Fakultät Informatik

Paper title

Bachelorarbeit im Studiengang Informatik

vorgelegt von

Florian Meißner

Matrikelnummer: 3210376

Erstgutachter: Prof. Dr. Hans Löhr
Zweitgutachter: Prof. Dr. Michael Zapf

Kapitel 1
Introduction

- Rust is increasing usage in system level
- but still many big projects (linux etc.) are written in C

Since the beginning of programming, there has been a discrepancy between the input states an interface formally accepts, and the input states that are sound to handle. For example, a reciprocal function $f(x) = 1/x$ might formally accept a 32 bit integer — and therefore all of its $2 \hat{} 32$ input states —, but the mathematical formula it tries to model will not give a sensible result for $x = 0$; least of all if the function in turn returns an integer, since there is no integer n: $1 / x = n$.

One straightforward solution has always been to limit the function domain via documentation. Users of that function are expected to read that documentation and recognize that it is a violation of interface contract to call it with $x = 0$. Violation of that contract would in turn result in an error, a crash, or — worse yet — Undefined Behaviour. An approach with drawbacks, as there are now two sources of truth about the function domain. One of these — the function signature, expressed in code — is already technically incorrect, as we have not stated a way to express "integers without zero" as a valid type. Additionally, as the program developes, the chance of both sources of truth to get further out-of-sync increases. This discrepancy,between the high-level contract, expressible only in additional information in text form, and the function signature the compiler handles, raises the following question: Can we encode this precondition in such a way that the function signature makes it impossible to pass in values that violate any API contract?

In languages where we have strict and strong typing , we can enforce invariants about types we create, since we have to explicitly provide the methods of construction for these types, and we can make then fail if some invariants are not upheld. This allows us to express function signatures of the kind discussed previously, by creating a new type `NonZeroI32`Listing 1 that represents the idea of an integer that cannot be zero. By making the inner value private, we then ensure that users of our library are forced to use the only construction method we provide them with, `NonZeroI32::new(i32)`. This `new()` function can be total, since it returns a result value representing a fallible computation. In that sense, the function signature of `new()` expresses that **every 32-bit integer is either a valid non-zero 32-bit integer or an error**.

[1]

Kapitel 2
Background

2.1 Rust

- Modern language with many features that can increase correctness and safety
- is marketed/intended as a low level language, usually competes near C in benchmarks
  - Rust in Linux kernel
  - RedoxOS
  - coreutils / libc rewrite
- features
  - borrow checker / lifetime tracking / ownership tracking
  - strict types , (almost) no implicit conversions
  - RAII / destructors / `Drop` trait
  - no data races
  - fearless concurrency
  - error handling
  - ADTs / modeling complex types
  - generics
  - typestate pattern
- Unsafe rust
  - **what we keep, what we lose**
  - additional promises to uphold
  - (ausblick: additional tools, static analysis, sanitizers etc.)

2.2 FUSE

- filesystem as process in userspace
- don't have to build kernel modules (safer, easier dev workflow)
- should be comparable though (why? give reasons)
- architecture ()

- fuse kernel module
- libfuse
- FS impl
- **=> our layer**

```rust
pub struct NonZeroI32(i32);

impl NonZeroI32 {
  pub fn new(n: i32) -> Result<Self> {
    if n == 0 {
      Err("n == 0 is not allowed!")
    } else {
      Ok(n)
    }
  }
}

pub fn reciprocal(n: NonZeroI32) -> NonZeroI32 {
  // ...
}
```

Listing 1: A new type `NonZeroI32` that represents the idea of a 32-bit integer **guaranteed** to not be zero

Kapitel 3
Related work

3.1 `rust-fatfs`[2]

A Rust reimplementation of the FAT filesystem standard created by Microsoft. It is implemented as a kernel module without usage of FUSE. Although the authors claim exploring security and safety benefits as motivation, the evaluation focuses on performance, benchmarking the work against the established C kernel module.

3.2 `fuser`

3.3 Rust for Linux[3]

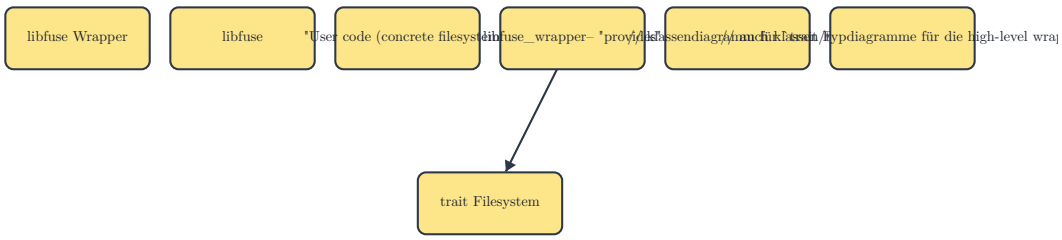
- rust-fatfs
- fuser
  - auch in rust
  - fuse LowLevel statt "normal" API (mine)
  - **aber**: verwendet eh niemand
- rust in linux kernel

Kapitel 4
Concept

[4], [5], [6], [7], [8]

- read similar rust projects, get idea about how the structure and approach would look for the libfuse bindings
- read up about `cbindgen` by mozilla (will def. need to use it)
- read up about theoretical foundation of type systems and using them to encode programmer contracts
- for every libfuse API call:
  - decide if in-scope
  - enumerate a list of (sensible) contracts
  - encode through type system
    - if that fails or becomes too hard, skip them and document that
- (if possible) collect filesystem related CVEs from databases
- (else) CWEs allgemein sammeln
- match CVEs/CWEs with libfuse calls, find potential weaknesses/threads
- evaluate if my rust constructs can fix those weaknesses. if not, try to improve bindings.
- create stats and tables (e.g. percentage CVEs prevented, taken from a) sub section X, b) time span Y, etc.)
- write introduction with foundational concepts

Kapitel 5
Implementation



5.1 Basic C interop

Safe Rust (Rust) can never (sans compiler errors) cause UB in the resulting binary program. In unsafe Rust, this is not the case; the programmer now has to uphold several invariants to ensure soundness (soundness). In the C standard, where behavior in any situation not explicitly defined by the language standard is implicitly "undefined", Rust limits these invariants to a set of specific, well-documented cases. This makes reviewing the soundness property of unsafe code easier.

5.1.1 Pointers

Regarding use of raw pointers in unsafe Rust, the following invariants exist:

1. No dereferencing of dangling or unaligned pointers.
2. Respect aliasing rules: no pointer is allowed to point to memory that's also pointed-to by a mutable reference, since a mutable reference in Rust is guaranteed to be exclusive.
3. Respect immutability: no pointer is allowed to modify data that's also pointed-to by a shared reference, since a value behind a shared reference is guaranteed not to change.
4. Values in memory must be valid for their respective types: pointers must not be used to change the representation in memory of to a value — or reference — to a state which is not valid for the type this value — or reference — has. E.g. a `NonZeroU8`, represented in memory

as a `u8`, will have one bit pattern that would correspond to a numeric zero and is therefore illegal.

Because TODO libfuse (libfuse) calls all our callbacks with at least one C pointer, we have to check these invariants as rigorously as possible before we call into user code, if we want to eliminate them as sources of UB.

1. We have to differentiate between three cases:
   - **Unaligned pointer**: this is easy, as Rust provides `ptr::is_aligned()`, which takes a pointer and detects misalignment.
   - **Dangling null-pointer**: this is also easy, both manually and through the Rust-provided `ptr::is_null()`, which takes a pointer and detects null-ness.
   - **Dangling non-null pointer**: this happens when a pointer is used-after-free or if pointer arithmetic goes wrong, and is much harder to avoid. Since we don't control memory allocation in C, we largely have to trust C code to not pass us pointers from this category. This would cause UB and should therefore be documented visibly as soundness assumption.
2. For pointers passed to us by libfuse, the solution is simply to not create a reference to it. If it is necessary to pass a mutable reference into user code, an intermediate owned value must be created, and the target value must be copied in and out of that intermediate.
3. When dealing with non-const pointers, care must be taken to not create a shared reference to it. Const pointers don't matter for that aspect, since it is impossible to modify values through them, given they are not cast to non-const pointers.
4. This only matters when primitive C-style casts or `mem::transmute()` are used, as otherwise the Rust typesystem protects us from writing values of the wrong type, even inside unsafe blocks. Writing to a pointer can involve writing raw bytes; if that is required, extra care must be taken, and it is therefore usually better to avoid this.

5.1.2 Strings and Unicode

Rust's native string types (`str`, `String`) exclusively store UTF-8.[9, ch. 8.2] The main kind of strings this library needs to handle are the file paths that filesystem callbacks are called on. The encoding of those is platform-dependent, usually being C-like ASCII strings on Unix-like systems and UTF-16 on newer Windows versions. Correctly detecting and handling string encodings is a hard problem , and since UTF-8 is a superset of ASCII, we chose to not handle UTF-16 or other cases and emit an error when encountering non-UTF-8 input. This limits the complexity of the prototype without limiting the scope of the reseach question.

5.1.3 Unwinding across FFI boundaries

- => is UB

- have to wrap every possible panic point inside `catch_unwind()`
- not provably panic-free with just compiler
  - but there is an interesting crate: `https://github.com/dtolnay/no-panic` => **future work**

When a Rust program is compiled with stack unwinding support and a panic is triggered, the default uwind handler will walk up the stack in order to react to the panic, collecting debug information or cleaning up data. In a program using FFI, this can lead to crossing into another language runtime while walking the stack. Doing so correctly is a non-trivial task and can easily lead to UB.[10, ch. 14] On the other hand, turning unwinding off loses helpful stack traces and debug information when a panic happens. We therefore decided to keep unwinding behaviour while preventing any panic from propagating across a FFI boundary.

Every function that is visible to C can potentially be called from an environment where unwinding works differently or not at all. Therefore each of those functions must be panic-free. As of now, there is no compiler flag or lint that detects or prevents use of panicking functions, operators or language keywords. As a result, this must be done manually by reviewing the source code of the functions in question, and, recursively, the functions they call. A convention exists to note possible panics in a section of the function documentation, but even the standard library doesn't consistenly follow it.

One crate that tackles this problem is `no_panic`[1] by David Tolnay, a prominent figure amongst the Rust community. It provides the ability to annotate function declarations with an attribute macro, and promises to halt the compilation with an error if the function is **not provably panic-free**. This implies that it is possible to write functions that would not panic, but would still not compile if the compiler is unable to prove that property. The crate thereby takes a stance typical of Rust philosophy: it is preferable to reject sound programs, than to accept unsound ones.

While preventing panics in self-maintained code requires careful manual analysis, this is not possible for user-provided functions. For this, there exists a function `panic::catch_unwind()`[11] that takes a closure and executes it, catching any unwind that would occur and returning an error instead. Wrapping the call to user code inside this function ensures that no panic will be propagated up the call stack from this point on.Listing 2

[1] https://docs.rs/no-panic/latest/no_panic/

```rust
fn call_into_user_code<FS: Filesystem, T>(
    method: &str,
    user_fn: impl FnOnce() -> Result<T, Errno>,
) -> Result<T, (String, Errno)> {
    let fs = std::any::type_name::<FS>();
    std::panic::catch_unwind(core::panic::AssertUnwindSafe(user_fn))
        .map_err(|panic| {
            // abort, since internal state of filesystem impl can now be inconsistent
            state::clear();
            (
                format!("PANIC on `{fs}::{method}`:\n\n{panic:?}\n"),
                Errno::ENOTRECOVERABLE,
            )
        })
        .and_then(|inner| inner.map_err(|e| (format!("Error in user code `{fs}::{method}
`"), e)))
}
```

Listing 2: An exemplary trampoline function signature implementing compile-time static dispatch via generics

5.2 FUSE operations

- these 4 functions seem to be the bare minimum for a R/O filesystem (see libfuse example `hello`)
- open can be a noop

5.2.1 getattr

- "bread and butter" call, is the first one executed on all filesystem paths, lets user decide how to continue (readdir on dirs, read/open on files e.g.)
-

5.2.2 readdir

5.2.3 open

5.2.4 read

5.3 Initialization and Global State Management

- We need to supply a number of C functions that know which user impl to call
- Possibility: just use data pointer from `libfuse::init`

    - Con: push raw pointers around, prone to corruption
- My choice: use generics, overload generic trampolines with user Filesystem type
    - that way, the compiler generates a concrete version (with its own address and hard-coded user code address) of our generic trampolines
    - since generic parameter is the only type with `impl Filesystem` in scope, type system prevents any confusion/programmer error.
    - since compiler generates hardcoded version, memory corruption due to logic errors anywhere is also not a problem
    - con: only one instance per struct type per process.
        - workaround: just use wrapper structs (can be done easily from user code)

Since the libfuse initialization routine takes a struct of callback function pointers (`fuse_ops`), that creates the following problem. Since the C signature is predetermined, user functions cannot be used, because that would force signatures of user functions to use the lower-level C types which we try to avoid. That means, even though there is a one-to-one correspondence between FUSE operation callbacks and trait methods on the `Filesystem` trait, they are not compatible and cannot be used interchangably. The obvious approach is to provide **trampoline functions (trampoline_functions)**, which then wrap, transform and safety-check the C type values on call and dispatch into user code. A non-trivial problem, one that is not obvious at first sight, is how the trampoline knows which filesystem implementation to dispatch to. There are two basic options how to use the trampolines:

1. Use one global trampoline per callback, and somehow transport the choice on which filesystem to use inside the C arguments that **the libfuse wrapper (libfuse_wrapper)** gets passed by **libfuse**.
2. Somehow generate a set of trampolines per user filesystem, which are then hard-coded towards the specific filesystem implementation.

A way to implement option 1 is provided in the form of a `void *private_data` pointer that can be passed to **libfuse** during filesystem registration. This pointer can contain arbitrary user-specified data, and is not used by **libfuse** except for making it available to every fuse operation via the `fuse_get_context`[2] function.

Since it is possible to store a Rust pointer inside a C void pointer, **libfuse_wrapper** can submit a pointer to the user implementation as payload for `private_data`, then let each trampoline poll the FUSE context struct, cast the void pointer back to a trait object reference and dispatch into the corresponding trait method. This has the following disadvantages:

- Decaying a managed Rust reference into a raw pointer loses the advantage of lifetime tracking that is one of Rusts fortes in the undertaking of creating safe systems-level code. Manual care has to be taken not to invoke a use-after-free, accessing an uninitialized or unauthorized memory location or — in the best case — simply leaking memory. In fact, the safest option

[2]https://libfuse.github.io/doxygen/fuse_8h.html#a5fce94a5343884568736b6e0e2855b0e

would be to initialize this data pointer once, and then never free it, since it is the dealloc part that introduces memory unsafety to a system , and even if leaking memory (and not calling destructors) is acceptable, since libfuse passes around non-const pointers to everything, bugs at any point of both our trampolines and libfuse can easily lead to access of corrupted pointers and therefore to UB. This is usually a tradeoff that must be accepted when dealing with FFI into unsafe languages, but should be mitigated whenever feasible.

Both disadvantages would in theory prevented by a solution after option 2, and thankfully, with the use of generics, Rust brings includes the tools to implement such a solution. As seen in Listing 3, this exemplary trampoline function is generic over types implementing our `Filesystem` trait. This leads the Rust compiler to generate a concrete, independent `getattr` trampoline function for every trait implementation of `Filesystem` that is used to call our initialization function. The generic approach is then combined with a singleton registry[3] which provides a global map of values, indexed by types. We can now store the concrete user-supplied filesystem struct inside this registry and use the type of this filesystem struct as index, which additionally will be deduced implicitly by the compiler from the argument types of our initialization function. That means, given there are no other implemenations of our `Filesystem` trait in scope when declaring the generic functions, the type system guarantees us that the user's type is the only one that can be used for dispatching, shielding even against potential programmer oversight.

This has the drawback of only allowing one instance of a concrete `Filesystem` type to be mounted per process. But since — if needed — newtype structs (newtype_structs) can be used to create different concrete types with minimal boilerplate, this was deemed tolerable.

```
pub unsafe extern "C" fn getattr<FS: Filesystem>(
    path: *const i8,
    stat_out: *mut libfuse::stat,
    _fuse_file_info_out: *mut libfuse::fuse_file_info,
) -> i32 {
```

Listing 3: An exemplary trampoline function signature implementing compile-time static dispatch via generics

5.4 Type modeling

Creating thin high-level representations of the low-level data types that make up the libfuse API, that nonetheless verify as many correctness properties as possible, is the main focus of this project. Where feasible, these properties are checked during compile-time, which gives the additional advantage of not impacting runtime performance. Otherwise, runtime checks are emitted to still provide correctness to a very high degree. It would be common practice in low-level Rust crates to provide `*_unchecked()` variants for these runtime-checked methods, to give users the choice of

[3] https://crates.io/crates/singleton-registry

circumventing those checks and trading performance for possible UB. Due to the goals of this work, and time constraints, this was mostly skipped.

5.4.1 `stat`

The libfuse `stat` struct is very similar to the namesake found in the POSIX standard (POSIX). Both describe an entry in an abstract filesystem, and contain most of its attributes. This set of attributes is needed for most interaction, because it provides data not limited to: the type of the entry — file, directory, symbolic link or other — it's permissions, size and modification dates. It is usually the set of information our wrapper has to provide to the surrounding system when some interaction with the filesystem takes place, e.g. listing or changing into a directory, or opening a file. [12]

Our attempt at modeling lead us to break down the struct into smaller parts, which require more attention:

- `FileType`, which is an enum flag of several possible values that have to specifically match magic IDs from the corresponding C header.
- `FilePermissions`, which are stored as a positive integer and usually displayed as an octal number in the range of `0o000` to `0o777` and represent restrictions on reading, writing and executing the underlying entry.
- three bitflags (`setuid`, `setgid`, and `vtx_flag`), that are context-dependent and enable additional features. These are stored inside the permissions integer in the underlying Unix APIs.

Other fields, like file size and modification time, were not deemed as interesting, since it can be correct for them to assume every valid bit pattern the underlying C type can represent, and checking the correctness semantically would introduce significant runtime overhead. E.g. validating modification time would have to detect modification in arbitrary files, and file size is an attribute that the wrapper has no insight into as per abstraction.

5.4.2 `fuse_file_info`

5.4.3 `FileMode`

5.4.3.1 Typed builder

- question: how do I model type creation?
  - free function: no named parameters, gets unreadable quickly, no optional parameters
  - struct init: grundsätzlich recht sicher, aber
    - pro: parameter sind benannt
    - manche felder mandatory, manche optional: geht nicht
    - struct muss default trait implementieren, dann sind alle felder basically optional, und es ist möglich, potentiell invalide objekte zu erstellen

        - keine schicken auto-converts und transformations, bounds checking etc.
    - "normal" runtime builder
        - pro: sehr flexibel, ergonomisch
        - con: wird ein mandatory feld vergessen, gibts erst zur runtime nen fehler
    - typed builder:
        - pro: flexibilität und mächtigkeit eines runtime builders, trotzdem werden fehler schon zur compilezeit gefangen
        - con: state ist im typ encodiert, macht es schwer bis unmöglich (type erasure stunts), z.b. in einer if-bedingung konditional ein feld zu setzen
- da für jeden einsatzzweck ein anderes pattern optimal sein kann, habe ich mehrere für meine struct(s) implementiert

5.4.4 `OpenFlags`

5.5 Error handling

Kapitel 6
Evaluation

[13]

6.1 A prototype filesystem implementation: `hello2`

To test our wrapper library, we created a minimal filesystem using it. It implements only three callbacks — `getattr`, `read`, `open` — as this is enough to provide a complete, usable filesystem. [14, /example_2hello_8c.html]

This filesystem is read-only, since that narrows down the functionality we have to implement. Files are declared in a static global array, and are even associated with a closure object, to facilitate files with dynamic content. The following example shows a global file table of two entries: `time.txt`, which always reads the current system date and time, and `pid.txt`, which always reads the ID of the filesystem process.Listing 4

Besides some boilerplate to iterate over files in a folder, the only logic consists of the block `impl Filesystem for Hello2`, where we implement methods on libfuse_wrapper's filesystem trait. The implementations were straight-forward and simple, which which was one of libfuse_wrapper. Most low-level details and pitfalls were abstracted away. One aspect that required a proportionally high amount of SLoC was dealing with partial and offset reads, but offloading that to the wrapper would mean that the user has to provide an array with the full file content already contained, only for the wrapper to calculate the correct offsets. This could be made possible as an additional opt-in API, but would almost certainly result in major inefficiencies, as the filesystem has to procure the whole file's content every time a partial read is requested.

```rust
static FILES: LazyLock<[FileEntry; 6]> = LazyLock::new(|| {
    [
        ("/pid", Box::new(|| std::process::id().to_string())),
        (
            "/time",
            Box::new(|| format!("{}", chrono::Local::now().format("%c"))),
        ),
    // …
```

Listing 4: A global file table for our `hello2` example filesystem

Kapitel 7
Conclusion

7.1 Limitations

Kapitel 8
Future work

Bibliography

[1] W. Bugden and A. Alahmar, "Rust: The Programming Language for Safety and Performance." [Online]. Available: https://arxiv.org/abs/2206.05503

[2] S. Oikawa, "The Experience of Developing a FAT File System Module in the Rust Programming Language," in *Software Engineering, Artificial Intelligence, Networking and Parallel/ Distributed Computing*, R. Lee, Ed., Cham: Springer International Publishing, 2023, pp. 45–58. doi: 10.1007/978-3-031-19604-1_4.

[3] "Rust for Linux." Accessed: Feb. 18, 2026. [Online]. Available: https://rust-for-linux.com/

[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in Rust: The promise and the challenge," *Communications of the ACM*, vol. 64, no. 4, pp. 144–152, 2020.

[5] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System Programming in Rust: Beyond Safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, in HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 156–161. doi: 10.1145/3102980.3103006.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019, doi: 10.1145/3360573.

[7] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020, doi: 10.1145/3428204.

[8] L. Seidel and J. Beier, "Bringing Rust to Safety-Critical Systems in Space," in *2024 Security for Space Systems (3S)*, 2024, pp. 1–8. doi: 10.23919/3S60530.2024.10592287.

[9] S. Klabnik, C. Nichols, C. Krycho, and Rust Community, "The Rust Programming Language." Accessed: Feb. 18, 2026. [Online]. Available: https://doc.rust-lang.org/book/

[10] The Rust Project Developers, "The Rust Reference." Accessed: Feb. 18, 2026. [Online]. Available: https://doc.rust-lang.org/1.92.0/reference/

[11] The Rust Project Developers, "The Rust Standard Library." Accessed: Feb. 18, 2026. [Online]. Available: https://doc.rust-lang.org/1.92.0/std/

[12] "stat(3type) — Linux manual page." May 02, 2024.

[13] The MITRE Corporation, "2025 CWE Top 25 Most Dangerous Software Weaknesses." Accessed: Feb. 13, 2026. [Online]. Available: https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

[14] “libfuse API documentation.” Accessed: Feb. 21, 2026. [Online]. Available: https://libfuse.github.io/doxygen/

Kapitel 9
Glossary

***POSIX* – the POSIX standard** 15

***Rust* – Rust**: TODO 9, 9

***UB* – Undefined Behaviour**: TODO 1, 9, 10, 11, 14, 15

***libfuse* – TODO libfuse**: TODO 10, 10, 13, 13, 13, 14, 14, 14, 15

***libfuse_wrapper* – the libfuse wrapper**: TODO - our lil' project 13, 13, 17, 17

***newtype_struct* – newtype struct**: TODO 14

***soundness* – soundness**: TODO define & quote 9, 9

***trampoline_function* – trampoline function**: TODO 13