# Contents



# 1 Outline

1. Introduction
2. Motivation
3. Review of similar solutions
4. Methodology
5. Implementation
6. Evaluation
7. Conclusion
8. Future work

# 2 Introduction

[1]

# 3 Motivation

[1], [2], [3], [4]

- Low level APIs are dangerous when misused (by concept)
- Documentation is rarely read completely and correctly, and rarely updated consistently
- Would be nice if Compiler could enforce correct usage

- you (usually) need a strong type system for that
- Rust provides that and is usable as system language
- (see linux kernel efforts to move rust into the project, especially in filesystems area)
- can CVEs be effectively prevented?
- (Or, if non-exploitable, can crashes be prevented?)

# 4 Review of similar solutions

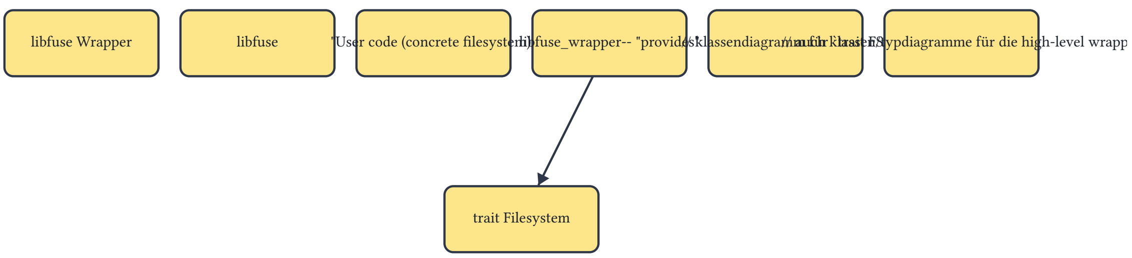
- rust-fatfs
- fuser
  - auch in rust
  - fuse LowLevel statt “normal” API (mine)
  - **aber**: verwendet eh niemand
- rust in linux kernel

# 5 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

# 6 Implementation



## 6.1 Basic C interop

### 6.1.1 Pointers

- for every C pointer we have to use
  - is it aligned?
  - is it non-null?
  - (is it valid? not checkable without allocator control or sanitizer or sim.)

### 6.1.2 Strings and Unicode

- rust only allows UTF-8 Strings
- although there are wrappers for C-style strings, most APIs are built to only work on Unicode
- => we disallow non-UTF-8 strings for simplicity

### 6.1.3 Panics 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**

## 6.2 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

### 6.2.1 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.)
- 

### 6.2.2 readdir

### 6.2.3 open

### 6.2.4 read

## 6.3 Initialization / 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)

## 6.4 Type modeling

### 6.4.1 `stat`

- gives basic info about FS entry

- returned by getattr
- 

#### 6.4.2 `fuse_file_info`

#### 6.4.3 FileMode

##### 6.4.3.1 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

#### 6.4.4 OpenFlags

### 6.5 Error handling

# 7 Evaluation

[9]

### 7.1 Beispiel-FS `hello2`

# 8 Conclusion

# 9 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] □□□, "A Study of Memory Safety in Unsafe and Safe Languages," 2024.

[3] I. Bang, M. Kayondo, H. Moon, and Y. Paek, "TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 6947–6964. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/bang

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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] 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