version 6.1 or above).

With a newly constructed kernel, it is now possible to introduce Rust into our kernel. While files for the Rust area of the kernel are available within the sources, they cannot be used without the Rust compiler. It is therefore necessary to install Rust, it is advised to run this installation as the root user. The Rust setup can obtained via '`curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`' which, upon running, will download and execute the aforementioned setup. It is best to utilise this command as obtaining Rust via other package managers may result in the installation of out-of-date versions. The above command ensures the installation of an up-to-date, supported version of the Rust language.

After installing the Rust language and its compiler, it is then necessary to install additional components of Rust including `clippy`, `rust-docs`, `rust-src` and `rust-fmt`. Each of these can be downloaded and installed via the '`rustup`' command which manages Rust installations, toolchains and components. It is also necessary to switch to a different version of the Rust language, this can be perfomed via '`rustup override set $(scripts/min-tool-version.sh rustc)`'. As the virtual machine is not currently updated to the latest version of Rust for Linux, it makes use of Rust version 1.62.0 where more recent and future versions of the project are increasingly likely to use more updated editions. Alongside the documents, source code and formatting, `clippy` is a component which holds 'lints' in order to "catch common mistakes and improve your Rust code" (The Rust Clippy Developers, 2018). These lints are typically used to improve source code with several contained within the rust compiler. They often produce warnings and error alerts when used (Rust Community, 2023). Clippy can be utilised as necessary by passing '`CLIPPY=1`' within the make command.

With all necessary changes to the Rust compiler, it is then possible to begin introducing Rust support into the Linux kernel. It is first necessary to run ‘`make LLVM=1 -j4 rustavailable`’ which verifies that Rust is available for use within the kernel. Following this various other parameters can be passed such as ‘`rust-analyzer`’ (commonly used to generate documentation suitable for use within IDEs), ‘`rustfmt`’ (which enables and applies Rust formatting within code) and ‘`rustdoc`’ (which is used to generate documentation). With all options and configurations set, it is now possible to recompile the kernel with Rust by, once again, following the previous steps. Upon successfully compiling the new kernel, it is then possible to build kernel modules and drivers with Rust.

3.3.4 Building on physical hardware

Attempts were made throughout development to build a Rust-enabled Linux kernel on physical hardware. One such machine was a Raspberry Pi 400 model. It was eventually found that the aforementioned method of rebuilding a Linux kernel is not the most optimal method with regards to the Raspberry Pi. The built in command '`rpi-update next`' was instead used to build Linux 6.1 on the machine however this, too, was unsuccessful as the kernel reverted to its previous version on restarting the machine. It was also found that while the new kernel could be built and temporarily utilised, it was not possible to enable Rust support therefore it was ultimately not possible to utilise the raspberry pi within development.

Alongside the Raspberry Pi, an attempt was made to build the Rust for Linux kernel on a workstation which was much more similar to the systems created in VirtualBox. Similarly to the Pi, the new kernel would build and install though issues were encountered when attempting to rebuild the kernel with Rust support. As neither attempt to enable Rust support on the Raspberry Pi or workstation was successful, it was necessary to return to working on the functional VirtualBox instance.

4. Experiments

4.1 Rust applications

Various applications were written in Rust throughout this project, from a simple guessing game to the use of Unix domain sockets and calling C in Rust via '`unsafe`'. These programs were written with a focus on learning Rust fundamentals while gaining experience with the language itself.

4.1.1 Guessing Game

A program where the user must guess a secret number, this example (found in Appendix APPEND) can be used to demonstrate the similarities and differences between Rust and C. While the overall structure is comparable to that of C, there are distinct changes in syntax that can be observed including the use of '`let`' and '`loop`' rather than the C method of using the name of the declared type and '`for`', '`while`' or '`do`'.

As expected, library calls can be found at the start of the program but simply consist of the keyword 'use' alongside a C++ style library path. This may considered an improvement as the library calls are much more brief and clear where, in comparison, C library calls are slightly more cluttered using a special character alongside a string.

It can also be observed that where C utilises a function - '`printf`' - for output, Rust makes use of a macro function (denoted by the use of the exclamation mark). Gathering input also differs, with Rusts '`stdin`' calling both '`read_line`' to read the input and '`expect`' to simplify error handling. This case of error handling may be considered more convenient than that of C++ with its try-catch-finally block and the C approach of checking an error code with an '`if`' statement.

4.1.2 BMI Calculator

A simple application to calculate a given BMI, previously discussed features can once again be observed. A difference in type keywords can also be observed in the use of '`f64`' which represents a 64-bit floating point value. It is also possible to observe Rusts format specifier in the way of '`{:.64}`'.

Within this program, it is also possible to observe how Rusts enforced variable mutability affects the program itself. Unlike most programming languages, where variables are mutable by default, Rust variables are immutable therefore mutable data must be specified via the '`mut`' keyword. As the string variable 'input' is mutable, it is then necessary to clear the stored data each time that new data should be saved which can observed after the 'height' variable is created. This is dissimilar to many other programming languages, which typically implement opposite behaviour.

As the primary focus of Rust is memory safety, the presence of these features is likely aimed at preventing common issues such as overwriting with calls to '`clear`' therefore consolidating security by ensuring proper cleansing of mutable data.

4.1.3 Calculator

Within this program, which simply evaluates a calculation given by the user, it is possible to observe how 'crates' are applied. A crate refers to an external Rust library, such crates can be specified within '`cargo.toml`' under the dependencies section. The previously mentioned file can be compared to a windows '`.ini`' file as it holds information and configurations for a given Rust project. This file is also utilised by Cargo to properly install and manage related dependencies.

Within Rust, a crate can be declared with '`extern crate`', as observed in Appendix APPEND. With this call, it is then possible to utilise the crate. In this example, the '`meval`' crate is used to evaluate the calculation provided by the user. Crates and related information can be found via a resource named 'crates.io'. This resource provides information about crates and their functionality alongside installation steps, license information, crate owners, documentation, previous versions, dependent crates and more. The use of crates alongside the cargo tool is clearly a highly attractive feature of Rust as library installation and management is much more simple.

4.1.4 Unix Domain Sockets

This Unix exclusive project consists of a client and server source files. The client connects to the server and simply awaits user input, which is written to the server. The server is responsible for dispatching new threads to handle client connections and reading from the client, printing data to the terminal.

Within the server program, the '`match`' block can be observed which serves to provide pattern matching. This mechanism is somewhat similar to a '`switch`'. In this case it is used for error handling by matching the return value of the '`stream`' type. If the given '`stream`' returns '`Ok`', this is matched and a thread to handle the client is produced. If an error is returned then the given error handling, in this case a short message and program halt, is called.

Within each program, the server name is declared alongside the library calls as '`common::SERVER_NAME`'. This call is possible due to the presence of '`mod common`' where '`common.rs`' is an additional source file in the project directory. This source file simply declares a public static string which is the server name that will be provided to both the client and server.

4.1.5 Calling Unsafe C

This program declares a C library, '`libbadmath`', which is called within Rust via '`unsafe`'. To use each language within the same project, it is necessary to utilise cmake to properly build the project. Cmake can simply be called as a dependency within the projects '`cargo.toml`'. The C library can be placed within its own directory containing its source code and '`CMakeLists.txt`' which details build steps for the library. The library source code is extremely simple, containing a single float function '`bad_add`'.

Next, within Rust, the link to C can be observed in the form of the '`extern`' block, which declares the library function, as well as the '`#[link]`' attribute, responsible for linking the library to the Rust source. Within the main function, the application of '`unsafe`' can be observed where the '`bad_add`' C function is called from Rust. Alongside the source files, it is also necessary to write '`build.rs`'. This file calls the cmake crate and is responsible for the automatic building of the C library and printing of build information.

4.2 Rust drivers

4.2.1 ‘Hello, World.’

It should be noted there are, in fact, no additional or differing steps required in order to build a Rust driver. So the same steps described for C drivers can generally be used with Rust with some minor changes such as the makefile.

Found in Appendix APPEND, the Rust 'Hello, World.' driver structure is loosely similar to that of its equivalent in C. First, is the library calls as expected in Rust. Next, is a structure known as the module descriptor. This data structure replaces equivalent macro functions utilised in C including the driver name, author, description and license. Following this is the '`RustMouse`' structure which, in this case, is unused but must be declared as it represents the driver itself. Next is a structure to hold our driver functions which holds our initialiser which simply prints 'Hello, World!' before loading the rest of the driver. Finally, is the '`Drop`' function which automatically deallocates the driver and related resources after printing a 'Goodbye!' message.

In comparison to C, this Rust driver is much more compartmentalised as can be observed with the license and author macro functions reimplemented as its own structure. However, the functions themselves are encapsulated within '`impl`' structures which is not as clean as C which has no such feature. In an improvement over the C version, the Rust driver does not require the use of macro functions to declare initialiser and exit functions. Overall, Rust seems to have cleaned up driver code where such a change is most needed but consequently uses '`impl`' to hold functions which may almost be viewed as negating the previous improvement.

4.2.2 Character driver

A rust equivalent of the character driver example, this driver further highlights benefits of Rust code alongside Rust implementations of driver concepts. As always, calls to libraries can be observed first with calls to several subsystems compacted into a single line via brackets. Following the typical module information structure, the driver structure can be observed which holds a declaration of the drivers entry file. Next, is the Rust implementation of the previously discussed file operations structure which holds the new open, read and write functions. This is prepended by `#[vtable]`, a macro used in creating virtual tables which allow the use of traits with the `bindgen` system.

The `bindgen` system is important as it is used to automatically generate language bindings between the Rust foreign function interface (FFI) and C and C++ libraries (Wong et al, 2023). Within the Rust for Linux project, bindings created via `bindgen` are used in order for C to be called from Rust. These bindings declare C functions and types within Rust facilitating the aforementioned behaviour. With this, Rust code that wraps kernel functionality from C is known as an ‘abstraction’ (Ojeda, 2022).

Finally, within the kernel module structure lies the initialiser function which, alongside other actions, registers the driver and sets which name should be utilised (which in this case is 'scull').

4.2.3 USB driver

Research eventually became focused on developing a Generic USB Mouse driver, delivering on the proposed Linux driver as outlined in the project goal. Ultimately, it was found that such a driver is not currently feasible to develop using the Rust for Linux project. At the time of writing, not all kernel subsystems are fully implemented within the Rust for Linux project. The USB subsystem that would be required for such a driver is not yet included though there is work being conducted which has resulted in a USB sample written in full Rust (Rodriguez, 2023) this has yet to be merged into the main Rust for Linux repository.

5. Conclusion

The addition of Rust into the Linux kernel has clearly brought a much needed change, as was previously highlighted. While not all subsystems and features from the C side of the kernel are available, constant progress and evolution driven by the Rust for Linux community means the new language will inevitably reach a stage where it is on par with (and, with time, may supersede) its predecessor. It is also clear that the Rust language is fulfilling its objectives as seen in Android 13 where the adoption of Rust resulted in a significant drop in memory safety vulnerabilities. This also proves Gaynors belief that the use of unsafe languages brings significant risk and his conclusion that memory safe alternatives to such languages should be considered (Gaynor, 2019). This showcases clearly that the adoption of safe languages, such as Rust, serves to benefit software by reducing the potential of memory safety issues through a plethora of attractive features which will no doubt go on to benefit a multitude of future projects. As was previously discussed, the risk related to memory safety is high and is most prevalent in unsafe programming languages. Such risks are made clear as it was highlighted that memory safety accounts for the highest percentage of vulnerabilities within a given large-scale codebase. Furthermore, the threats posed by such vulnerabilities are high in comparison to other types of vulnerability thus it is clear that the problem of memory safety must be mitigated.

While Rust can be observed as being an effective solution to combat memory safety vulnerabilities, it is nevertheless necessary to consider other methods. There is potential that a revolution in the wider design of operating systems could contribute to such issues. Operating system design as a field could actually been seen as falling behind device drivers, it is clear that the field is stagnating and in dire need of improvement. The exo-kernel and its liberation of hardware access and control may be attractive, however the cost of re-design and re-implementation is likely too high, this design is likely best suited to fresh, new operating systems. Garbage collection may be another such alternative though it should be noted that the negative impacts to performance introduced by the additional code are unlikely to benefit drivers, instead acting as a hindrance therefore this method would likely require specific adjustment before seeing utilisation in the driver and kernel space.

Ultimately, Rust has an extremely bright and beneficial future within the Linux kernel. However, with its adoption still recent, it is yet to properly implement all kernel features and subsystems. Linux will eventually benefit from the inclusion of Rust, this is clear from previous examples, however it is also clear that the implementation is yet to reach a point where the Rust language can be used practically and widely within the kernel and drivers.

6. Reflection

References

Appendices