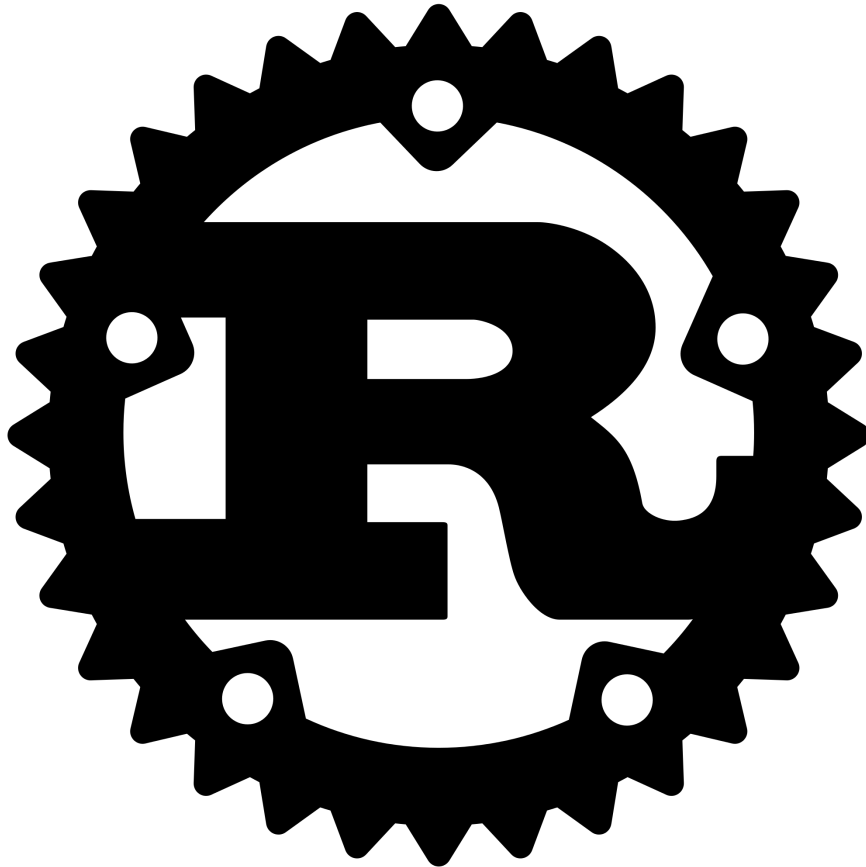


Oxidizing the Kernel

Studying security and performance implications of introducing the Rust language to the Linux kernel

Master's thesis in Computer science and engineering

Joakim Hulthe
Vidar Magnusson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Master's thesis 2022

Oxidizing the Kernel

Studying security and performance implications of introducing the Rust language to the Linux kernel

Joakim Hulthe
Vidar Magnusson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Oxidizing the Kernel
Studying security and performance implications of introducing the Rust language to the Linux kernel
Joakim Hulthe
Vidar Magnusson

© Joakim Hulthe & Vidar Magnusson, 2022.

Supervisor: Miquel Pericas, Department of Computer Science and Engineering
Examiner: Vincenzo Massimiliano Gulisano, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The Rust logo, used under Creative Commons Attribution license:
`https://foundation.rust-lang.org/policies/logo-policy-and-media-guide/`

Typeset in LaTeX
Gothenburg, Sweden 2022

Oxidizing the Kernel
Studying security and performance implications of introducing the Rust language to the Linux kernel
Joakim Hulthe
Vidar Magnusson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Linux operating system is one of the most significant and used projects globally. It is written in the C programming language, which has become almost the sole language for systems-level programming. C has achieved this status by providing direct and often complete control over the underlying hardware and memory to the developer. This allows the developer a lot of freedom which, for example, can be used to optimize execution performance to a high degree. However, this access is easily misused, which has led to several common bug patterns within C programs such as Linux. Particularly several of these bug patterns concern memory-safety such as buffer-overflows, double-free and use-after-free. *Rust* is a newer language which aims to operate on the same level as C but with compile-time protections against these memory-safety issues without sacrificing run-time performance.

This project aims to evaluate the viability of using Rust in the Linux kernel. To accomplish this, we have rewritten a read-only version of the exFAT file system driver in Rust and evaluated it in terms of security and performance. The security evaluation was split into two parts. For the first part, we have tried to determine the scope of memory-safety related issues in the kernel by looking at previous vulnerabilities. For the second, we have studied usages of the unsafe keyword, a way of circumventing the rules of Rust to perform memory-unsafe actions such as reading from random memory, in the implemented driver. A way of circumventing the rules of Rust to perform certain operations. Performance was measured using various benchmarking tools comparing the execution times of different systems calls in the two implementations.

Using these evaluations, we have been able to find that Rust is well suited to improve the security of Linux, with potentially 72% of all studied vulnerabilities being preventable by Rust. Furthermore, we have found that Rust can keep up with C in terms of performance being as fast or just slightly slower for the studied system calls.

Keywords: Linux, Kernel, Rust, Performance, Memory-safety

Acknowledgements

We would like to extend a huge thanks to our supervisor Miquel Pericas as well as our examiner Vincenzo Massimiliano Gulisano for taking time out of their busy schedules to assist us throughout the project. Furthermore, we would like to thank the contributors of the Rust for Linux project, without which much of this project would not have been possible. Finally, we would like to thank our office colleagues for providing constant support and motivation throughout the project.

Joakim Hulthe, Vidar Magnusson
Gothenburg, October 2022

Contents

List of Acronyms **xi**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Problem Statement 2

2 Background **3**
2.1 The Rust Programming Language 3
2.1.1 Ownership, references, and borrowing 3
2.1.2 Unsafe 4
2.1.3 Arrays, slices, & fat pointers 5
2.1.4 Algebraic data types 6
2.1.5 Generics 6
2.1.6 Traits 7
2.1.7 Memory layout 8
2.1.8 Iterators 8
2.2 Memory-safety of Rust 9
2.3 Rust For Linux 9
2.4 The exFAT file system 10
2.4.1 Boot regions 10
2.4.2 Data region 10
2.4.3 File Allocation Table 11
2.5 Structure of the Linux exFAT driver 12

3 Methods **13**
3.1 Implementation of a driver in Rust 13
3.2 Security 13
3.3 Performance 14

4 Implementation **17**
4.1 Structure of the driver 17
4.2 Data structures 17
4.2.1 Iterators 18
4.2.2 Algebraic data types 18

4.2.3 Locking . . . 20
4.2.4 Reading data from memory . . . 21
4.2.5 Deviating from the specification . . . 23

5 Evaluation **25**
5.1 Memory-safety in Linux . . . 25
5.1.1 Out of bounds read/write . . . 25
5.1.2 Use-after-free . . . 26
5.1.3 Null pointer dereference . . . 26
5.1.4 Non memory-safety related CVEs . . . 27
5.2 Usages of unsafe within the implemented driver . . . 27
5.2.1 Unsafe Rust functions & macros . . . 28
5.2.1.1 Zeroed macro . . . 28
5.2.1.2 Boxed values . . . 28
5.2.1.3 Spinlocks and Mutexes . . . 29
5.2.1.4 KMemCache . . . 29
5.2.1.5 Linked lists . . . 29
5.2.1.6 Built-in functions . . . 29
5.2.1.7 Methods on raw pointers . . . 30
5.3 Performance . . . 30
5.3.1 Macros & inlined functions . . . 30
5.3.2 Bounds checking . . . 31
5.3.3 Benchmarking the driver . . . 31
5.3.3.1 Performance of `getdents` . . . 32
5.3.3.2 Performance of `stat` . . . 32
5.3.3.3 Performance of `read` . . . 32
5.3.3.4 Summary of benchmarks . . . 32

6 Discussion **35**
6.1 Security . . . 35
6.2 Performance . . . 36

7 Related Work **39**
7.1 Memory-safety of Rust . . . 39
7.2 Performance of Rust . . . 40
7.3 Rust for other operating systems . . . 40
7.4 Other languages for Operating system development . . . 41
7.5 Security of the Linux operating system . . . 42
7.6 Static checking of memory-safety . . . 43

8 Conclusion **45**

Bibliography **47**

A Structure of the existing Linux exFAT driver **I**

B CVE Classifications **III**

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ADT	Algebraic Data Type
CVE	Common Vulnerability and Exposure
exFAT	Extensible File Allocation Table
FAT	File Allocation Table
FFI	Foreign Function Interface
IoT	Internet of Things
MSRC	Microsoft Security Response Center
MAC	Mandatory Access Control

List of Figures

2.1 Example of ownership in Rust. 4
2.2 Example of an unsafe block and an unsafe function in Rust. 5
2.3 Example of enum in Rust. 6
2.4 Example usage of enum from Figure 2.3. 6
2.5 Example of generics in Rust. 7
2.6 Example implementation of the `Default` trait in Rust. 8

4.1 Find dir in the Rust implementation of exFAT. 18
4.2 C implementation of the exFAT directory entry values, data values omitted for brevity. 19
4.3 Rust implementation of the exFAT directory entry values, the values of variants are structs defined separately. 20
4.4 BufferHead struct in the Rust implementation. 21
4.6 The `bytes` method for the BufferHead struct in the Rust implementation. 22
4.5 The block_read method on BufferHead in the Rust implementation. . 22
4.7 The `Drop` trait implementation for the BufferHead struct in the Rust implementation. 23

5.1 Distribution of the CVE categories presented in Appendix B. 26
5.2 Distribution of the unsafe usages within the Rust driver. 27
5.3 An example of implicit bounds checking in Rust. 31
5.4 Performance comparison of `getdents`. 32
5.5 Performance comparison of `stat`. 33
5.6 Performance comparison of `read`. 33

List of Tables

2.1 Example of a File Allocation Table in exFAT. 12

5.1 Average slowdown of Rust implementation, compared to C. 34

1 Introduction

The Linux operating system is becoming an ever more critical part of today's society. It exists everywhere, with almost every type of device running on top of it, from Internet of Things (IoT) devices and phones to servers and supercomputers. Moreover, all are built on top of the vast open-source software that is the Linux kernel, a monolithic piece of software spanning more than 25 million lines of code [1]. Practically all of this code is today written in C, which has been the universal language for systems-level programming for most of the existence of the kernel. C has likely maintained this position because it allows the programmer near-total control over the underlying hardware and its ability to work in any environment.

However, there are some drawbacks to this. C is a relatively old language dating back to the early 1970s [2]. Perhaps due to the constraints of that time, the C language provides few guarantees regarding the safety and sanity of a compiled program. This lack of guarantees is especially noticeable with memory management, which is practically left entirely to the programmer. Furthermore, this lack of assistance from the language has given rise to many memory-related bug patterns such as buffer overflows, use-after-free, double-free and memory corruption.

Memory bugs can lead to potentially catastrophic security issues. According to an article by Microsoft Security Response Center (MSRC) from 2019, 70% of the vulnerabilities in Microsofts C and C++ code, assigned as a Common Vulnerability and Exposure (CVE), were related to memory-safety [3]. Similar numbers have been presented by the Chromium browser project, led by Google [4].

Contrary to C, Rust is a relatively new systems-level programming language, only reaching version 1.0 in 2015, which aims to be a viable alternative to C [5]. It aims to solve many issues that have haunted C for the past five decades by providing compile-time safety guarantees. Compared to many other modern languages, Rust does not achieve memory-safety by using a garbage collector since using one can negatively impact performance and thus be unviable for systems-level programs. Instead, Rust utilizes a new ownership based memory-model which performs compile-time checks to ensure that memory is properly managed.

Likely due to Rust's safety guarantees, some groups are working to introduce Rust as a second language in the Linux kernel. Most notably, the *Rust for Linux* group is working towards adding build support for Rust to the kernel. However, theirs is solely an engineering effort; They have not conducted any formal study of the feasibility of introducing Rust in the kernel. Instead, the Rust for Linux group opted to focus on the practical work of modifying the Kernels custom build system and implementing core kernel libraries in Rust.

1.1 Problem Statement

Contrary to the engineering efforts of the Rust for Linux group, this project aims to assess the viability of using Rust in Linux in terms of Security and Performance. The goal is to see if any gains within the area of security could be justified with possible penalties for performance. As such, the work can be divided into two research questions:

1. *Can the security of the Linux kernel, in terms of memory-safety, be improved by utilizing Rust?*
2. *Can any such improvements be made without sacrificing performance in terms of execution time?*

A number of metrics and goals have been set in order to evaluate if these questions have been answered. For the first research question, we utilize two metrics. Firstly, how many of the current security issues in the kernel are caused by memory-safety issues. This is relevant because if this number is low then preventing memory-safety issues will have only a small effect on improving the security of Linux overall.
The second metric is to what extent Rust code in Linux has to utilize unsafe Rust. As later described in Chapter 2, the unsafe keyword circumvents the memory-safety guarantees of Rust, which has otherwise been shown to be memory safe. Therefore, should Rust code for Linux require a high number of unsafe usages then it will not necessarily be safer than a C implementation.
Finally, regarding performance, the aim is to not be significantly slower, in terms of execution time, than existing C implementations. This goal will be considered reached if the evaluations show that Rust is at most 10% slower than C code in Linux. The threshold of 10% was set to allow for some variance but for the performance difference not to be noticeable under most circumstances.

2

Background

This chapter will present the different subjects necessary to follow later parts of the report. First an overview of the Rust language. Following this is previous research into Rust's performance and the current state of the Rust For Linux project. After this, the exFAT file system will be explained based on its specification. Finally, we will describe the structure of the current Linux exFAT driver.

2.1 The Rust Programming Language

The Rust programming language was initially developed as an open-source project started by Mozilla Research but was later moved to its own foundation, the Rust Foundation, in 2021 [6]. The first stable version of Rust was released in 2015 and, as of April 2022, the version is at 1.60 [7]. Below, different aspects and features of Rust will be presented.

2.1.1 Ownership, references, and borrowing

The mainstay feature of Rust is the concept of "ownership" [8]. This is the core feature that enables Rust code to be totally memory safe, while still being an efficient language. In short, ownership requires every value to have one "owner". An owner, in this case, is simply a variable or another value such as a struct. The owner is responsible for dropping (freeing) the value when it, for example, goes out of scope. The process of dropping is generally done automatically by the language and is not something the developer needs to consider. Additionally, we can "borrow" owned values by creating references to them (see the example in Figure 2.1).
There are two kinds of references in Rust: shared and mutable references [9]. Shared references (`&`) allow read-only access to the value, while mutable references (`&mut`) allow both reading and writing to the value. The rules of these are simple: For any given value, there may only ever exist *one* mutable reference, *xor* any number of shared references. In terms of mutability, `&` and `&mut` can be thought of as a `const` C pointer and a regular C pointer, respectively. Rust also has C-style pointers, referred to as "raw pointers". However, since Rust references have many more rules associated with them than raw pointers, casually converting between them can easily result in bugs.
In Rust, all values must always be properly initialized and aligned [10]. This also translates to references, meaning that only references that point to valid values can be created; ergo, no references can be dangling or uninitialized. Additionally,

```
struct Value;

fn main() {
    // The variable `x` owns the value.
    let x = Value;

    // Here, through the use of a reference,
    // the value of `x` is borrowed by `y`.
    let y = &x;

    // Here, through an assignment, ownership
    // is transferred from `x` to `z`.
    let z = x;

    // Since ownership of `x` was transferred, the borrow
    // `y` is no longer valid. This will fail to compile.
    let w = y;

    // The value is automatically freed at the end of
    // the scope when the owner `z` goes out of scope.
    // drop(z);
}
```

Figure 2.1: Example of ownership in Rust.

references in Rust always have a particular 'lifetime' associated with them. Lifetimes of references exist to ensure that a reference never outlives the value it is referencing. All the rules of ownership and borrowing are enforced at compile-time [8]. Thus, if one tries to compile code that would produce, for example, dangling references, that code would not compile. This means that all memory-safety guarantees that garbage collectors are usually used for essentially come without a runtime cost.

2.1.2 Unsafe

The rules of ownership are the features which make Rust memory safe. However, these rules can sometimes be too strict. The Rust compiler is inherently limited in its ability to analyse code. For example, it cannot look across function boundaries when validating lifetimes. This means that in its attempt to ensure memory-safety, the compiler might reject code which is, in fact, memory safe.
Sometimes, there is a need to bend the rules of the Rust compiler. This can be because the programmer needs to express behaviour more complicated than the compiler can understand. For example, when implementing a data structure that defies ownership. Another example is calling functions from another language (e.g. C) that the Rust compiler cannot verify for correctness. Situations such as these are where unsafe Rust comes in.
According to the Rustonomicon [11], the official documentation for unsafe Rust,

unsafe Rust doesn't remove any of the existing checks, but instead allows the programmer to perform the following specific actions:

- Dereference raw pointers
  (Unlike references, pointers have no lifetime, and do not follow the rules of borrowing)

- Call unsafe functions
  (including C functions, compiler intrinsics, and the raw allocator)

- Implement unsafe traits

- Mutate statics

- Access fields of unions

The second item in the list is of particular interest because there are two different ways one can implement unsafe Rust code. The first is to declare an unsafe code block. The second is to declare an unsafe function which, as can be seen in the list above, itself can only be called from an unsafe context. Both of these can be seen in Figure 2.2. Apart from these two, there are also unsafe traits. However, they do not provide an unsafe context but rather act as a warning to their implementors. To avoid having an entire codebase be declared as unsafe (thus defeating the point of the feature), unsafe blocks can be declared in safe contexts, such as in normal (non-unsafe) functions.

```
// Unsafe block
unsafe {
    // In this context, we can perform unsafe actions
}

// Unsafe function
unsafe fn unsafe_function {
    // In this context, we can also perform unsafe actions.
}
```

Figure 2.2: Example of an unsafe block and an unsafe function in Rust.

2.1.3 Arrays, slices, & fat pointers

Like many languages, Rust has a concept of arrays as the most primitive form of collection [12]. Arrays in Rust represent non-growable statically sized series of values, and the syntax for array types reflects this. The type of a simple array of 42 integers can be expressed as `[i32; 42]`. Rust also has the concept of *unsized* arrays, where the number of elements is omitted in the type (e.g. `[i32]`). Since these arrays do not have a known number of elements, the size of the type is not known at compile-time. Therefore, it can only be used in specific contexts, such as when creating a reference to an array (e.g. `&[i32]`).

References to arrays are called slices in Rust [13], and they have the notable property of being implemented as a combination of a pointer and a length, otherwise referred to as a *fat pointer*. This means that the compiler can ensure that every access is automatically checked at run-time to be within the bounds of the slice length, at the cost of the reference being twice the size of a conventional pointer.

2.1.4 Algebraic data types

Algebraic data types (ADTs), according to the Haskell wiki [14], are data types formed through algebraic operations on other types. The enumeration type (`enum`) in Rust is such a type. Implemented as a compile-time checked tagged union, the `enum` type is an ADT in the sense that it is the *sum* of its possible variants [15]. Like unions, each variant of an enum can be of any type. In Figure 2.3 we can see an example of a Rust enum that can be either of the three values `Number`, `Bool` and `Void` where `Number` and `Bool` also contain values of types `i32` and `bool` respectively. In Figure 2.4 we can see how this can then be used in combination with Rusts pattern matching keyword `match` which itself is compile-time checked to guarantee that all possible scenarios are handled.

```
1    enum Value {
2        Number(i32),
3        Bool(bool),
4        Void,
5    }
```

Figure 2.3: Example of enum in Rust.

```
1  let val: Value = Value::Number(24);
2
3  match val {
4      Value::Number(numValue) => println!("We have a number {}", numValue),
5      Value::Bool(boolValue) => println!("We have a boolean {}", boolValue),
6      Value::Void => println!("We have a void"),
7  }
```

Figure 2.4: Example usage of enum from Figure 2.3.

2.1.5 Generics

Like many programming languages, Rust employs generics as a tool to create abstractions and reduce code duplication [16]. Generics are effectively placeholders for concrete types that get filled in during compile-time. In Figure 2.5 we define a custom type `Tuple` which contain a pair of values, whose types are declared as the type-parameters `L` and `R`, which are placeholders that gets replaced when a concrete value of type `Tuple` gets declared. Generics in Rust can be employed in functions

and types, and all use of generics gets erased during compile-time through monomorphization. For example, this means that every appearance of `Tuple<L, R>` in the code will be replaced by a new type that is unique with respect to `<L, R>`.

Because Rust resolves generics using monomorphization, it does not rely on dynamic dispatch. Therefore, the use of generics in Rust comes for free in terms of run-time performance. Although, relying heavily on generics can adversely impact binary size since every type or function that carries generics will be compiled once for each unique combination of generic parameters.

```
// Generics can be used with structs...
struct Tuple<L, R> {
    left: L,
    right: R,
}

// ...and functions
fn flip_tuple<L, R>(t: Tuple<L, R>) -> Tuple<R, L> {
    Tuple { left: t.right, right: t.left }
}

let pair_of_numbers: Tuple<i32, &str> = Tuple { left: 42, right: "42" };

```

Figure 2.5: Example of generics in Rust.

2.1.6 Traits

Traits are another Rust concept that does not have a counterpart in C. The closest we can get is function header declarations done in `.h` files, although they are also quite different from the Rust traits [17]. Traits in Rust are more similar to `interfaces` in languages such as Java or C#, although traits are still a bit more potent in certain aspects. Rust defaults to static dispatch unless otherwise specified, which ensures that calling a trait function does not incur additional run-time overhead. Additionally, many traits can be `derived`, meaning that the compiler itself can generate a trait implementation for a type, given that the trait that supports it [18].

An example of a trait is `Default`, which requires a method called "default" that returns an instance of the implementing type. The derived variant of this trait requires that all members of a type, such as the members of a struct, also implement `Default`. Fortunately, most primitive types implement Default, with the default values for integers and booleans being `0` and `false`, respectively. When deriving the `Default` trait for a type, all its member values will be set to their respective defaults. If more advanced behaviour is required, the trait must be implemented manually. In Figure 2.6, we can see an example of a custom implementation of the Default trait.

```rust
// Here we could use #[derive(Default)] which would provide a
// default() method which would initialize 'a' to 0.
struct A {
    a: u32,
}

impl Default for A {
    fn default() -> Self {
        // Initialize "a" to a value that we want to be the default.
        Self { a: 42 }
    }
}
```

Figure 2.6: Example implementation of the `Default` trait in Rust.

2.1.7 Memory layout

Interfacing between different languages is often not straightforward, mainly because different languages often have made different design decisions that fit the way the particular language is intended to function. Communication between different languages is commonly done using a Foreign Function Interface (FFI). Thus, only a couple of parts of the languages need to be compatible for them to work [19]. The first is how functions are stored. Another is how parameters and return values of those functions are handled. Functions and data types in Rust are not compatible with C by default, with Rust not providing any guarantees as to how a particular struct (for example) will be structured in memory [20]. However, because the Rust language aims to be utilized partially as a replacement for C and C++, it includes quite a bit of support for communicating with those languages. This support includes declaring functions as `extern "C"` to make them callable from C and using the `#[repr(C)]` attribute on data types to make their layout consistent with C.

2.1.8 Iterators

To iterate over any form of data structure in Rust, one typically utilizes an `Iterator` which is a trait provided by the Rust core library [21]. It is supported inherently by some language constructs such as the `for` keyword. The `Iterator` trait itself has only one method that requires implementation, the “next” method, which is expected to return the next element in whatever is iterated over. Apart from the required method, the trait also has numerous pre-implemented methods for modifying, searching through or getting meta-information about the collection being iterated over.

2.2 Memory-safety of Rust

As mentioned previously, one of the core features of Rust is its ability to guarantee memory-safety in the absence of unsafe Rust code. A couple of studies have evaluated how well these guarantees work in practice. One, *"Memory-Safety Challenge Considered Solved?"* by Xu et al. [22], have looked into reported security issues in Rust codebases. They find, among several other findings, that all memory-safety bugs do indeed require unsafe code, with the only exception being the Rust compiler code itself. Furthermore, Xu et al. have also studied the patterns behind existing Rust security issues and come up with a few suggestions on how to write safer unsafe Rust code. A notable culprit among the dataset that was studied is the unsound declaration of unsafe functions as safe. Their suggested solution for this is to prefer to declare a function as unsafe if it, in some way, can be misused to cause undefined behavior or other memory-safety issues.

2.3 Rust For Linux

The Rust for Linux project [23] is an open-source effort to introduce Rust to the Linux kernel. So far, this effort has consisted of several parts, perhaps the most crucial being adding build options for Rust to the Linux build system so that Rust code can be built and linked with the existing C code. Another essential part has been to create a basic "kernel" library (crate), which includes several parts of the Rust standard library. These are otherwise not supported for the project and have thus had to be reimplemented for the project. Examples of what the kernel crate provides include; macros for printing, support for allocated data types such as `String` and list types, and synchronization primitives such as locks. The kernel crate also includes examples such as a simple General Purpose Input Output (GPIO) driver, which demonstrates how such a driver can be implemented using the work of the Rust for Linux group. Finally, they have also added and configured support for `rust-bindgen` [24], also known as just bindgen, which is a tool for automatically generating bindings between Rust and C.

In the case of Rust for Linux, bindings from C to Rust are handled in a couple of ways. The primary way of generating bindings consists of a C header file (*bindings_helper.h*) which contain a list of `include` statements. Bindgen is configured to generate bindings for all possible functions, structs, unions and constants that are included with these `include` statements. Because C `include`s are transitive, this soon comes to include a large part of the kernel libraries. It should be noted that the generated function calls are all unsafe as they call on C code for which the Rust compiler have no guarantees on how it behaves.

This method is somewhat limited, and there are some cases in which bindgen cannot generate bindings. The most notable such is *static inline* C functions as they appear to be inlined before linking, and thus they do not appear to exist when the Rust and C codes are linked together. For those cases, an alternative method is available in a C file (*helpers.c*). This file contains wrapper functions for the necessary calls. Bindgen then generates bindings for these wrapper functions rather than the original

functions.
Bindings such as these are not strictly necessary for the opposite direction, i.e. bindings from Rust to C code. This is due to Rust's built-in ability to declare functions, and certain data-types, as `extern "C"` which compiles the function down to the same format as their C counterparts.
The only remaining step is to ensure that the C code includes the compiled Rust code. In Linux, this is relatively simple. Generally, to include a new `module`, e.g. driver, one utilizes a macro called "module_init" which can be provided with an initialization function. This initialization function lets one do whatever other initialization is necessary for the driver to function. In Rust for Linux, this macro has been ported and combined with several other macros into one convenient macro called "module". Apart from the initialization function, the "module" macro can also be provided with meta-information about the driver, such as name, author and license. Within the initialization function, a driver can call different registration functions and provide them with structs containing pointers to functions that should handle different tasks. A file system for example might provide a `file_operations` struct which contain pointers to functions for managing files and directories such as `read` or `iterate`, for reading a file and iterating through a directory respectively.

2.4 The exFAT file system

The extensible File Allocation Table (exFAT) file system is a successor to the FAT32 file system, and the first exFAT specification was first created in 2008 by a group at Microsoft [25]. As with all file systems in the File Allocation Table (FAT) family of file systems, it revolves around a table to keep track of where files exist on the disk [26]. Apart from the FAT, there are two other important parts of an exFAT file system: the boot and the data regions.

2.4.1 Boot regions

The boot regions of an exFAT file system exist at the very start of a disk partition and contain all globally necessary information about the file system [25]. If the file system is intended to be bootable, it also contains information about how to initiate that boot procedure. Regardless of whether the partition is intended to be bootable, it contains several numbers that can be used to calculate how large the file system is. The disk itself is divided into many small chunks of equal size called *sectors*. The exact size of a sector is specified in the boot region and is somewhere between 512 and 4096 bytes. In the data region of the file system, these sectors are grouped to form *clusters* and the boot region once more gives the number of sectors per cluster. Finally, the boot region provides information to identify the file system and to verify its integrity, such as a globally unique ID and a checksum.

2.4.2 Data region

The data region of exFAT is where the actual data is stored [25]. It almost exclusively consists of the *Cluster Heap* which is all the clusters of the file system packed together

on the disk. Apart from the data of all the files on the file system, some of these clusters also contain the file system's directory structure. The directory structure of an exFAT file system is a tree-based structure where the root node is pointed to from the boot region. On the disk, the directory structure consists of several 32-byte chunks called *Directory Entries*, a combination of which forms the entries in the tree.

A directory entry can be of several different types [25]. One type is the *File* entry type, which represents either a file or a directory depending on a flag contained within the directory entry. Apart from this, the *File* directory entry contains timestamps for when the file was last accessed, modified, and when it was created. A *File* directory entry must be directly followed by *Stream extension* directory entry and at least one *File name* directory entry. A *Stream extension* contains some more metadata about the file, such as how big it is, as well as a pointer to the first cluster that contains the content of the file. Finally, it specifies the number of *File name* entries that follows the *Stream extension* entry. The *File name* entry practically only contains a series of bytes representing the name of the file. To get the complete filename, one needs to merge all the *File name* entries.

2.4.3 File Allocation Table

The exFAT file system utilises a File Allocation Table (FAT) to track how the clusters are linked together on the disk [25]. The FAT is a series of four-byte chunks called *Fat Entries*, and the purpose is to describe *Cluster Chains*. Each Cluster Chain is a series of clusters that contain the same information, such as the same file or directory contents. The number of entries in the FAT is the same as the number of clusters in the cluster heap. This is because each *Fat Entry* represents precisely one cluster. The contents of each of these *Fat entries*, except for the first two that have a special meaning, can be one of three things. Firstly it can be the index of the cluster where the cluster chain at the current index continues. This index is a number between 2, due to the two reserved blocks at the start, and *the total number of clusters* + 1, which can be at most 0xFFFFFFF6. Secondly, if this is the last, or only, cluster for that cluster chain, the content should instead be 0xFFFFFFFF. Finally, it can be 0xFFFFFFF7, meaning that the corresponding cluster is "bad", i.e. corrupted somehow.

As an example, if the cluster size is 2kb and there are two files, one that is 1kb (less than one cluster) and one that is 5kb (3 clusters), the FAT could look like Table 2.1. As we can see in the table, the first file is located entirely in the cluster at index 0x4, and the second file exists in clusters at indices 0x5, 0x150 and 0x151.

Table 2.1: Example of a File Allocation Table in exFAT.

Index	Content	Comment
...		
0x4	0xFFFFFFFF	Last cluster in chain
0x5	0x150	Cluster chain (File) continues at index 0x150
...	...	
0x150	0x151	Cluster chain continues at 0x151
0x151	0xFFFFFFFF	Last cluster of chain
...	...	...

2.5 Structure of the Linux exFAT driver

The structure of the current implementation of exFAT in Linux is based to a large degree on the structure of any file system within Linux [1]. Generally, drivers in Linux are introduced into the codebase using the Linux kernel build-system [27]. The build-system allows them to register themselves as a kernel module by providing an initialization function to the kernel as well as some basic information about the driver [28].
In the initialization function, the driver typically calls other registration functions, more specific for the module type. In the case of exFAT, this is done using a call to the `register_filesystem` function, which takes a struct with information for further initialization relevant to a file system. This form of passing structs with references to functions is used in several layers and allows the file system to specify which functions should be responsible for handling different operations. A default function exists for many operations that can be used if no specific code is required for that particular operation in the driver.
Together these various structs provide multiple entry points into the file system. In Appendix A we find an overview of how some of these entry point functions are called during common file system operations in the case of the exFAT driver. In the figure, we can see a descriptive name of the operations marked by rectangular boxes at the top of the figure with the specific command used displayed within parenthesis. The functions themselves have been put into four categories depending on their general purpose and are all prefixed with "`exfat_`". This naming scheme is used for all functions within the exFAT driver to avoid name collisions with other file systems, as the remaining parts of the names are the generic name for the operation. It should be noted that the `exfat_uf8_d_hash` and `exfat_utf8_d_cmp` functions are used because the file system is configured to work with utf8, but there are alternative implementations for these when utf8 support is not configured.

3
Methods

In this chapter, we discuss the process of finding answers to the research questions of the project.

3.1 Implementation of a driver in Rust

For both research questions, it was deemed essential to have a basis for analysis. This was decided to be in the form of a driver implementation for the Linux kernel. Specifically, it was decided to implement a Rust alternative to the existing exFAT file system driver that currently exists in the kernel and is implemented in C. Rewriting a driver that already existed ensured we had something to which we could compare our implementation. At the start of the project, no full driver implementation existed built on the Rust for Linux group's work to our knowledge. Implementing the driver ourselves also allowed an insight that we might not have gained had we only looked at existing code. Furthermore, a driver had the benefit of being relatively self-contained, which simplified the process of drawing the boundary of what was going to be reimplemented and studied.
We selected exFAT specifically because it does not require any special hardware to use it natively, which simplified the performance analysis. Furthermore, working with a file system, such as exFAT, also allowed performance to be a factor and was thus crucial for the second research question. Finally, exFAT is one of the smaller file systems in Linux, which was required to keep the project's scope in line with its timeframe. However, during the implementation of the driver, we realized that even a smaller file system driver was too much for the timeframe. Because of this, only a read-only version of the driver has been implemented.
For the driver implementation to be feasible, much work had to be put into researching the current exFAT driver implementation, file system drivers in the kernel in general, and the work that had already been done by the Rust for Linux group. This was to understand the foundations of what would later become the driver implemented in this project which is further presented in Chapter 4.

3.2 Security

The first research question "Can the security of the Linux kernel, in terms of memory-safety, be improved by utilizing Rust?" can be broken down into three smaller questions.

Firstly, what guarantees can "safe" Rust provide in terms of security and particularly memory-safety? As described in the previous chapter, this has already been answered by previous research which found that Rust is memory-safe in the absence of unsafe. However, this leads us to the next question which was also one of the metrics presented in Section 1.1. To what extent are we required to use unsafe Rust to provide a useful replacement for existing C code in the Linux kernel? For this, we studied the usages of unsafe within the implemented driver. Here, memory-safety of the driver is mainly determined by the number of unsafe usages there are and thus specific focus has been put into understanding if and how any usages can be avoided.
Finally to evaluate the second metric presented in Section 1.1, how prevalent memory-safety issues are in Linux overall and whether they are addressed by any of the guarantees that Rust provide. Here we have studied a number of CVEs concerning Linux and try to find their root causes. In order to ensure that the scope was feasible and to ensure that the CVEs in question were relevant, we set two criteria to limit the number of CVEs to be studied. The first criteria was to only look at CVEs published in 2021. It should be noted that this does not necessarily mean that issue was discovered in 2021 as CVEs are sometimes allocated long before they are published. Thus, had we instead looked at CVEs discovered or allocated in 2021, we would have worked with a possibly volatile list as CVEs discovered in that year could be published several decades into the future. The second criterion was that the CVEs have a Common Vulnerability Scoring System (CVSS) score of 5.0 or higher [29]. This leaves us with a total of 53 CVEs that have been studied in the project [30].

3.3 Performance

For the second research question, "Can any such improvements be made without sacrificing performance in terms of execution time?", which is also one of the metrics presented in Section 1.1, we evaluated the performance of both the C and Rust exFAT drivers and compared the results. The evaluation focused not only on general problems with combined C and Rust codebases but also on the Linux-specific performance-related challenges encountered.
From the operating system's point of view, the interaction with a file system occurs through a set of syscalls (system calls). We have chosen a narrow set of syscalls that encompass most of the heavy lifting when interacting with a read-only file system. The performance has then been evaluated through benchmarking and profiling of both drivers. The syscalls we have chosen are as follows:

- The `stat` family of syscalls, which returns metadata about files from their paths. This syscall was timed when repeatedly invoking it on a folder.
- The `getdents` syscall. Which returns a list of files in a directory. This syscall was timed when repeatedly invoking it on all entries in a folder.
- The `read` syscall. Which reads bytes from a file. This syscall was timed when repeatedly invoking it on a file.

We use custom tools to benchmark and profile the various syscalls to measure the latency and throughput when calling them, measuring execution time when the size of the file or folder was successively increased. Additional detailed profiling was performed using **perf**, the Linux profiling tool, to generate detailed stack traces and identify hot code paths.

4 Implementation

This chapter covers the implementation of the exFAT driver and the issues encountered throughout this part of the project.

4.1 Structure of the driver

The structure of the driver is, to a large degree, similar to the existing C implementation. Partially because of the interface that the driver has to present to the kernel. Furthermore, this has proven to be one of the challenges with the rewrite, as this structure conflicts with the general way that Rust is intended to be used. One example of this is that the kernel relies heavily on late initialization, i.e. data being assigned at a different point from where it is allocated. This is contrasted with Rust's philosophy of never storing meaningless, such as uninitialized, values and keeping data immutable as much as possible.
If we look at Appendix A which represents most of the functions exposed by the existing C driver, many of them also exist in the new Rust variant. Some exceptions are the functions in the writing category, as this project only covers a read-only driver implementation. Furthermore, only the `exfat_alloc_inode` method exists from "General maintenance," thus excluding `exfat_d_revalidate`, `exfat_utf8_d_hash`, and `exfat_utf8_d_cmp`. This is because the `exfat_d_revalidate` function is only necessary to ensure that the cache is kept up to date with what is actually on the disk and is thus not needed for a read-only variant. Regarding the `exfat_utf8_d_hash` and `exfat_utf8_d_cmp` functions, these are also mainly used for the write part of the driver and thus we elected to use the default implementations of these. This decision to use default implementations for these two functions has led to only one noteworthy difference between the Rust and C implementations. The C implementation is case-insensitive, whilst the Rust implementation is case-sensitive. For example, this means that in the C implementation `cat SOME_FILE` will output the content of a file called "SomE_FilE" or other combinations of letter casings, but the Rust version will say that the file could not be found.

4.2 Data structures

The usage of data structures represents one of the most significant differences between our Rust implementation and the existing C implementation of the exFAT driver. In the C implementation, few external data structures are used. Instead,

```
let reader = DirEntryReader::new(sb_info, sb_state, inode.data_cluster)?;

for entry in reader {
    // Check if we got an error whilst reading the entry
    let entry = entry?;

    if entry.name == name {
        return Ok(entry);
    }
}

// No entry matched the name
Err(Error::ENOENT)
```

Figure 4.1: Find dir in the Rust implementation of exFAT.

data structures are pseudo integrated directly into the codebase. One of the exceptions to this is linked lists, which use the kernels defined macros for working with this particular data structure. However, on the Rust side, we heavily utilize data structures from both the kernel crate and the Rust core library. Below, we will detail several cases where we have used such data structures and contrast them with the C implementation.

4.2.1 Iterators

In the implementation, iterators are used in a few places. They are perhaps most notably used when searching through the directory tree. In this case, three Iterators are being used on top of each other to iterate over the contents of a directory. The first one is the “FatChainReader”, which iterates over the exFAT clusters on the disk and is the only one that reads data from the disk directly. Reading the cluster chains is the second iterator, the “ExFatDirEntryReader” which iterates over exFAT directory entries, as specified in the exFAT specification. Finally, the “DirEntryReader” iterates over files and directories in the more typical sense. While this might seem complex, it creates a straightforward interface for the rest of the codebase.
An example of how these iterators can be utilized can be seen in Figure 4.1 which is the initial implementation for looking up an entry (file/subdirectory) in a directory. Whilst this is missing some later optimizations, such as comparing hashes of the names instead of the full names, it is still the essence of how this task is performed. As shown in Figure 4.1, only the most high-level of the iterators, “DirEntryReader”, need to be used as the others are used internally.

4.2.2 Algebraic data types

Algebraic data types have been used at several points throughout the implementation. One notable example is the implementation of exFATs directory entries

```c
/* dentry types */
#define EXFAT_UNUSED         0x00           /* end of directory */
#define EXFAT_DELETE         (~0x80)
#define IS_EXFAT_DELETED(x)  ((x) < 0x80)   /* deleted file (0x01~0x7F) */
#define EXFAT_INVAL          0x80           /* invalid value */
#define EXFAT_BITMAP         0x81           /* allocation bitmap */
#define EXFAT_UPCASE         0x82           /* upcase table */
#define EXFAT_VOLUME         0x83           /* volume label */
#define EXFAT_FILE           0x85           /* file or dir */
#define EXFAT_GUID           0xA0
#define EXFAT_PADDING        0xA1
#define EXFAT_ACLTAB         0xA2
#define EXFAT_STREAM         0xC0           /* stream entry */
#define EXFAT_NAME           0xC1           /* file name entry */
#define EXFAT_ACL            0xC2           /* stream entry */

struct exfat_dentry {
    __u8 type;
    union {
        struct {
            // ...
        } __packed file; /* file directory entry */
        struct {
            // ...
        } __packed stream; /* stream extension directory entry */
        struct {
            // ...
        } __packed name; /* file name directory entry */
        struct {
            // ...
        } __packed bitmap; /* allocation bitmap directory entry */
        struct {
            // ...
        } __packed upcase; /* up-case table directory entry */
    } __packed dentry;
} __packed;
```

Figure 4.2: C implementation of the exFAT directory entry values, data values omitted for brevity.

```
pub(crate) enum ExFatDirEntryKind {
    Deleted,

    // Critical primary
    AllocationBitmap(AllocationBitmap),
    UpCaseTable(UpCaseTable),
    VolumeLabel(VolumeLabel),
    File(File),

    // Benign primary
    VolumeGuid(VolumeGuid),
    TexFatPadding(TexFatPadding),

    // Critical secondary
    StreamExtension(StreamExtension),
    FileName(FileName),

    // Benign secondary
    VendorExtension(VendorExtension),
    VendorAllocation(VendorAllocation),
}
```

Figure 4.3: Rust implementation of the exFAT directory entry values, the values of variants are structs defined separately.

(Section 7 of the specification [25]). The C implementation of this can be seen in Figure 4.2 and as can be seen they utilize a `struct` consisting of a `union` of the different variants as well as a type value to determine which of the `union` values is currently used (i.e. a `tagged union`). This type value is also expected to be one of the `constants` defined above the `struct`. The `structs` within the `union` contains the values that each of the different type represents. However, in Figure 4.2, these have been omitted for brevity. In Figure 4.3 we can see the Rust implementation of the same concept. Note that instead of defining the contents of the different variants' data values, we elected to put those in structs defined separately.

4.2.3 Locking

Locking is a critical component within the kernel and is required to prevent data races and avoid critical errors. Within the Linux kernel, there are several data structures for managing locking. One of the widely used ones is the `spinlock_t` struct. It would appear as if praxis when using it is to put the lock next to whatever data it is supposed to lock and keep the names similar, although this does not appear to be mentioned in the documentation. E.g. a struct containing a value `some_data` would have a lock next to it called `some_data_lock`.
In Rust, the kernel crate also provides some synchronization primitives, among them a `SpinLock`. Instead of simply keeping the lock nearby the data it is supposed to lock,

this SpinLock implementation utilizes Rusts support for generics to encapsulate the data and thus enforces that the data must be locked before usage. Furthermore, when the data is retrieved, and thus the lock is locked, this is done using the lock method. However, the lock method returns the data wrapped in a Guard data type, which will automatically unlock the spinlock when the data goes out of scope, similarly to how freeing is handled in Rust. Finally, there are cases where one might want to avoid handling the lock and can be sure that it is safe to do so. For those cases, there exists a method locked_data which will return the data wrapped in an UnsafeCell data type, which will require an unsafe block when working with that data.

4.2.4 Reading data from memory

A critical aspect of any file system is reading data from memory. In the original C implementation of the exFAT driver, this is mainly accomplished using calls to different bread (block read) functions. This is also the case in the Rust implementation. However, to avoid calling the native C function at multiple locations, a wrapper struct and accompanying methods have been implemented. The struct itself can be seen in Figure 4.4, and as can be seen, it is relatively simple, only containing two fields. The first field, sector, is simply a number representing the index of the current sector, whilst the second field is a mutable raw pointer pointing at where the data can be read.

```
1 pub(crate) struct BufferHead {
2     sector: sector_t,
3     ptr: *mut buffer_head,
4 }
```

Figure 4.4: BufferHead struct in the Rust implementation.

As described earlier, working with raw pointers is an unsafe operation in Rust. Thus, their usage should preferably be contained within safe wrapper functions that perform whatever actions are necessary to avoid undefined behaviour. In this spirit, several helper methods have been implemented for the struct. The first such function is the one that creates the struct, block_read, which can be seen in Figure 4.5. In the figure, we see that the method tries to create an instance of the struct by requesting the memory from the kernel using a call to __bread_gfp, which returns a pointer to a buffer_head struct. The buffer_head struct mainly consists of two values, a pointer to where the data can be found and the size of the data.

```rust
pub(crate) fn bytes(&self) -> &[u8] {
    unsafe {
        let bh = &*self.ptr; // bh is of type buffer_head, a struct from C.
        slice::from_raw_parts(bh.b_data as *const u8, bh.b_size)
    }
}
```

Figure 4.6: The `bytes` method for the BufferHead struct in the Rust implementation.

```rust
pub(crate) fn block_read(sb: &super_block, sector: sector_t) -> Option<Self> {
    let ptr = unsafe {
        __bread_gfp(sb.s_bdev,
                    sector,
                    sb.s_blocksize as c_uint,
                    ___GFP_MOVABLE
        ).as_mut()?
    };

    Some(BufferHead { sector, ptr })
}
```

Figure 4.5: The block_read method on BufferHead in the Rust implementation.

Another important method on the BufferHead struct is the `bytes` function which can be found in Figure 4.6. As can be seen, the function creates a new slice using the Rust core function `from_raw_parts`. Using a slice rather than a char pointer, which is used in the C implementation, has several benefits. As previously described, Rust slices have their lengths tracked internally so that automatic boundary checks can be added at compile-time. This also appears to be the purpose of the `buffer_head` struct provided by the kernel, although it still requires manual additions of boundary checks. In this case, the Rust implementation also has a potential memory-safety issue as it still relies on the `__bread_gfp` method to never return an instance of that struct with an invalid length. Should this not be the case, it could result in a potential out of bounds read bug on line 4 of Figure 4.6 in the call to the unsafe function `slice::from_raw_parts`. This is because the function leaves it to the caller to provide valid values, hence it being unsafe. However, it should be noted that the same problem exists in the C implementation.
Finally, whilst the kernel allocated and provided the `buffer_head` struct during the call to `__bread_gfp`, we are the only ones that know when we are finished with it and are thus responsible for freeing it. As described earlier, one of the main reasons Rust has its ownership and lifetime features is to avoid having the programmer handle the freeing of memory. While the compiler can generally detect when to free a piece of memory using these features, it cannot always determine how to do it. This is one of those cases. When freeing memory that has been allocated

```
1 impl Drop for BufferHead {
2     fn drop(&mut self) {
3         unsafe { __brelse(self.ptr) }
4     }
5 }
```

Figure 4.7: The `Drop` trait implementation for the BufferHead struct in the Rust implementation.

using a `bread` function, the function `__brelse` has to be called. Fortunately, there is the `Drop` trait which is typically inferred but can be manually implemented in situations such as this. In Figure 4.7, we can see how this trait is implemented for the BufferHead struct. A call to this `drop` function will be automatically inserted when the compiler determines that the data is no longer needed. This is in contrast to the original implementation, where such a call was needed for every possible code path to avoid a memory leak.

4.2.5 Deviating from the specification

As a basis for the implemented driver, the main one has been the original C implementation. However, we have also made great use of the exFAT specification, to which the C implementation also largely conforms. However, during the implementation, we discovered one significant deviation from the specification made by the original implementation. According to the specification, as previously described, the contents of a *FatEntry* should be either in the range 2 to 0xFFFFFFF6 (inclusive), exactly 0xFFFFFFF7 or exactly 0xFFFFFFFF. However, during our work, we found that the current Linux implementation of exFAT sets most of these values to 0, which is not within the range of allowed values. Further studies revealed that this was also the case for the Windows exFAT driver written by Microsoft, the same company that wrote the specification. Initially, we believed this to represent that the cluster at the index had not been allocated as this case is not mentioned within the specification. However, later tests revealed that this was the case even for partitions close to or entirely filled with files and directories. From what we have gathered, 0 simply means that the file is not fragmented and that it either ends in that entry or continues in the cluster immediately following it. Since the total length of a file is specified in its *File* directory entry, one can distinguish between the two cases, and thus this can be thought of as a performance improvement. However, this could be a big issue as this area handles memory that might be shared between different file system implementations. For future implementations to be compatible with the ones for Linux and Windows, they too must follow this unspecified rule.

5
Evaluation

In this chapter, evaluations of the implemented driver are presented. It begins with the security evaluation, which is split into two parts: a study of the selected CVEs for Linux overall, and unsafe usages within the implemented driver. This is in line with the first two metrics set out to measure the success of the project. Following this is the performance evaluation, which consists of some high-level considerations when it comes to the performance of Rust in Linux and a benchmarking comparisons between the C and Rust implementations which was the last metric of the project.

5.1 Memory-safety in Linux

The first part of the security evaluation of this project was to study the prevalence of memory-safety issues in Linux overall. For this purpose, 53 CVEs were selected to be studied. This study was carried out on a case-by-case basis, and a summary of it can be found in Appendix B. All CVEs have been given a general category which comprises the cause or the particular danger with that CVE. Furthermore, each CVE was evaluated to determine if it is related to memory-safety and thus could be affected or even prevented using safe Rust. Apart from the category and whether it was memory-safety related, the appendix also includes the ID of the CVE as well as its CVSS score. The distribution of CVEs within each category can be found in Figure 5.1. In summary, 38 ($\approx 72\%$) of the CVEs were determined to be memory-safety related issues and the remaining 15 ($\approx 28\%$) of other vulnerability categories. In the figure, these two groups have been separated using different colours. Below is a description of each category and, if it has been determined to be memory-safety related, how exactly safe Rust could have prevented that issue. It should be noted that all cases are based upon the assumption that there are not any bugs in the Rust language itself.

5.1.1 Out of bounds read/write

The most common types of CVEs in the dataset are different kinds of out-of-bounds reading or writing. These can appear in the form of buffer overflows or any other cases where data is read or written outside the expected or allocated memory space. Most of the CVEs within this category were caused by insufficient or missing bounds checking before accessing or modifying data from memory. Safe Rust avoids these issues by checking indices during compile-time to as large an extent as possible. Furthermore, for cases which are not knowable during compile-time, the compiler

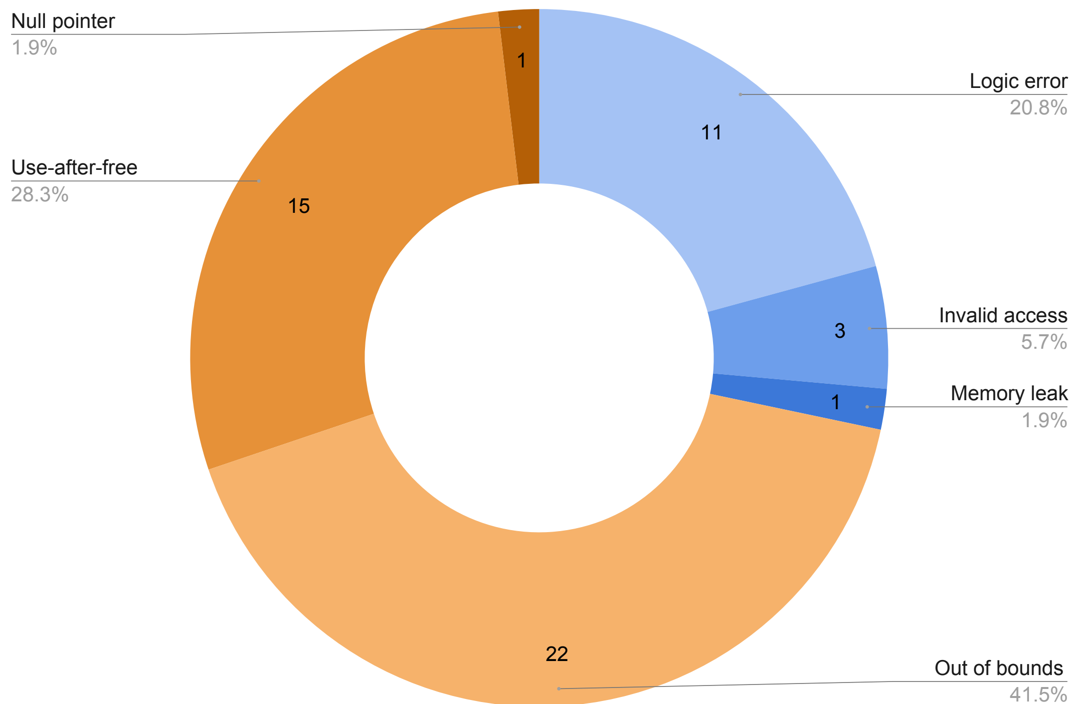


Figure 5.1: Distribution of the CVE categories presented in Appendix B.

automatically adds bounds-checking instructions, which will generate a panic during runtime if the bounds are exceeded [22].

5.1.2 Use-after-free

Another common problem in the dataset is use-after-free, i.e. usage of references to data that has been freed. In the dataset, these were mainly caused by multiple references being held to the same memory and not being correctly synchronized when that memory should be freed. In safe Rust, these problems are prevented during compile-time by checks performed by the compiler. These checks are mainly based around its lifetime and ownership components which ensures that it is known, during compile-time, exactly how long a given piece of data will live and where it can be used. Thus, code that would lead to use-after-free will not be able to be compiled using safe Rust.

5.1.3 Null pointer dereference

Only one case of a null-pointer dereference was found in the dataset, which was caused by a failure to check for validity before using a variable. Since dereferencing a raw pointer is inherently an unsafe operation in Rust, and since Rust otherwise does not have a concept of null, such issues cannot arise in safe Rust.

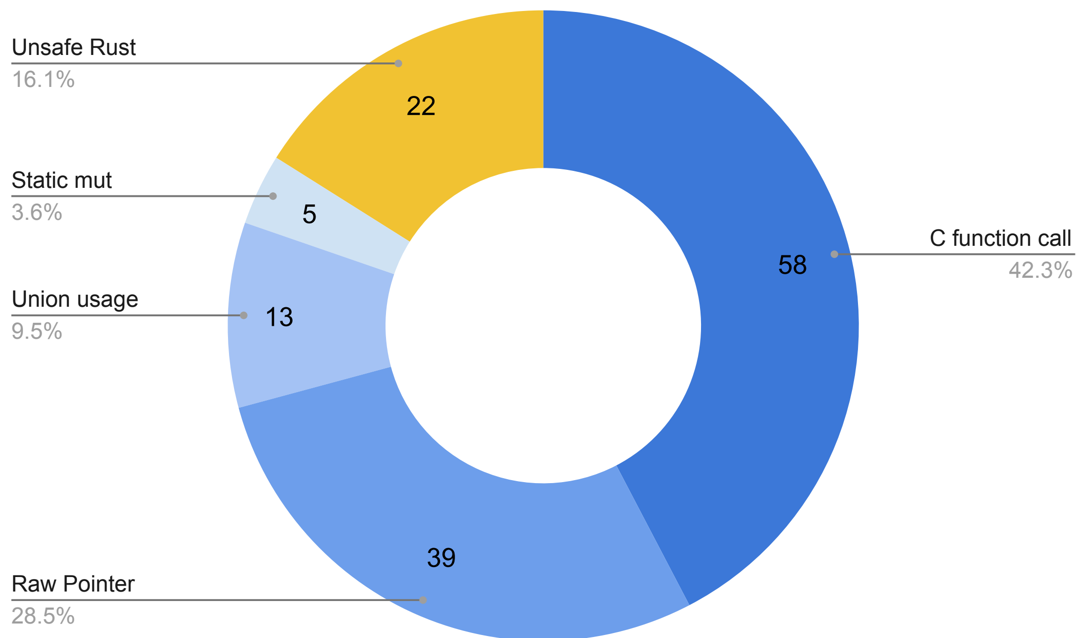


Figure 5.2: Distribution of the unsafe usages within the Rust driver.

5.1.4 Non memory-safety related CVEs

The remaining CVE types were determined to have other causes than memory-safety issues. These primarily consist of what have been classified as "Logic errors", which represent cases where the issue is solely algorithmic in nature. An example of this is CVE-2002-2438 which was caused by a failure to check flags during the reception of a TCP SYN packet. This oversight could lead to invalid packets being allowed through the firewall. The other CVE types within this category are: cases of invalid access management, meaning that users were allowed to perform actions that they should not have had access to; and cases of improper locking, leading to race conditions. These are all cases unrelated to memory-safety and thus are not covered by the memory-safety guarantees of Rust. Finally, there was a single case of a memory leak which is not considered a memory-safety issue in Rust and can thus occur in safe Rust code.

5.2 Usages of unsafe within the implemented driver

In total there are 131 usages of unsafe blocks within the implemented driver. There are several different reasons for the existence of these unsafe blocks, and their distribution can be seen in Figure 5.2. It is worth noting that some unsafe blocks contained more than one of these reasons, and thus the total number of "causes" is 137, not 131. Below are explanations of these different categories.
As shown in Figure 5.2, we see that 58 unsafe blocks contained calls to C functions or specifically the generated bindings for them. With these function calls come many other unsafe usages such as 39 dereferences of raw pointers. These dereferences are

necessary because many values handled in connection to the function calls are either primitive types, such as numbers, or raw pointers to more complex types, such as structs. To use these complex types, they first have to be dereferenced to have their actual values retrieved. Similarly, there are 13 accounts of usages of Rust unions which are inherently unsafe. These union usages are due to unions in the C code, for which bindgen will generate equivalent Rust unions. Finally, there are 5 usages of variables declared as `static mut`. These are used on the structs that are provided by our file system to the C code when registering some functionality, as described in Section 2.5.

5.2.1 Unsafe Rust functions & macros

Apart from the usages of unsafe blocks caused directly by the communication with the C part of the codebase, there are 22 occurrences when the code calls unsafe Rust functions or macros not generated by bindgen. These are not generally as straightforward as the other usages of unsafe and thus will be dealt with individually.

5.2.1.1 Zeroed macro

The first of these unsafe Rust functions and macros are the usages of `zeroed!`, a utility macro which initializes all values of a given struct to be zero. The reason for it being unsafe is that it can lead to undefined behaviour if any of the values within the struct does not have a defined behaviour when set to zero. In all the locations that the macro is used, it initializes a C struct. More specifically, a struct translated from C by bindgen, and since 0 is a valid value for all C types, this is safe for all those cases. Furthermore, these structs are then provided to the C code, which expects unused values to be initialized to zero. Thus, this avoids unnecessarily extending the initialization of these (sometimes very large) structs. Rusts way of handling this would have been to implement (or derive) the `Default` trait. However, since the structs are generated by bindgen, we cannot implement this on our end. Furthermore, a default value in Rust does not necessarily mean being represented by 0, which could break assumptions made in the C code.

5.2.1.2 Boxed values

The next unsafe Rust function that is used is `Box::from_raw` which, given a pointer, creates a “Boxed” value i.e. a value allocated on the heap. This function effectively casts a possibly unknown piece of memory into a value, for which the memory could be invalid and can therefore be dangerous. However, this function is only used in a couple of places. The first is to create a string from a union value where the safety of the operation is based on a tag accompanying the union, promising that it is indeed a string value. Furthermore, the same assumption is also made within the C implementation. The second is to free a raw pointer initially allocated as a Boxed value by Rust, and the value was first turned into a raw pointer to be passed to the kernel. When the kernel needs to free the value, we remake it into the type it was allocated as since only then can we know how it should be freed.

5.2.1.3 Spinlocks and Mutexes

There are a couple of calls to `Pin::new_unchecked`. A "Pinned" value in Rust is a value whose placement in memory is guaranteed not to change. When initializing the locks and mutexes provided by the Rust for Linux kernel crate, the lock or mutex to be initialized must be Pinned. The `new_unchecked` method specifically creates a Pinned value from a reference and for a type that does not implement the `Unpin` trait. Calling this method is unsafe because it effectively promises the compiler that the value in question will not be moved.
Similarly, there is also the `SpinLock::new` and `Mutex::new` methods that create a SpinLock and a Mutex respectively. They are unsafe because the SpinLock or Mutex is required to be initialized before being used as usage before then is considered undefined behaviour.

5.2.1.4 KMemCache

A couple of unsafe methods that we have written ourselves are `KMemCache::create` and `KmemCache::free` methods. These methods create and free the primary memory cache we use within the driver. The first, `KMemCache::create` creates the cache itself and is unsafe as the developer must only call this only once, before anything is inserted into the cache. The second, `KMemCache::free`, on the other hand, handles the freeing of objects put into the cache and takes one such object as an argument. This method is unsafe because the developer must ensure that the object provided must have been previously inserted into the cache. Furthermore, to avoid a use-after-free issue, the developer must ensure that the pointer is not used after the call.

5.2.1.5 Linked lists

Another unsafe method on a datatype we use is the `LinkedList::remove` method, which removes a value from the list. The reason for it being unsafe is because the developer is required to ensure that the value that is to be removed either exists in the list the method is called on or is not within any such list. If the value exists in another list than the one the method is called on, that could lead to certain memory-safety issues.

5.2.1.6 Built-in functions

We also use the Rust built-in functions `transmute` and `MaybeUninit::assume_init`. The first of these, `transmute`, takes a number of bytes and reinterprets them as a specific type, i.e. casting the bytes to the type. It is unsafe because the developer must ensure that those bytes can correctly represent the type they are cast into. When reading certain parts of the disk, we use this function to cast them into structs representing different parts of the exFAT file system. These structs only contain numbers which means that all values are valid for those types, and certain checks are made to ensure that the values of those numbers are valid and conform with the specification.

The other function, `MaybeUninit::assume_init`, acts on the `MaybeUninit` type, which, as the name suggests, wraps a value that may or may no have been initialized. The method, `assume_init`, assumes that the value has been initialized and is unsafe because it can lead to undefined behaviour if called before the value has been initialized. In the implemented driver, `MaybeUninit` is used only once to store a value that is to be parsed by a function from C. The function that is supposed to parse the value takes a pointer to the variable that should store the parsed value. Since the value is not initialized before the call to the function and might be, depending on whether an error occurred during parsing, we use the MaybeUninit type to represent this behaviour.

5.2.1.7 Methods on raw pointers

At one point in the codebase we used the `mut_ptr::write` and `mut_ptr::drop_in_place` methods on mutable raw pointers and as previously described, working with raw pointers is inherently unsafe in Rust. The first method, `write`, writes a value to the place that the raw pointers point to in memory, and we use this to initialize an object to which we have received a (raw) pointer from the C part of the kernel.
The second method, `drop_in_place`, is used to call the so-called “destructor” for the type being pointed to. The point of a destructor is to perform any actions needed before the value is freed, such as freeing member values. The function is unsafe because as it is being called on a raw pointer, we cannot be sure that the value being pointed to is actually of the expected type and can thus lead to undefined behaviour.

5.3 Performance

The second goal of this project was to establish whether using Rust in the kernel comes with negative performance implications, primarily in terms of execution time, i.e. making the kernel slower. In this section, we first present some high-level findings on inlined functions and bounds checking. Then we follow it up with the benchmarking comparison of the Rust and C exFAT drivers.

5.3.1 Macros & inlined functions

The Linux codebase frequently uses both macro functions and explicitly inlined functions. Linux 5.2 contains over 11 000 instances of `static inline` functions in the core kernel header files, some of which are used by the C exFAT driver.
In the C driver we catalogued 126 calls to external inline functions and macros. We found that 43 of them had equivalent functions in the Rust core library or the Rust for Linux library. Out of the remaining ones, 45 were simple wrappers of other non-inline functions that could trivially be ported to Rust. The performance of the ported functions should be equivalent since Rust also supports explicitly inlining functions.
For the functions that were not ported, there exists a special helper file, as described in Section 2.3, where non-inline functions written in C are created to wrap the

```rust
fn sum_first_three(slice: &[i32]) -> i32 {
    // This would result in three seperate bounds checks
    return slice[0] + slice[1] + slice[2];

    // Alternatively, this would result in only one bounds check
    if slice.len() < 3 { panic!() }
    return slice[0] + slice[1] + slice[2];
}
```

Figure 5.3: An example of implicit bounds checking in Rust.

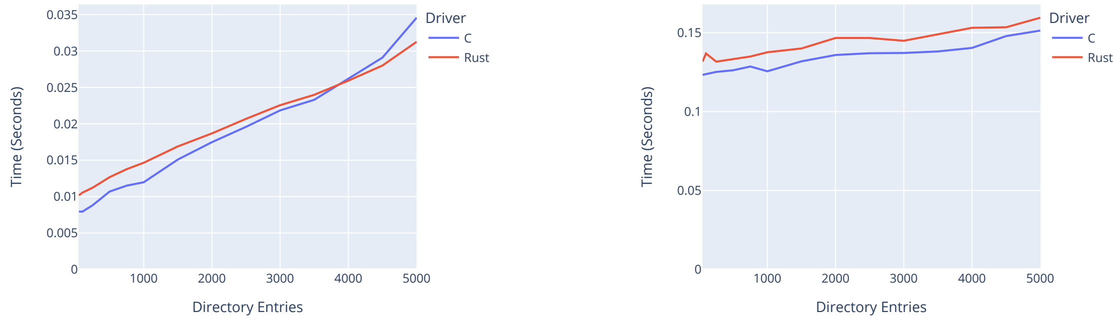
`static inline` functions from the kernel. This introduces the overhead of an extra function call, but it allows these functions to be used from Rust.
The compiler can still inline functions which are not explicitly marked as inline for performance reasons. Although since kernel drivers are compiled into individual object files, this kind of automatic inlining of kernel functions is not possible. This is the case regardless of what language a driver is written in.

5.3.2 Bounds checking

As previously mentioned, Rust tries to resolve the issue concerning out-of-bounds array accesses by introducing automatic bounds checking. This means that for every array access, the compiler will ensure an appropriate bounds check exists. Rust will use context to determine when bounds checking is necessary in order to remove unnecessary instructions. There may, however, exist many cases where Rust cannot efficiently optimize away bounds checks. Either because it does not have enough information or because the source code is written in such a way as to make it impossible. A simple example of the latter is shown in Figure 5.3, where the three separate array accesses might individually panic, meaning that each must be accompanied by a bounds check. This can be fixed by, for example, introducing an explicit bounds check.
An overabundance of bounds checks might incur a performance penalty for compute-intensive code. However, we have not found that to be an issue for the exFAT driver since the driver is mainly I/O bound on performance.

5.3.3 Benchmarking the driver

Below we present the results of the performance comparison of both exFAT implementations. All timings presented are averaged from repeated benchmarking. Each figure contains two results. The first is the benchmark running with the Linux dentry-, inode-, and page cache enabled. How the caches work is not important, except that the kernel will bypass most of the driver code when they are enabled. This means that the results should be more similar since only a small portion of time is spent within the respective driver implementations. The second graph for each benchmark shows the benchmark results with the caches disabled. This one is the most interesting since the majority of work is happening within the drivers.



(a) With cache. **(b)** Without cache.

Figure 5.4: Performance comparison of `getdents`.

It is important to note that due to time constraints, not all the algorithmic optimizations performed by the C driver were ported to Rust. Specifically, only the implementation of `read` includes all optimizations from the C driver.

5.3.3.1 Performance of `getdents`

The `getdents` benchmark was performed by iterating over all the entries in a directory using the `getdents64` syscall. The results are shown in Figure 5.4, where the x-axis represents the number of entries in the directory, and the y-axis is the time taken. With the cache enabled, both implementations perform roughly equal. With the cache disabled, the time on average differs by about 0.009 seconds.

5.3.3.2 Performance of `stat`

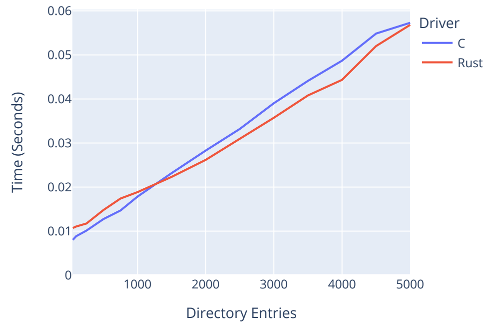
The `stat` benchmark was performed by calling `newfstatat` on all entries in a directory. The results are shown in Figure 5.5, where the x-axis represents the number of entries in the directory, and the y-axis is the time taken. With the cache enabled, both implementations perform roughly equal. With the cache disabled, the drivers perform about the same for small folders, but with larger workloads, the Rust drivers slow down significantly, up to $\approx 30\%$ in our tests.

5.3.3.3 Performance of `read`

The `read` benchmark was performed by reading entire files using the `read` syscall. The results are shown in Figure 5.6, where the x-axis represents the size of a file, and the y-axis is the time taken to read all of its contents. Regardless of whether the cache is enabled, both implementations perform about equally.

5.3.3.4 Summary of benchmarks

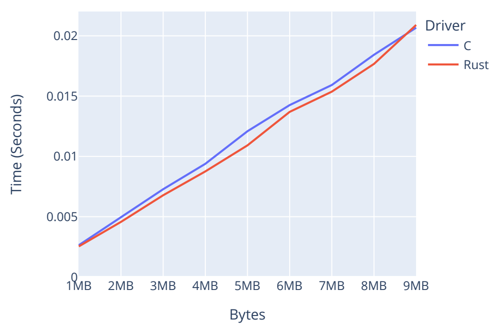
Table 5.1 shows the average slowdown of the Rust ExFAT implementation, when compared to the original C implementation. The `read` benchmark shows the best

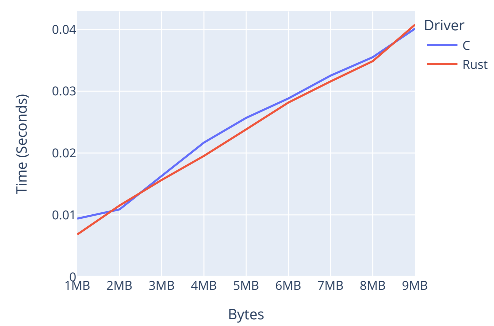


(a) With cache.

(b) Without cache.

Figure 5.5: Performance comparison of `stat`.





(a) With cache.

(b) Without cache.

Figure 5.6: Performance comparison of `read`.

Benchmark	**With cache**	**Without cache**
getdents	9.4%	6.4%
stat	2.9%	12.0%
read	-5.4%	-6.7%

Table 5.1: Average slowdown of Rust implementation, compared to C.

result, with an average speedup of about 6%. `stat` shows the worst results, with an average slowdown of 12%, and up to 30% in the worst case.

6
Discussion

The successful implementation of a read-only driver in this project has shown that it is entirely feasible to rewrite a driver for the Linux kernel in Rust, building on the work of the Rust for Linux group. What follows is a discussion of the evaluation. First, from the point of view of security, mainly in terms of memory-safety. Secondly, regarding performance, with discussions of the benchmarks and findings on Rust.

6.1 Security

In terms of security, the evaluation has shown several points. First, and perhaps most importantly, memory-safety is somewhat of an Achilles heel when it comes to the security of the Linux kernel. This is shown by the fact that almost 72% of the studied CVEs were found to be within this category. Furthermore, this appears to be in line with the 70% found by Microsoft and Google in their own security issues. These numbers show that memory-safety is genuinely a critical area to focus on when it comes to building secure and stable software.
As presented earlier, previous work by Xu et al. shows that Rust, in the absence of unsafe, is truly memory-safe. Rust achieves this safety without using a garbage collector or similar runtime since using one could lead to significant performance losses. However, the criteria "in the absence of unsafe" is worth further examination. As previously presented, the implemented driver contained 131 usages of unsafe blocks. This number is quite high for a relatively small file system driver, especially considering that only a read-only version has been implemented. Furthermore, these usages are spread throughout the entire codebase. The amount of unsafe usages could likely be reduced by increased use of different wrapping techniques, such as extracting them to macros and functions. However, that would still be inferior to removing the reasons for their existence, and instead, we must look at these reasons. For a vast majority of them, they proved to be due to the dependency on a foreign language, specifically the C part of the codebase.
One of the largest causes for the unsafe blocks were calls to different C functions, which are of differing sizes. Some of them are small enough for it to be feasible to rewrite them together with the driver. However, several of them are much larger, depending on several other parts of the kernel and are themselves depended upon throughout large parts of the kernel. Thus, to be able to significantly reduce the number of unsafe usages within the implementation, a relatively large part of the Linux kernel core libraries would also have to be rewritten. These libraries are an area where the Rust for Linux group has made some progress, yet much work

remains.
When looking at the studied CVEs, we notice that whilst most of them are memory-safety related, a full 28% of them are not. Some of these CVEs appear to be very difficult to solve with the forms of automated checks and guarantees that the Rust compiler provides, as they are purely algorithmic in nature. However, for other CVEs, we see some promising solutions, both within our implemented driver and within the Rust for Linux kernel crate. An example of one such solution is for the various CVEs concerning locking. The implementation of locks within the kernel crate, presented in 4.2.3, could prevent several of these CVEs outright, assuming the absence of unsafe operations circumventing these guarantees. Furthermore, whilst bugs caused by a failure to unlock locks for specific code paths did not occur in the studied dataset, they are certainly possible in how the C implementation handles locks. Since the Rust implementation of these locks handles this automatically, the risk of such bugs is also significantly reduced. The problem here, of course, is that this is just one implementation, and the developer could just as well implement their own locking mechanism where these guarantees do not exist. Thus, no general guarantees can be provided such as they can for memory-safety.
Finally, there is also an argument to be made that different usages of unsafe bring different levels of risk. Whilst, in theory, unsafe Rust can be less safe than C, this mainly comes down to how much can be considered undefined behaviour and what assumptions can be, and are, made for that code. The usages of unsafe that are necessary for interfacing with the C codebase generally make the same assumptions as the C implementation does and should thus be on a similar level from a security standpoint. Considering that the parts of our Rust driver that is free from unsafe blocks should be much safer, one can argue that, overall, the Rust implementation is safer, in terms of memory-safety, than the original C implementation.

6.2 Performance

From the evaluation results, we can see that typically, the Linux kernel heavily utilizes caching to keep a file system performant. However, even without the cache, we notice that the Rust driver can perform competitively in some cases. Specifically the `read` syscall benchmark, shown in Figure 5.6. It is worth noting that for the remaining benchmarks, the Rust implementation still lacks some optimizations in those areas, which can be found within the C implementation. The goal of being at most 10% slower than the original driver was met for two of the three system calls and we are certain that it can be met also for the last with more time spent on implementing optimizations.
We have seen that generally, Rust introduces many new language features compared to C. Throughout the rewrite, we extensively used generics and traits, with notable examples being the Iterator trait and the SpinLock struct with its generic data parameter. While one might expect these features to incur some performance penalty, Rust allows our driver to perform as well as the C driver. It does this using techniques such as static dispatch and monomorphization to eliminate any runtime overhead of using these abstractions. Thus, what we have seen is that Rust generally follows the Zero Cost pattern for these high-level abstractions, as coined by Bjarne

Stroustrup [31, page 5]:

> “What you don’t use, you don’t pay for. And further: What you do use, you couldn’t hand code any better.”

7
Related Work

The Rust language has received much attention for its different claims in terms of providing memory-safety whilst being performant and viable enough for systems-level programming. As such, this chapter will discuss what previous work has been done in studying the performance and the memory-safety of Rust and how this project differentiates itself from them. Moreover, this chapter presents other works that have been done within the area memory-safety of C and the security of the Linux kernel. Finally, this chapter will look at how languages other than C and Rust have been used in operating system development.

7.1 Memory-safety of Rust

There has been another study in the area of unsafe usages within Rust programs, *"Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs"* by Qin et al.[32]. This paper has studied known security issues in Rust codebases and unsafe usages in several open-source Rust projects with a specific focus on the Rust standard library. Among other findings, they find that "Most unsafe usages are for good or unavoidable reasons". Furthermore, similarly to the findings by Xu et al. [22] presented in Section 2.2, they find that all memory-safety issues encountered required unsafe code.
In contrast to the study by Qin et al., who focused on pure Rust or mainly Rust codebases, this project has studied the memory-safety of a mainly C project where Rust was added later. As such the proportions of unsafe usages that were caused solely due to communicating with another language is much higher within our project. Furthermore, whilst we agree that most of our unsafe usages are for good reasons we find that very few of our unsafe usages are for "unavoidable" reasons. Instead, we find that most of our unsafe usages can be avoided by porting a more significant portion of Linux to Rust.
Coming from another angle is *"Keeping Safe Rust Safe with Galeed"* by Rivera et al.[33]. In contrast to this project, Rivera et al., focuses on the effects of the unsafe keyword on safe Rust code. As mentioned in the paper, certain actions that can be performed in an unsafe codeblock can lead to memory-safety issues in safe Rust code within the same program. For example, it could overwrite a piece of memory otherwise owned by a part of the safe Rust code which in turn leads to undefined behaviour if that memory is used later on. In the paper they present a tool "Galeed" to provide run-time protection against such interactions. As this project has focused on problems relating to unsafe Rust, those issues has been deemed out of scope for

the project. Although for a future of using Rust in Linux, such issues should be taken into account.

7.2 Performance of Rust

There have also been some studies comparing the performance of C and Rust. One is, *"Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body"* from 2021 by Costanzo et al. [34], which looked at multicore high-performance computations in C, Rust and Fortran. They performed their study by benchmarking implementations of n-body physics calculations in the three languages. The study found that Rust and C are relatively evenly matched but that in some cases, C had a slight advantage of up to 18%.
The application studied by Costanzo et al. is a relatively small research application where the Rust version consists of a single file of about 220 lines. Meanwhile, the implemented driver in this project is about 4000 lines long. This is one part of the differing scopes of the projects, Costanzo et al., aimed to achieve the best possible performance from their application and as such, they have written multiple variants of the application with varying optimizations.
Our project, on the other hand, aimed to see if Rust could stand up to C in terms of performance for a real-world application to be used by consumers rather than researchers. As such, some optimization steps taken by Costanzo et al., not only in the Rust implementation but also for the compared languages, appear unrealistic to us for anything but the most performance-critical applications. Additionally, the performance metric used by Costanzo was FLOPS, and the resulting 18% performance advantage of C was attributed to the Rust compiler lacking some optimizations regarding single-precision floating point types. Since floating point operations are scarcely used in Linux, and not at all in ExFAT, those results hold little weight in the Linux domain.

7.3 Rust for other operating systems

Whilst Rust is a new language for the Linux operating system, there are several other projects exploring Rust as a language for operating systems. One such project is Redox [35], a Unix-like operating system with a micro-kernel design written in Rust.
Another is Theseus, presented by Kevin Boos in his PhD dissertation *"Theseus: Rethinking Operating Systems Structure and State Management"* [36] and further by Boos et al. in *"Theseus: an Experiment in Operating System Structure and State Management"* [37]. The design goals of Theseus include building a new variant of operating system, focusing on the Rust language and compiler and trying to empower those through the design of the operating system. Something that sets Theseus apart from other operating systems is its focus on isolation in software rather than in hardware. Furthermore, the paper includes micro-benchmark comparisons comparing the performance of Linux and Theseus. These benchmarks heavily favour

Theseus, all being at least twice as fast on Theseus and some being much faster than on Linux.
However, the paper also notes that these benchmarks are on a relatively small scale. Whilst such benchmarks are not direct benchmarks between C and Rust as the two operating systems appear to have vastly different designs, they do provide further evidence that Rust appears well-equipped for the operating system domain from a performance perspective. Apart from a focus on the language-level error-recovery and prevention that the Rust compiler can provide, the papers present efforts to recover from hardware errors and compares with how other micro-kernels such as MINIX handles similar errors. The results show that Theseus is much better equipped to recover from such errors and is often capable of unwinding and restarting rather than leading to an observable error.
In terms of security, the Theseus paper also provide some insights. It discusses the usage of unsafe Rust in the operating system as an *"Unfortunate necessity"* because of the requirement to interface directly with hardware. As previously presented, most of the unsafe usages found in this project were related to interfacing with the C part of the codebase, something that would not occur in a pure Rust operating system. However unsafe, for the purpose of interfacing with hardware, would also be required in the underlying libraries used by the driver presented in this project, should they ever be written in Rust.
Another Rust operating system is Rustpi, presented in "Rustpi: A Rust-powered Reliable Micro-kernel Operating System" [38] by Liang et al. Similarly to Theseus this OS is also based on a micro-kernel design and tries to utilize the features of Rust to improve the safety and error recovery of the system. Furthermore, Rustpi also puts some focus on error recovery in terms of avoiding crashes in favour of showing an error message to the user. One of the major features they utilize to accomplish this is the Rust drop trait to ensure that memory allocated before a potential crash is properly cleaned up during error handling. Contrary to the Theseus project however, RustPI is built solely for academic purposes and is much smaller in scope.
Whilst the efforts of the Theseus and Rustpi might seem inspiring, it is worth noting that both has yet to achieved any large usage. This is in contrast with the Linux kernel which is arguably the largest operating system kernel in the world. Furthermore, writing an operating system entirely in Rust from the ground up is very different from introducing Rust into an already existing project. However, such projects might provide insight into how to structure the Rust parts of the Linux kernel just as how the Rust for Linux project might provide insight into how to structure a Rust-only Operating system.

7.4 Other languages for Operating system development

Whilst the C language is by far the largest language for Operating system development today, there have been recent examples of other languages being used for this task. One such is Biscuit, presented in "The benefits and costs of writing a POSIX kernel in a high-level language" [39] by Cutler et al. Biscuit is an example

of a POSIX-subset compatible kernel written in a modified version of the garbage collected Go for x86-64 hardware.
Similar to many other garbage collected languages, Go generally disallows memory-unsafe actions that might be required in order to implement things on a systems level. To solve this, similar to Rust, Go has an unsafe module in its standard library which can be used for pointer arithmetics and similar unsafe operations.
Because Go has a runtime environment which expects to have a kernel, Biscuit includes a "shim"-layer to allocate memory, and other setup, for the Go runtime on which the actual kernel runs. As performance is one of the main reasons for generally not using garbage collected languages for systems-level development, the paper presents benchmarking comparisons between a few user-level applications running on Biscuit and Linux. These benchmarks shows some impressive results with Biscuit only being about 5-10% slower than Linux. However, it should be noted that the benchmarks tests relatively limited parts of the kernels and the authors point out that Linux might perform several operations more than Biscuit in some of these cases, as well as possibly sacrificing performance in favour of other use-cases.
If one ignores the caveats of Linux doing more things and supporting a vastly higher number of use-cases than Biscuit, the results are promising and even shows that garbage collected languages such as Go could be used for operating system development in terms of performance. This might lead one to question why similar efforts to Rust for Linux exists for Golang. Here, it is worth baring that the runtime nature of Go-lang requires the extra "shim" layer mentioned before which is not required for C or Rust. Furthermore, it is worth noting that the benchmarks were run for application layer applications and not specifically for kernel system calls as most other benchmarks presented both in this chapter and in this project overall.

7.5 Security of the Linux operating system

Because of the wide usage of the Linux operating system there have been many other attempts to provide general security improvements to it. One such that is integrated in the Linux kernel is Security-Enhanced Linux (SELinux), described by Loscocco and Smalley in *"Meeting Critical Security Objectives with Security-Enhanced Linux"* [40]. SELinux was first presented back in 2000 [41] and is combination of a Linux kernel subsystem and userspace applications to provide Mandatory Access Control (MAC) to the operating system.
The goal with SELinux has been to provide a flexible and general MAC implementation that can cover any and all of the use-cases that Linux might be used in. To accomplish this, SELinux utilizes a security-server that is responsible for taking security decisions for the entire system. The current implementation utilizes three different types of security mechanisms to take these decisions. First is Identity-based Access Control which generally reflects the identity of the user that runs a program, i.e. the currently logged in user. Next is Role-based access control consisting of a number of roles that each have certain permissions and a user being able to have multiple roles assigned to them at the time. Finally is Type Enforcement which is basically a matrix between types (processes and files) and roles/identities and determines if the role or identity has access to a particular type. Whilst SELinux

is integrated in Linux it can run in three different modes permissive, enforcing or disabled [42] and many Linux distributions has it disabled by default.
Comparing SELinux with the approach of utilizing Rust to address the security of Linux yields many differences but at the same time a few similarities. Both approaches try to provide overarching security improvements that covers more than singular modules or specific parts of the kernel. However, the problems that they try to solve might not necessarily overlap in most cases as protecting against memory-safety issues focuses on preventing unintended use from the kernels point of view. Contrary to this, Access-Control as provided by SELinux focuses on preventing unintended use of the system overall from a system administrators point of view. This means that preventing against memory-safety issues can also protect SELinux itself in its ability to guarantee Access-Control and as such both approaches can be used in consort to further bolster the security of a system.

7.6 Static checking of memory-safety

Due to the prevalence of memory-safety discussed in this paper, Rust is far from the first attempt to solve these issues using static checks. Many modern languages try to solve this using garbage collectors as mentioned in Chapter 1. In *“Memory Safety Without Runtime Checks or Garbage Collection”* [43], Dhurjati et al. present a custom C compiler that uses static checks to enforce certain memory-safety checks. It accomplishes this by imposing certain restrictions on the underlying program and thus only supports a subset of C programs. This approach is, of course, very similar in many ways to the idea behind the Rust programming language and as such, could provide similar benefits. However, some of the restrictions imposed by the compiler, such as requiring the program to be single-threaded, make it unviable for use in the Linux kernel. It should be noted though that the amount of work required to solve these problems is likely less than the work required to rewrite Linux entirely in Rust.

8

Conclusion

In this project, we have found that it is feasible to rewrite a driver in Linux to the Rust programming language. We have also found that memory-safety related issues cause a large portion of the security issues currently plaguing the kernel. Moreover, previous work has shown that Rust is highly capable of preventing such issues through its ownership model and rigorous compile-time checks. However, we have found that much work will have to be put into porting core parts of the Linux kernel to Rust in order to benefit fully from the security guarantees that the language can bring. Currently, the high coupling between the Rust and C codebases lead to a high number of unsafe usages which circumvent those guarantees. Regarding performance, our findings indicate that Rust appears capable of being competitive with C. With the presented benchmarks showing that at best the Rust implementation was almost 7% faster than the C implementation and at worst being 12% slower. Both our findings and the ones by Costanzo et al. have come to similar results; that Rust appears to be close, although sometimes slightly slower, than C. Although, it is worth bearing in mind that the performance measurements come from just two sources and more work has to be done within this area to draw more conclusive results. Finally, even with a slight performance penalty, we argue that the significantly reduced risk of memory-safety related security issues is well worth the effort.

Bibliography

[1] "Linux kernel source tree". (2022), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/ (visited on Jun. 1, 2022).

[2] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[3] M. S. R. Center, Ed. "A proactive approach to more secure code". (2019), [Online]. Available: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (visited on Jan. 20, 2022).

[4] chromium.org, Ed. "Memory safety". (2020), [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on Mar. 29, 2022).

[5] rust-lang.org, Ed. "Announcing rust 1.0". (2015), [Online]. Available: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html (visited on May 2, 2022).

[6] blog.mozilla.org, Ed. "Mozilla welcomes the rust foundation". (2021), [Online]. Available: https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/ (visited on May 12, 2022).

[7] blog.rust-lang.org, Ed. "Announcing rust 1.60.0". (2022), [Online]. Available: https://blog.rust-lang.org/2022/04/07/Rust-1.60.0.html (visited on May 12, 2022).

[8] rust-lang.org, Ed. "What is ownership?" (2022), [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (visited on Mar. 10, 2022).

[9] rust-lang.org, Ed. "References and borrowing". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html (visited on Mar. 10, 2022).

[10] rust-lang.org, Ed. "Checked uninitialized memory". (2022), [Online]. Available: https://doc.rust-lang.org/nomicon/checked-uninit.html (visited on Mar. 10, 2022).

[11] rust-lang.org, Ed. "The rustonomicon". (2022), [Online]. Available: https://doc.rust-lang.org/nomicon/intro.html (visited on Mar. 10, 2022).

[12] rust-lang.org, Ed. "Data types". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch03-02-data-types.html (visited on Mar. 10, 2022).

[13] rust-lang.org, Ed. "The slice type". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch04-03-slices.html (visited on Mar. 10, 2022).

[14] haskell.org, Ed. "Algebraic data type". (2022), [Online]. Available: https://wiki.haskell.org/Algebraic_data_type (visited on Mar. 10, 2022).

[15] rust-lang.org, Ed. "Enums and pattern matching". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch06-00-enums.html (visited on Mar. 10, 2022).

[16] rust-lang.org, Ed. "Generic types, traits, and lifetimes". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch10-00-generics.html (visited on Mar. 10, 2022).

[17] rust-lang.org, Ed. "Traits". (2022), [Online]. Available: https://doc.rust-lang.org/rust-by-example/trait.html (visited on Mar. 14, 2022).

[18] rust-lang.org, Ed. "Derive". (2022), [Online]. Available: https://doc.rust-lang.org/rust-by-example/trait/derive.html (visited on Mar. 14, 2022).

[19] rust-lang.org, Ed. "Foreign function interface". (2022), [Online]. Available: https://doc.rust-lang.org/nomicon/ffi.html (visited on Mar. 15, 2022).

[20] rust-lang.org, Ed. "Type layout". (2022), [Online]. Available: https://doc.rust-lang.org/reference/type-layout.html (visited on Mar. 15, 2022).

[21] rust-lang.org, Ed. "Processing a series of items with iterators". (2022), [Online]. Available: https://doc.rust-lang.org/book/ch13-02-iterators.html (visited on Mar. 10, 2022).

[22] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–25, 2021.

[23] "Github: Rust-for-linux/linux". (Mar. 25, 2022), [Online]. Available: https://github.com/Rust-for-Linux/linux (visited on Mar. 28, 2022).

[24] rust-lang.github.io, Ed. "Introduction - the 'bindgen' user guide". (Mar. 15, 2022), [Online]. Available: https://rust-lang.github.io/rust-bindgen/ (visited on Mar. 28, 2022).

[25] docs.microsoft.com, Ed. "Exfat file system specification". (2022), [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification (visited on Mar. 10, 2022).

[26] docs.microsoft.com, Ed. "Overview of fat, hpfs, and ntfs file systems". (2021), [Online]. Available: https://docs.microsoft.com/en-US/troubleshoot/windows-client/backup-and-storage/fat-hpfs-and-ntfs-file-systems (visited on Mar. 17, 2022).

[27] kernel.org, Ed. "Linux kernel makefiles". (2022), [Online]. Available: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html (visited on Apr. 26, 2022).

[28] kernel.org, Ed. "Overview of the linux virtual file system". (2022), [Online]. Available: `https://www.kernel.org/doc/html/latest/filesystems/vfs.html` (visited on Apr. 26, 2022).

[29] nvd.nist.org, Ed. "Nvd - vulnerability metrics". (2019), [Online]. Available: `https://nvd.nist.gov/vuln-metrics/cvss` (visited on May 11, 2022).

[30] cvedetails.com, Ed. "Linux kernel : Security vulnerabilities published in 2021". (2022), [Online]. Available: `https://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&version_id=&page=1irefox&hasexp=0&opdos=0&opec=0&opov=0&opcsrf=0&opgpriv=0&opsqli=0&opxss=0&opdirt=0&opmemc=0&ophttprs=0&opbyp=0&opfileinc=0&opginf=0&cvssscoremin=0&cvssscoremax=0&year=2021&month=0&cweid=0&order=3&trc=159&sha=28a2f51a672c94e4f5764cd0848d032c1497207f` (visited on May 11, 2022).

[31] B. Stroustrup, "Abstraction and the c++ machine model", in *Embedded Software and Systems*, Z. Wu, C. Chen, M. Guo, and J. Bu, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–13, ISBN: 978-3-540-31823-1.

[32] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs", in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[33] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping safe rust safe with galeed", in *Annual Computer Security Applications Conference*, 2021, pp. 824–836.

[34] M. Costanzo, E. Rucci, M. Naiouf, and A. D. Giusti, "Performance vs programming effort between rust and c on multicore architectures: Case study in n-body", in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10. DOI: `10.1109/CLEI53233.2021.9640225`.

[35] redox-os.org, Ed. "Redox". (2022), [Online]. Available: `https://www.redox-os.org/` (visited on Sep. 15, 2022).

[36] K. A. Boos, "Theseus: Rethinking operating systems structure and state management", Ph.D. dissertation, Rice University, 2020.

[37] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: An experiment in operating system structure and state management", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1–19, ISBN: 978-1-939133-19-9. [Online]. Available: `https://www.usenix.org/conference/osdi20/presentation/boos`.

[38] Y. Liang, L. Wang, S. Li, and B. Jiang, "Rustpi: A rust-powered reliable microkernel operating system", in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2021, pp. 272–273.

[39] C. Cutler, M. F. Kaashoek, and R. T. Morris, “The benefits and costs of writing a POSIX kernel in a high-level language”, in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 89–105, isbn: 978-1-939133-08-3. [Online]. Available: `https://www.usenix.org/conference/osdi18/presentation/cutler`.

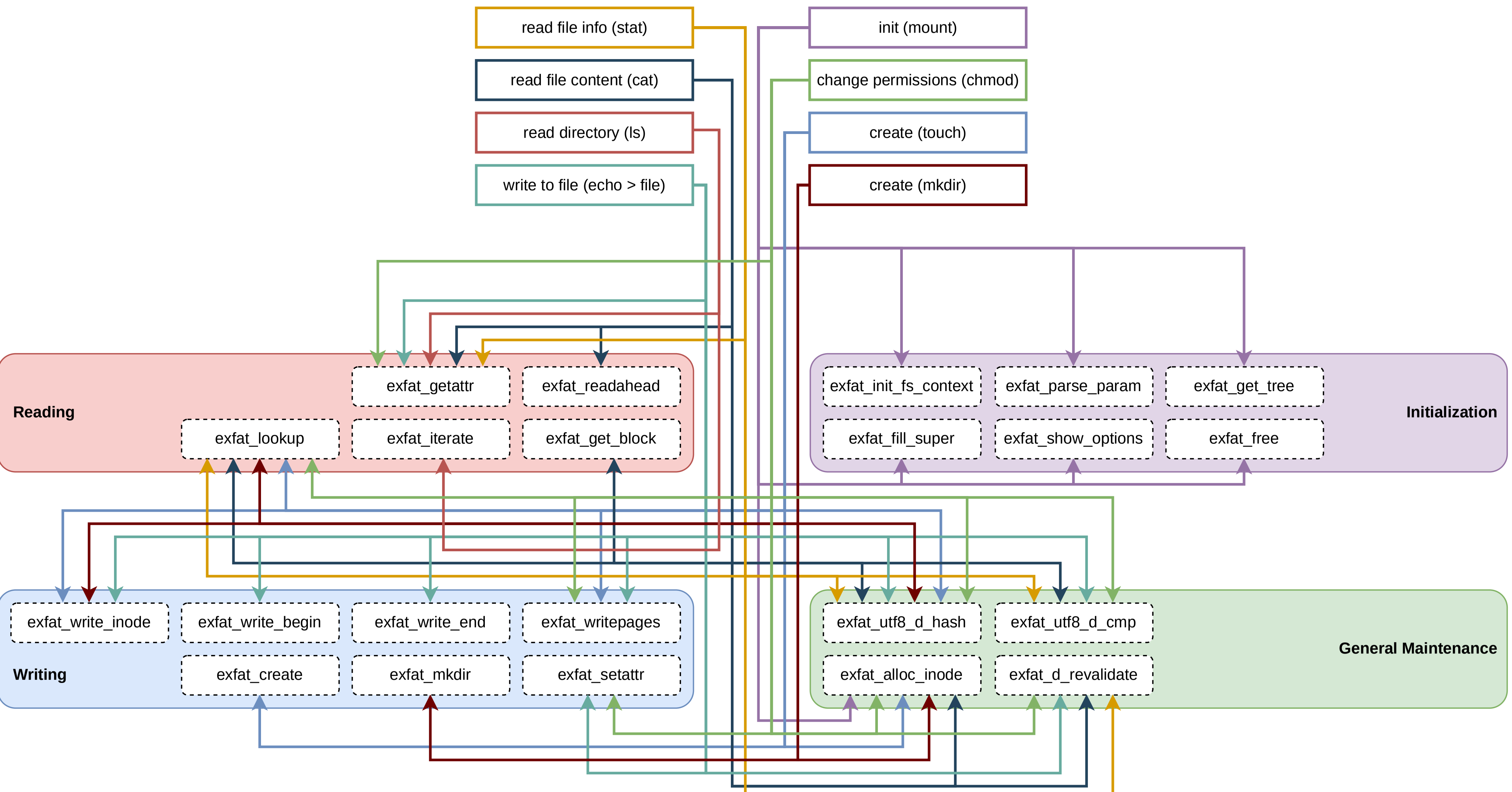
[40] P. Loscocco and S. D. Smalley, “Meeting critical security objectives with security-enhanced linux”, in *Proceedings of the 2001 Ottawa Linux symposium*, vol. 27, 2001, pp. 303–314.

[41] P. Loscocco. “Selinux”. (2000), [Online]. Available: `https://marc.info/?l=linux-kernel&m=97749381725894` (visited on Sep. 24, 2022).

[42] redhead.com, Ed. “What is selinux?” (2019), [Online]. Available: `https://www.redhat.com/en/topics/linux/what-is-selinux` (visited on Sep. 24, 2022).

[43] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, “Memory safety without runtime checks or garbage collection”, in *Proceedings of the 2003 ACM SIGPLAN conference on language, compiler, and tool for embedded systems*, 2003, pp. 69–80.

A
Structure of the existing Linux exFAT driver


read file info (stat)
read file content (cat)
read directory (ls)
write to file (echo > file)
init (mount)
change permissions (chmod)
create (touch)
create (mkdir)
Reading
exfat_getattr
exfat_readahead
exfat_lookup
exfat_iterate
exfat_get_block
Initialization
exfat_init_fs_context
exfat_parse_param
exfat_get_tree
exfat_fill_super
exfat_show_options
exfat_free
Writing
exfat_write_inode
exfat_write_begin
exfat_write_end
exfat_writepages
exfat_create
exfat_mkdir
exfat_setattr
General Maintenance
exfat_utf8_d_hash
exfat_utf8_d_cmp
exfat_alloc_inode
exfat_d_revalidate

B
CVE Classifications

CVE	Score	Cause/problem	Memory-safety related?
CVE-2021-43267	7.5	Out-of-bounds read/write	Yes
CVE-2010-2525	7.2	Invalid access management	No
CVE-2019-25044	7.2	Use-after-free	Yes
CVE-2020-25669	7.2	Use after free	Yes
CVE-2020-25670	7.2	Use after free	Yes
CVE-2020-25671	7.2	Use after free	Yes
CVE-2020-35499	7.2	Null pointer dereference	Yes
CVE-2020-36158	7.2	Out-of-bounds read/write	Yes
CVE-2020-36387	7.2	Use-after-free	Yes
CVE-2021-3347	7.2	Use-after-free	Yes
CVE-2021-3489	7.2	Out-of-bounds read/write	Yes
CVE-2021-3490	7.2	Logic error	No
CVE-2021-3491	7.2	Out-of-bounds read/write	Yes
CVE-2021-3612	7.2	Out-of-bounds read/write	Yes
CVE-2021-20292	7.2	Use-after-free	Yes
CVE-2021-28375	7.2	Invalid access management	No
CVE-2021-28660	7.2	Out-of-bounds read/write	Yes
CVE-2021-28972	7.2	Out-of-bounds read/write	Yes
CVE-2021-29154	7.2	Incorrect calculation	No
CVE-2021-29266	7.2	Use-after-free	Yes
CVE-2021-32606	7.2	Use-after-free	Yes
CVE-2021-33200	7.2	Out-of-bounds read/write	Yes
CVE-2021-33909	7.2	Out-of-bounds read/write	Yes
CVE-2021-37576	7.2	Out-of-bounds read/write	Yes
CVE-2021-38160	7.2	Out-of-bounds read/write	Yes
CVE-2021-38300	7.2	Logic error	No
CVE-2021-41073	7.2	Use-after-free	Yes
CVE-2021-43057	7.2	Use-after-free	Yes
CVE-2020-25668	6.9	Out-of-bounds read/write	Yes
CVE-2021-3573	6.9	Use-after-free	Yes
CVE-2021-23133	6.9	Race condition	No
CVE-2021-26708	6.9	Race condition	No
CVE-2021-29657	6.9	Use-after-free	Yes

CVE	Score	Cause/problem	Memory-safety related?
CVE-2021-31440	6.9	Logic error	No
CVE-2021-35039	6.9	Logic error	No
CVE-2021-42008	6.9	Out-of-bounds read/write	Yes
CVE-2020-35519	6.8	Out-of-bounds read/write	Yes
CVE-2020-36385	6.8	Use-after-free	Yes
CVE-2021-32078	6.6	Out-of-bounds read/write	Yes
CVE-2020-27815	6.1	Out-of-bounds read/write	Yes
CVE-2021-3653	6.1	Logic error	No
CVE-2021-20226	6.1	Use-after-free	Yes
CVE-2021-31916	6.1	Out-of-bounds read/write	Yes
CVE-2020-36386	5.6	Out-of-bounds read/write	Yes
CVE-2021-3506	5.6	Out-of-bounds read/write	Yes
CVE-2020-28374	5.5	Invalid access management	No
CVE-2021-3178	5.5	Logic error	No
CVE-2002-2438	5.0	Logic error	No
CVE-2020-25672	5.0	Memory leak	Yes
CVE-2021-38201	5.0	Out-of-bounds read/write	Yes
CVE-2021-38202	5.0	Out-of-bounds read/write	Yes
CVE-2021-38207	5.0	Out-of-bounds read/write	Yes
CVE-2021-45485	5.0	Logic error	No