

Rust in Kernel Development



Hooriyeh Maaleki



Why Rust for The Linux Kernel?

Motivation

C is powerful but error-prone: buffer overflows, null dereferencing, use-after-free.
Increasing codebase complexity makes it harder to ensure safety manually.
Linux aims to reduce security vulnerabilities.

Rust as a Solution

Rust offers memory and thread safety without garbage collection.
Strong compiler checks eliminate many runtime bugs.
Proven adoption in critical software systems (e.g., Firefox, Android).

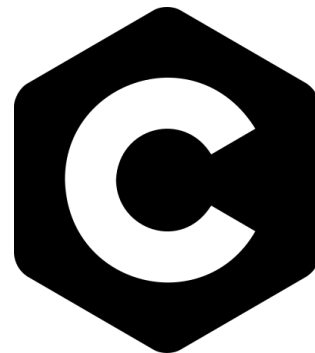
Quote

Linus Torvalds (2022): "Rust has real potential."

Memory Safety Challenges in C

Types of Memory Errors

- Buffer overflows
- Use-after-free
- Null pointer dereference
- Dangling pointers
- Double free
- Memory leaks
- Stack smashing
- Out-of-bounds access



Memory Error Type	Description	How Rust Prevents It
Buffer Overflow	Writing outside allocated memory bounds	Compile-time checks, safe array access
Use-after-Free	Accessing memory after it's freed	Ownership system, borrow checker
Null Pointer Dereference	Dereferencing a null pointer	Option types, no implicit null
Dangling Pointer	Pointer to deallocated memory	Ownership system
Double Free	Freeing memory more than once	Ownership model prevents double free
Memory Leak	Allocated memory never freed	Automatic RAI memory management
Stack Smashing	Overflow corrupting stack metadata	Bounds checking on stack structures
Out-of-Bounds Access	Indexing past array limits	Index checks with panic on violation



```
// Buffer overflow
char buf[8];
strcpy(buf, "this is too long");

// Use-after-free
char *p = malloc(10);
free(p);
*p = 42;

// Null pointer dereference
int *ptr = NULL;
*ptr = 5;

// Dangling pointer
int *dangling;
{
    int temp = 42;
    dangling = &temp;
}
*dangling = 99;
```



```
// Double free
int *df = malloc(4);
free(df);
free(df);

// Memory leak
int *leak = malloc(100);
// forgot to free(leak);

// Stack smashing
char stack_buf[4];
strcpy(stack_buf, "overflow");

// Out-of-bounds access
int arr[3] = {1,2,3};
int x = arr[5];
```



Rust's Advantages for Systems Programming

Memory safety without garbage collector

Rust enforces safety at compile-time through ownership and lifetime checks, ensuring predictable performance without the need for runtime garbage collection.

Ownership and borrowing

These concepts ensure that only one mutable reference exists at a time, avoiding data races and use-after-free bugs.

Zero-cost abstractions

Rust's high-level constructs (e.g., iterators, pattern matching) compile down to efficient machine code, offering C-like performance without sacrificing clarity or safety.



Rust's Advantages for Systems Programming

Strong static typing

Rust's rich type system catches errors at compile time and enables safer abstractions like enums with exhaustive pattern matching.

Compile-time guarantees

Rust detects many issues during compilation—null references, uninitialized variables, and improper memory access—reducing runtime crashes.

Immutability by default

Data is immutable unless explicitly declared mutable, which prevents accidental state changes and makes concurrent programming safer.



Timeline of Rust in the Kernel

2019: Initial discussions about adopting Rust began on LKML (Linux Kernel Mailing List). Prototypes started showing how Rust might integrate with the kernel's build system and architecture.

2020: The rust-for-linux project was publicly announced. Early contributions explored creating idiomatic abstractions for common kernel patterns. Linux Plumbers Conference featured talks on Rust in kernel.

2021: Google and ISRG (Internet Security Research Group) started backing the initiative. Rust was proposed for initial support in the kernel.

2022: Linux 6.1 officially added Rust as an experimental language, introducing the rust/ directory. This enabled the writing of out-of-tree modules.

2023–2025: Ongoing development for broader support—drivers (e.g., for Android, graphics), kernel modules, subsystems. Enhanced documentation, community contribution tools, and tooling like bindgen stabilized for kernel use.

Kernel Versions with Rust Support

Kernel Version	Date Released	Rust Support Status	Notes
6.0	Oct 2022	No	Rust support not yet merged
6.1	Dec 2022	Initial experimental	CONFIG_RUST=y, Rust compiler required
6.2–6.5	2023	Continued expansion	Bug fixes, new abstractions, community testing
6.6+	Oct 2023+	Ongoing improvement	Modules in-tree, more APIs exposed

What You Can Build with Rust Today

In-Tree Rust Kernel Modules

You can write kernel modules entirely in Rust using the new `rust/` directory and the `KernelModule` trait. These modules are compiled using the standard kernel build system with Rust support.

Character Device Drivers

Implement basic `/dev/` nodes with `read`, `write`, and `ioctl` methods. Rust ensures safer pointer operations and more readable logic due to ownership and type safety.

Platform-Specific Drivers

Used in embedded systems to access memory-mapped registers or I/O. Rust's strict typing helps avoid misaligned or invalid memory accesses. No undefined behavior from dangling or null pointers.



What You Can Build with Rust Today

In-Memory File Systems (Toy Filesystems)

Experimental file systems (e.g., RAM-based) can be built safely using Rust abstractions. Rust's bounds-checking prevents buffer overflows and array indexing bugs.

Memory-Safe Wrappers for C APIs

Wrap existing kernel C interfaces in safe Rust APIs using unsafe internally but exposing only safe methods externally.

Helps prevent misuse and enforces safety contracts at compile time.

Architecture of a Rust-Based Kernel Module

Module Layout

- Entry Point: `module!` Macro
 - The `module!` macro in Rust is used to define the metadata and lifecycle of a kernel module.
 - This macro is similar to the `module_init()` and `module_exit()` functions in C.
- Lifecycle: `init()` and `exit()`
 - `init()`: The entry function when the module is loaded into the kernel. It initializes resources, registers handlers, and logs module loading.
 - `exit()`: The cleanup function when the module is unloaded. It frees resources, unregisters handlers, and performs necessary cleanups.

Architecture of a Rust-Based Kernel Module

Components

- Kernel Bindings
 - Kernel crate:
 - The kernel-specific crate `kernel` provides the necessary abstractions and bindings for writing kernel code in Rust. It ensures safe interaction with kernel memory, synchronization primitives, and other kernel components.
 - Macros:
 - Rust kernel modules leverage macros like `pr_info!()`, `pr_warn!()`, and `module!` to simplify code and avoid common pitfalls.
- Rust Abstractions
 - Rust's `Result<T>` and `Option<T>` types help handle errors and optional values in a clear, type-safe way. These abstractions replace error-prone `-1` or `NULL` checks.
- Low-Level Interactions
 - Unsafe code:
 - Rust allows for safe abstractions, but when interacting directly with hardware or system calls, unsafe code is needed.
 - The `unsafe` keyword is explicitly used to mark operations that are not checked by Rust's safety guarantees, such as accessing raw pointers, interacting with hardware registers, etc.

Architecture of a Rust-Based Kernel Module

Toolchain Requirements

- Rust Compiler (rustc)
 - Kernel code is compiled with rustc, the official Rust compiler, supporting no_std mode for embedded and OS-level development.
 - Specific Rust version constraints: Kernel requires at least Rust 1.60+ for stable features and minimal bugs.
- Linker (ld)
 - ld or compatible linker is needed for linking Rust code into the kernel. It is configured to work alongside the existing C toolchain.
- Cargo
 - Cargo, Rust's build system and package manager, manages dependencies and compilation units, ensuring Rust code integrates seamlessly into the existing kernel build system.
- Kernel Build System Integration
 - The kernel's Makefile and Kbuild infrastructure has been adapted to work with Rust.
 - rustc is invoked directly in the kernel's Makefile for compilation of Rust code alongside C code.
- Dependency Management
 - Rust code uses Cargo.toml to declare dependencies on libraries like kernel and libc (if needed).
 - Only no_std crates are compatible, as the kernel cannot rely on the full Rust standard library.
- Cross-Compilation
 - Kernel Rust modules often require cross-compilation (especially for embedded architectures) to ensure compatibility with the target architecture's toolchain.
 - Common cross-compilers like gcc-arm and clang integrate well with rustc for seamless cross-compiling.

Architecture of a Rust-Based Kernel Module

Enabled Architectures

- **x86_64:**
 - Full support for Intel/AMD 64-bit processors.
- **ARM:**
 - Rust modules are supported for ARM-based systems, including ARM64 (AArch64).
- **RISC-V:**
 - Emerging support for RISC-V processors, enabling Rust on open-source hardware.
- **PowerPC:**
 - Rust is being explored for PowerPC architectures, a key focus for embedded systems and older servers.
- **MIPS:**
 - Support being gradually integrated for MIPS processors in specialized environments.
- **Why architecture support matters:**
 - Each architecture requires specific toolchain configurations and low-level details for memory access, interrupt handling, and system calls.
 - As the Rust integration into the kernel evolves, these architectures are being extended to fully leverage Rust's safety benefits.

Rust Module Layout and Interaction

- Kernel Types Used
 - `Result<T>` for fallible operations
 - `Pin`, `Box`, `Arc`, `SpinLock`, etc. for safe memory and sync primitives
- Isolation of Unsafe
 - All unsafe operations must be explicitly marked and ideally isolated
 - Use unsafe only when interacting with C code or low-level kernel ops
- Runtime Behavior
 - Logs and panics use `pr_info!`, `pr_warn!`, `panic!()`
 - No standard output; logging is done through kernel `printk` interfaces

Kernel APIs Exposed to Rust

Available APIs

- Printk Logging
 - `pr_info()`, `pr_warn()`, `pr_err()`:
 - These macros allow logging to the kernel log buffer. They are safe, and type-safe in Rust, helping developers safely log messages of various severity levels.
- Memory Allocation
 - `kmalloc()` and `kfree()`:
 - Rust exposes `kmalloc()` and `kfree()` for memory allocation and deallocation, allowing dynamic memory management within kernel modules.
 - Rust ensures that memory is managed without the risk of dangling pointers or double frees using the ownership model.
 - `vmalloc()`:
 - Exposes functions for allocating larger contiguous memory blocks, typically used in drivers for complex data structures.
- File Systems (Early Support)
 - Support for filesystem operations in Rust is still in its early stages but allows developers to use abstractions for managing files, directories, and devices.
 - APIs are evolving, with initial bindings for the virtual file system (VFS) and filesystem registration.

Kernel APIs Exposed to Rust

Safety in API Design

- Rust Traits and Lifetimes
 - Traits:
 - Traits in Rust allow us to define functionality that can be implemented by types, ensuring that only safe operations are performed with kernel structures.
 - Lifetimes:
 - Rust's ownership model with lifetimes ensures that pointers do not outlive the data they point to, preventing use-after-free errors.
 - The use of lifetimes in kernel APIs ensures that memory management is explicit, making the system safe and more predictable.
- Concurrency Primitives
- Atomic Types

Safety and Concurrency in Kernel Rust Code

Concurrency Constructs

- **Mutex:**
 - Rust exposes kernel-level synchronization mechanisms such as Mutex to handle shared resources safely in multi-threaded environments.
- **RwLock:**
 - Similar to Mutex, but allowing multiple readers or one writer at a time, ensuring efficient concurrency handling.
- **SpinLock:**
 - A low-level lock for situations where threads cannot be blocked. This is commonly used in interrupt contexts where sleeping is not an option.
- **Atomic Types**
 - AtomicBool, AtomicU8, AtomicUsize, etc.
 - These types allow for safe, atomic operations on primitive types, ensuring thread-safe access to values without using locks.
 - **Usage:**
 - Atomic operations are critical for synchronization in high-performance environments, like counters or flags shared across threads.
- **Why Concurrency Primitives Matter**
 - Rust provides these abstractions to ensure thread safety while maintaining performance in kernel space.
 - Safe concurrency means fewer data races and more predictable behavior, especially in multi-core or real-time systems.

Limitations and Challenges in Using Rust

Stable Rust Toolchain and Clang and Integration into Existing Kernel Makefiles

Kernel Rust modules require a stable version of Rust (rustc), and it must work seamlessly with existing C-based tools.

Challenge: Ensuring that the Rust toolchain is updated and compatible with the latest kernel headers and features, as Rust is still in early stages of integration.

Ownership Model Complexity

Rust's ownership, borrowing, and lifetime rules are powerful but can be complex for developers new to the language.

Challenges: Passing data between C and Rust, managing unsafe blocks, and maintaining alignment and padding between C structs and Rust structures.

Lack of Rust-Specific Debugging Tools

While tools like GDB and LLDB support debugging for Rust code, kernel-level debugging for Rust is still in its early stages.

Challenge: Developers need to manually instrument their code with logging and print statements for debugging, which can be inefficient for complex systems.



Limitations and Challenges in Using Rust

Limited Driver and Filesystem Support

Rust's kernel support is still in its early phases, and the driver and filesystem (FS) support is limited. Essential subsystems like device drivers, network stacks, and storage drivers are not yet fully functional or supported. Challenge: As new features are added to the kernel, Rust developers must implement bindings and abstractions for kernel subsystems, a task that can be slow and inconsistent due to the lack of widespread adoption.

Code Review Processes

The Linux kernel's code review process is rigorous and conservative, particularly when it comes to introducing new languages or paradigms.

The community, led by Linus Torvalds, has shown skepticism about adding Rust to the kernel due to potential long-term maintenance, stability, and performance concerns.

Challenges: Rust's integration into the Linux kernel is still experimental, and many patches are rejected or delayed due to concerns over long-term stability.

Performance and Benchmarks: Rust vs C

General Performance Comparison

- Rust Performance
 - Rust is designed to offer zero-cost abstractions, meaning that abstractions in Rust should not introduce any performance overhead compared to writing equivalent code in C.
 - Rust's ownership system guarantees memory safety without the need for garbage collection, enabling direct memory access and control, similar to C.
 - The borrow checker enforces rules at compile-time, ensuring safety without runtime overhead.
- C Performance
 - C is known for its low-level control over hardware and system resources, making it the go-to language for performance-critical applications like the Linux kernel.
 - C provides direct memory access, pointer arithmetic, and manual memory management, which contribute to its high performance in systems programming.
 -
- Key Performance Factors:
 - Execution speed: Both Rust and C produce high-performance executables, often indistinguishable in terms of execution speed for most use cases.
 - Memory efficiency: Both languages allow for fine-grained control over memory allocation and deallocation. Rust, however, provides memory safety guarantees at compile time.

Performance and Benchmarks: Rust vs C

Benchmark Results: Rust vs C

- Common Kernel Benchmarks:
 - Kernel Boot Time: Compare the boot time of a kernel with modules written in Rust vs C. Rust's memory safety checks and abstractions may introduce small delays in startup.
 - Throughput & Latency: Compare networking throughput or I/O latency in Rust and C, particularly with high-frequency, real-time tasks.
- Real-World Benchmarks:
 - A benchmark comparing a Rust-based module for network packet processing versus a C-based equivalent might show similar performance in terms of latency and throughput, but Rust may have additional safeguards like null pointer dereference prevention and automatic bounds checking.
 - Memory Allocation: Rust may show better performance in terms of memory safety when working with large datasets or high concurrency tasks, as the borrow checker prevents data races at compile time, preventing issues common in C.
- Rust-Specific Advantages:
 - Concurrency and Threading: Rust's ownership model allows better handling of concurrency in multithreaded environments without sacrificing performance, thanks to its thread safety mechanisms.
 - Safety Overhead: While Rust includes additional safety checks at compile time, these do not generally incur a runtime penalty, leading to performance comparable to C for many applications.

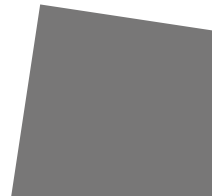



Community, Governance, and Contribution

Rust for Linux GitHub Repository

Official Repository: Rust for Linux

This repository hosts the ongoing work to integrate Rust into the Linux kernel. It serves as a reference point for the Rust kernel module work, including Rust bindings, kernel-safe abstractions, and tools for Rust-based development.



Community, Governance, and Contribution

Contribution Process

Patch Submission to LKML

LKML (Linux Kernel Mailing List): Patches to the Linux kernel must be submitted to LKML, which is the primary channel for submitting new code, discussing changes, and reviewing patches.

Rust Patch Workflow: Rust code contributions follow the same process as C, meaning Rust code must be properly formatted, tested, and reviewed before being accepted into the mainline kernel.

Important Considerations: Ensure compliance with coding standards, testing requirements, and appropriate documentation when submitting Rust patches.

RFC (Request for Comments) Discussions

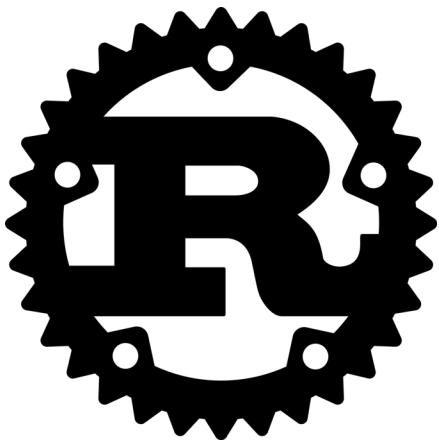
RFCs: Before submitting patches, developers are encouraged to start a discussion through the RFC process. This allows for community feedback, ensures ideas are well-received, and helps identify potential issues early.

Discussion Areas: These include design proposals, API interfaces, and integration with other kernel subsystems.

Maintainer Reviews

After submitting patches, the community and kernel maintainers conduct in-depth reviews to ensure that the code adheres to quality standards.

Key Points of Review: Maintainability, performance impact, code style, and safety concerns. Given that Rust is a new language for kernel development, the review process ensures that any code changes are robust and do not negatively impact kernel stability.



References

1. <https://doc.rust-lang.org/>
2. <https://github.com/Rust-for-Linux>
3. <https://www.kernel.org/doc/>
4. <https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html>
5. <https://www.kernel.org/doc/html/latest/rust/>
6. <https://www.linux.com/news/rust-linux-kernel/>
7. <https://doc.rust-lang.org/book/>
8. <https://lkml.org/>





**Thanks for your
Attention**

