Rust for Automotive

Presented at HAL4SDV
December 2, 2025

Pete LeVasseur
Lead @ Safety-Critical Rust Consortium
Co-lead @ Rust SIG, Eclipse SDV

Rust Fundamentals	3
Safety-Critical Rust	7
Addressing Topics	20
Close	37

01

Rust Fundamentals

Memory Safety

```
≝ no {
  use-after-free,
  out-of-bounds access
}
```

Informally, memory safety means there is no use-after-free, and that accesses cannot go out of bounds.

Out-of-bound accesses depends upon types and type safety.

Rust Programming Language

Rust is a high-performance systems programming language distinguished by its unique approach to achieving memory and thread safety and without a garbage collector.

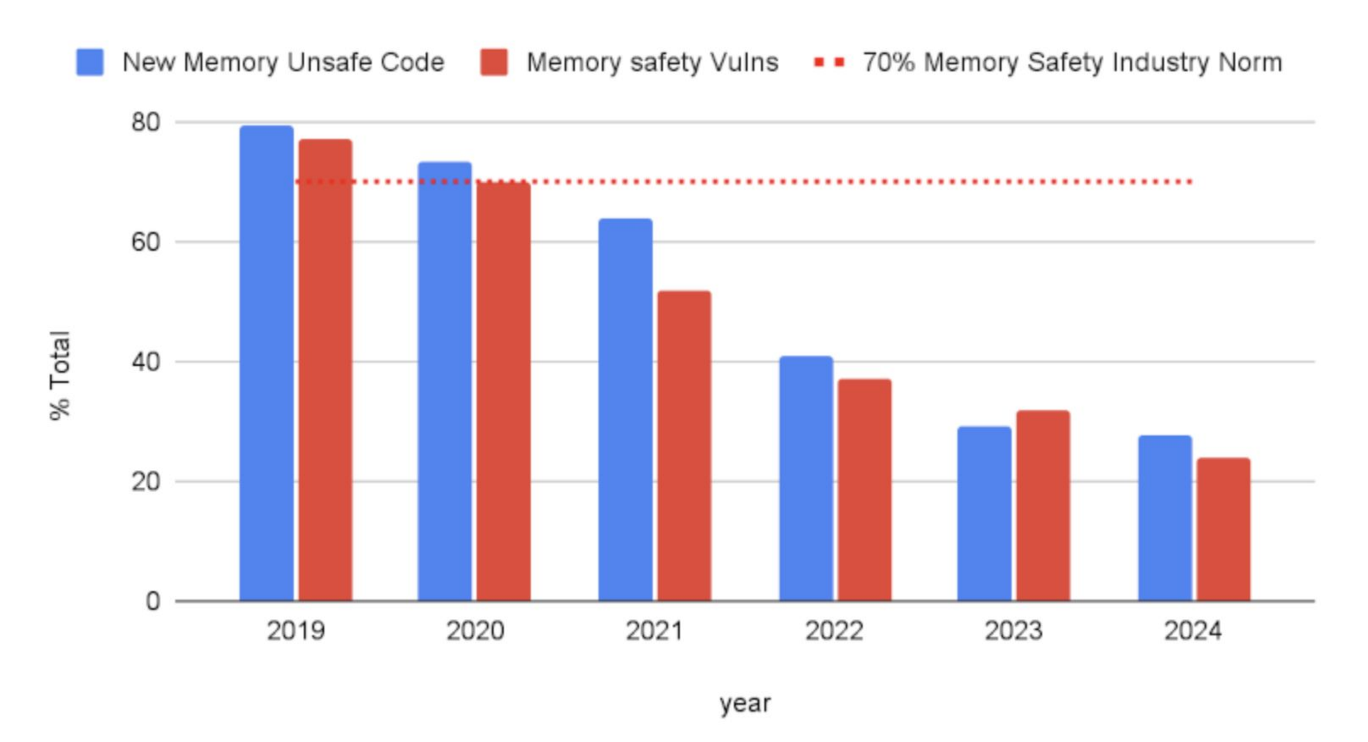
Each value in Rust has an owner.

- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

References can be borrowed to access data without taking ownership.

The Rust compiler's borrow checker prevents data races and ensure references remain valid, guaranteeing memory and thread safety.

Coexistence Possible: Android



https://queue.acm.org/detail.cfm?id=3773096

02

Safety-Critical Rust

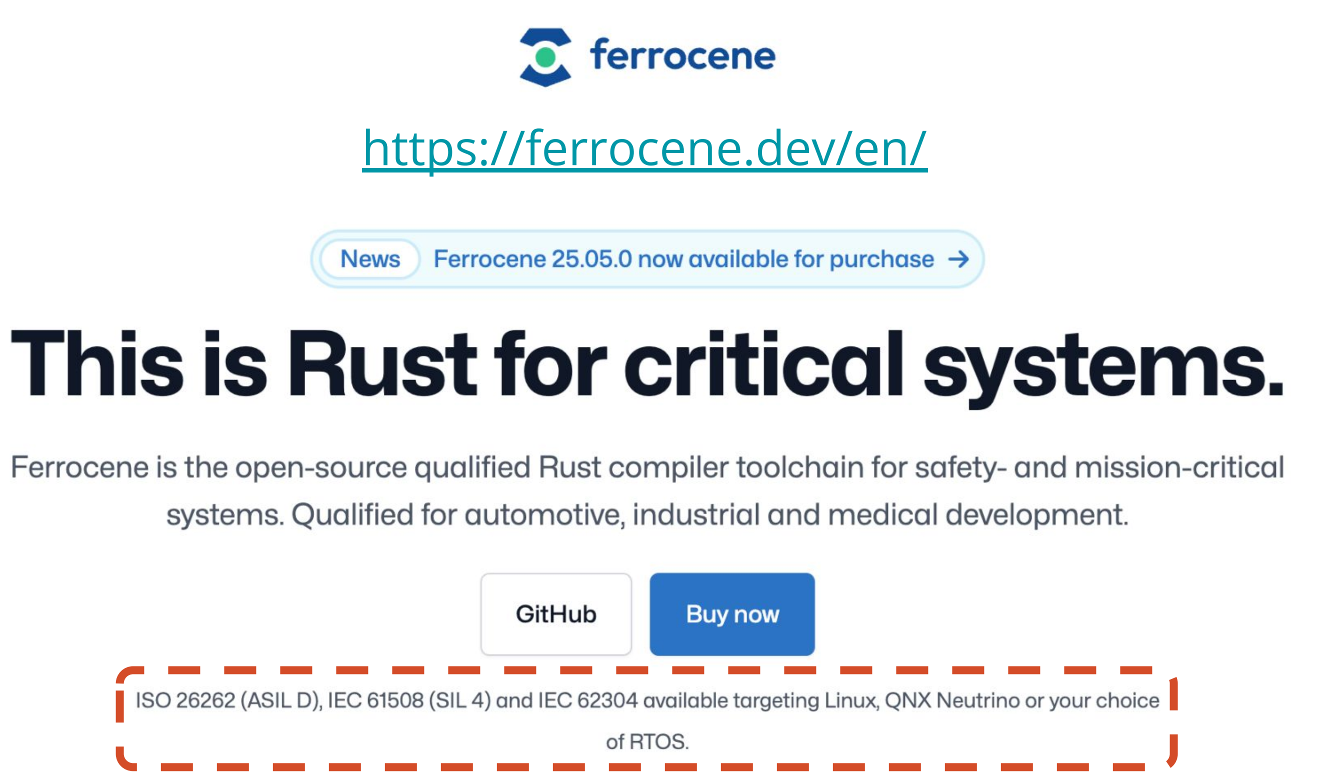
Automotive + Rust

https://tweedegolf.nl/en/blog/137/rust-is-rolling-off-the-volvo-assembly-line

2026 Toyota RAV4 in North America

https://filtra.io/rust/interviews/woven-by-toyota-nov-25

Open, Safety-Qualified Toolchain



https://ferrocene.dev/en/

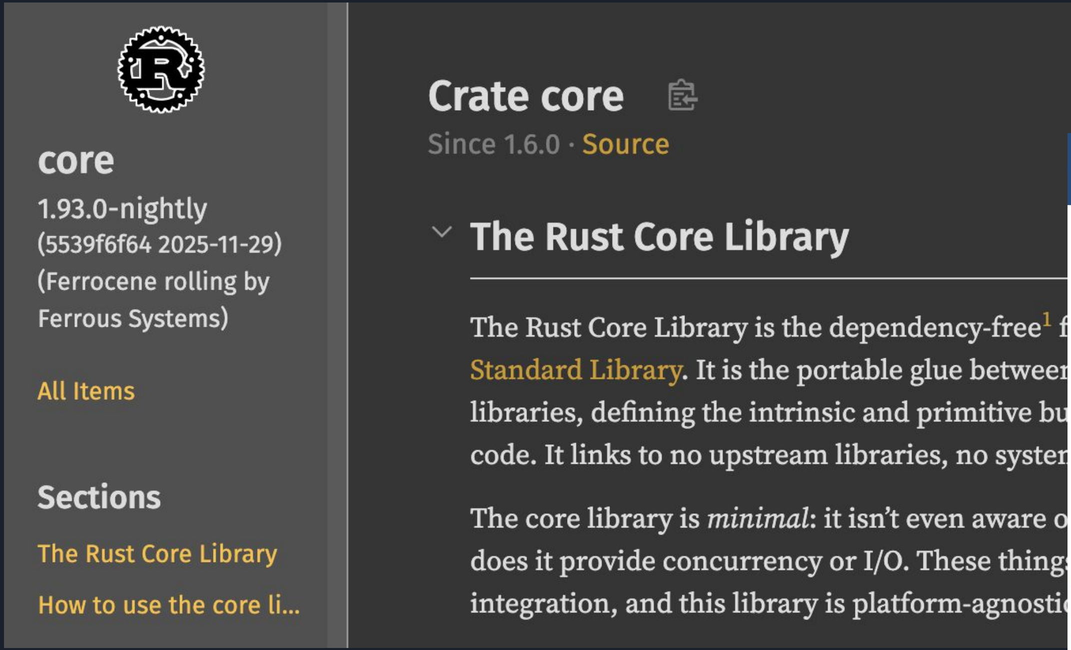
News Ferrocene 25.05.0 now available for purchase →

This is Rust for critical systems.

Ferrocene is the open-source qualified Rust compiler toolchain for safety- and mission-critical systems. Qualified for automotive, industrial and medical development.

ISO 26262 (ASIL D), IEC 61508 (SIL 4) and IEC 62304 available targeting Linux, QNX Neutrino or your choice of RTOS.

Subset of `libcore` Certified: ASIL B



core

1.93.0-nightly (5539f6f64 2025-11-29) (Ferrocene rolling by Ferrous Systems)

All Items

Sections

The Rust Core Library

How to use the core li...

Crate core

Since 1.6.0 · Source

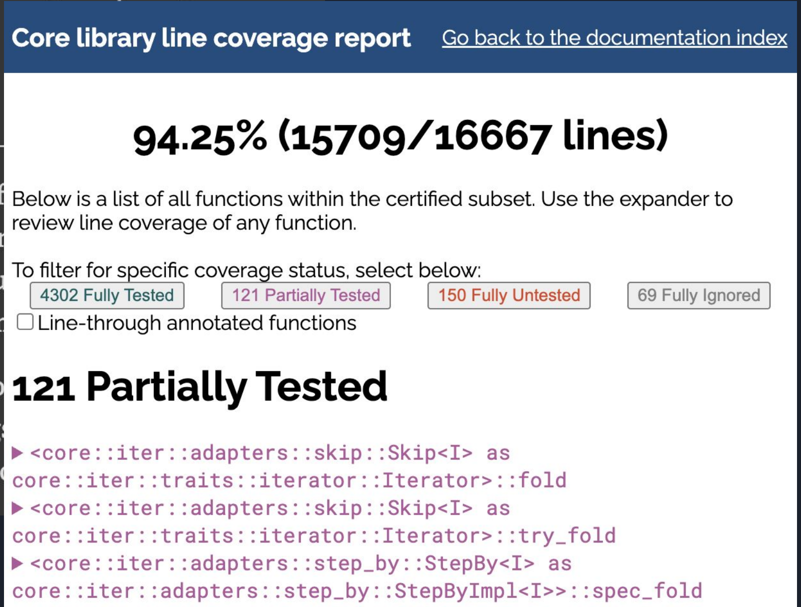
The Rust Core Library

The Rust Core Library is the dependency-free[1] f Standard Library. It is the portable glue betweer libraries, defining the intrinsic and primitive bu code. It links to no upstream libraries, no systen

The core library is *minimal*: it isn't even aware o does it provide concurrency or I/O. These things integration, and this library is platform-agnosti

https://public-docs.ferrocene.dev/main/core/index.html

https://public-docs.ferrocene.dev/main/coverage/index.html



Core library line coverage report Go back to the documentation index

94.25% (15709/16667 lines)

Below is a list of all functions within the certified subset. Use the expander to review line coverage of any function.

To filter for specific coverage status, select below:

4302 Fully Tested | 121 Partially Tested | 150 Fully Untested | 69 Fully Ignored

□ Line-through annotated functions

121 Partially Tested

```
▶ <core::iter::adapters::skip::Skip<I> as
core::iter::traits::iterator::Iterator>::fold
▶ <core::iter::adapters::skip::Skip<I> as
core::iter::traits::iterator::Iterator>::try_fold
▶ <core::iter::adapters::step_by::StepBy<I> as
core::iter::adapters::step_by::StepByImpl<I>>::spec_fold
```

Eclipse SDV WG: Rust SIG

- Knowledge-share for Rust best-practices in Automotive
- Lightweight consensus building around tools and methods
- Office hours to make sense of how to best use Rust in your software
    - Supported several companies in orienting themselves to the landscape of “how to do Rust”

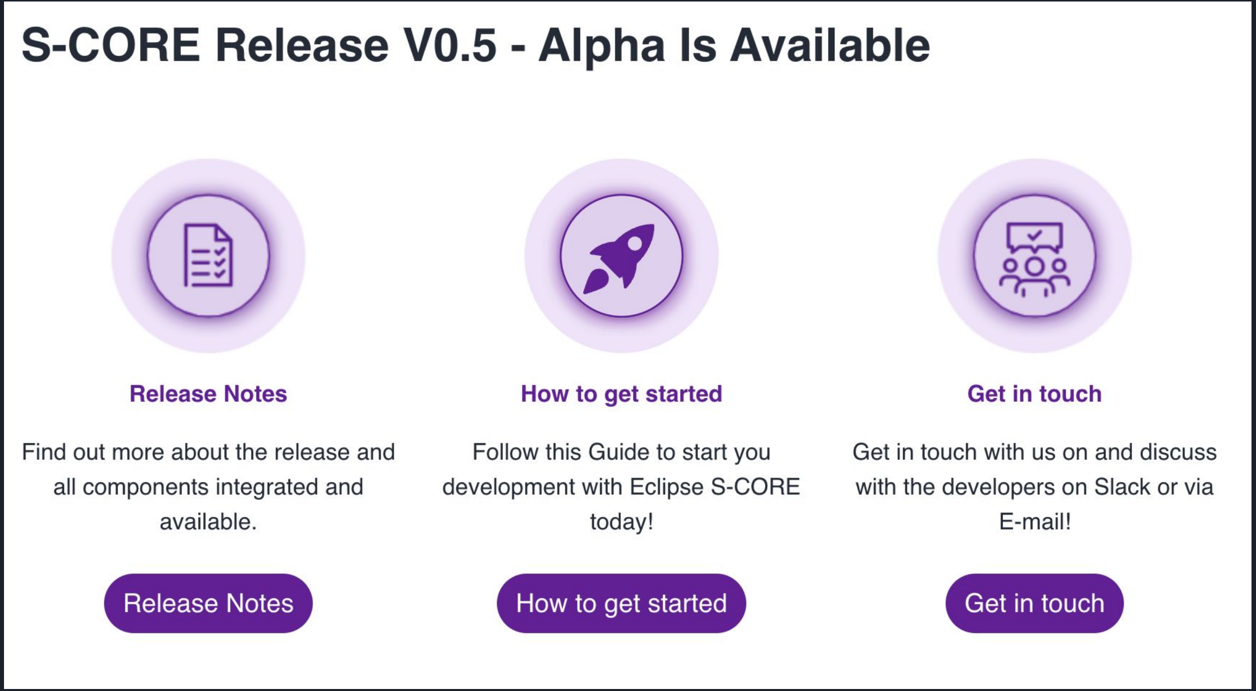
Eclipse SDV WG: Projects

SDV™
Eclipse Software Defined Vehicle

Ankaios	→	Container / service orchestration	https://github.com/eclipse-ankaios
uProtocol CONNECTING AUTOMOTIVE APPS AND SERVICES, EVERYWHERE	→	Service Mesh Abstraction	https://github.com/eclipse-uprotocol
Zenoh	→	Low-latency middleware	https://github.com/eclipse-zenoh
iceoryx2	→	Zero-copy shared-memory transport	https://github.com/eclipse-iceoryx

Roughly 37% of code in Eclipse SDV repos is Rust

Eclipse SDV WG: S-CORE



S-CORE Release V0.5 - Alpha Is Available

Release Notes

Find out more about the release and all components integrated and available.

How to get started

Follow this Guide to start you development with Eclipse S-CORE today!

Get in touch

Get in touch with us on and discuss with the developers on Slack or via E-mail!

Aims to enable
- both C++, Rust
- in initial release
- end of 2026
- up to ASIL B

Safety Critical Rust Consortium

+ Safety-Critical = Safety-Critical Rust Consortium

Subcommittees

- Coding Guidelines
- Liaison
- Tooling

SCRC Activities

Identify Requirements of Safety-Critical Standards and Satisfy Them

Driven *by* consortium members *for* consortium members and all in safety-critical.

Coding Guidelines Subcommittee
- Answering "how must Rust be written for Safety-Critical?"

Liaison Subcommittee
- Answering "how can we engage with language and safety communities and standards?"

Tooling Subcommittee
- Answering "what does Rust safety-critical development look like and what tools are missing?"

Safety-Critical
Rust Consortium

SCRC Membership & Turnout

Consortium

~180 members

Coding Guidelines Subcommittee

~50 members

Liaison Subcommittee

~15 members

Tooling Subcommittee

~30 members

2 x in-person meetups in 2025

- London, UK: ~30 members
- Utrecht, NL: ~30 members

Membership increased

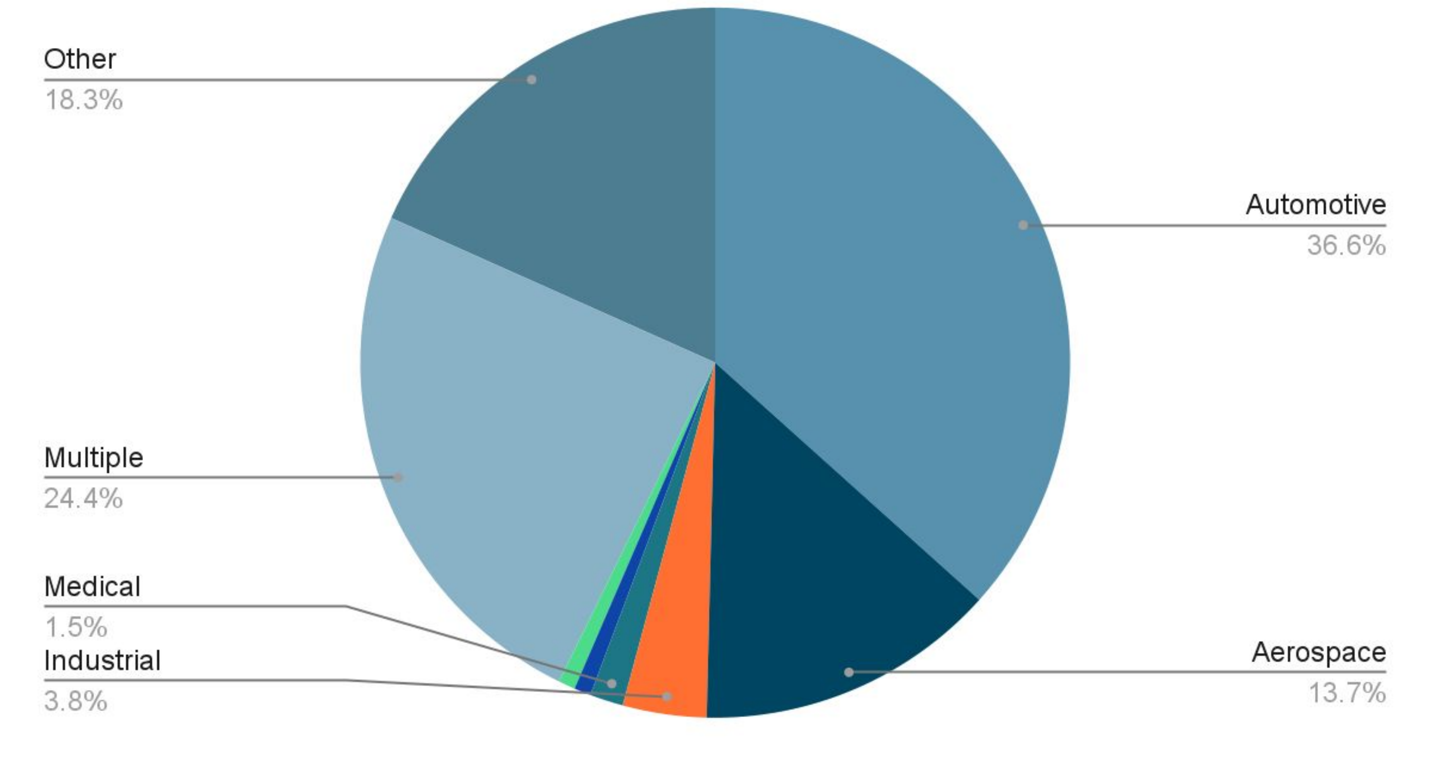
- by 29% since Utrecht

Global Virtual Meetup Oct 2025

~ 50 attendee turnout

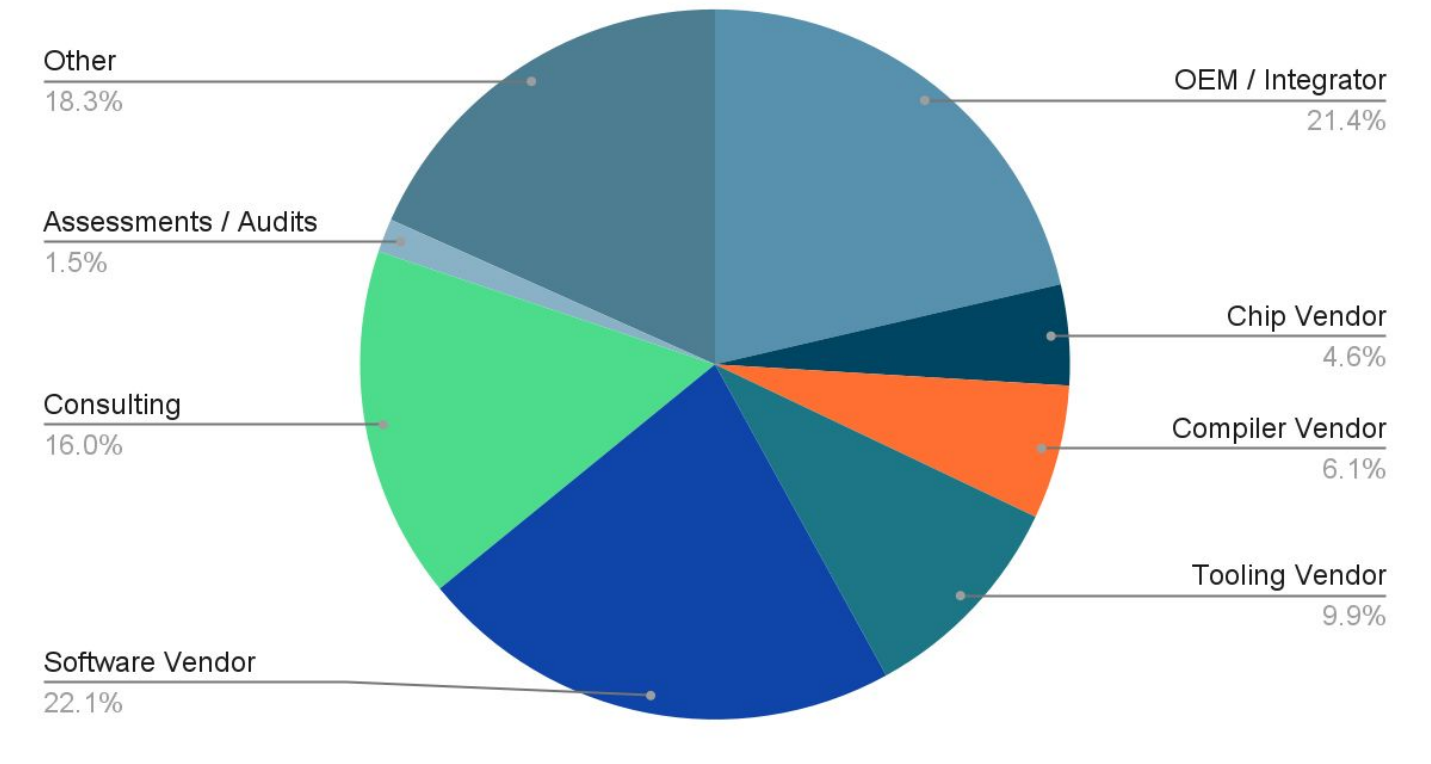
- Held over 2 days, 3 hours each

Makeup by Industry



Safety-Critical
Rust Consortium

Makeup by Domain



Safety-Critical Rust Consortium

Consortium Membership

Membership kept low-barrier to encourage both *observers* and *producers* to get involved

Application through GitHub:
- Consortium GitHub Template
- Subcommittee GitHub Template

Consortium membership gives mailing list access, invites to monthly SCRC all-hands, invitation to 2 x yearly in-person meetings.

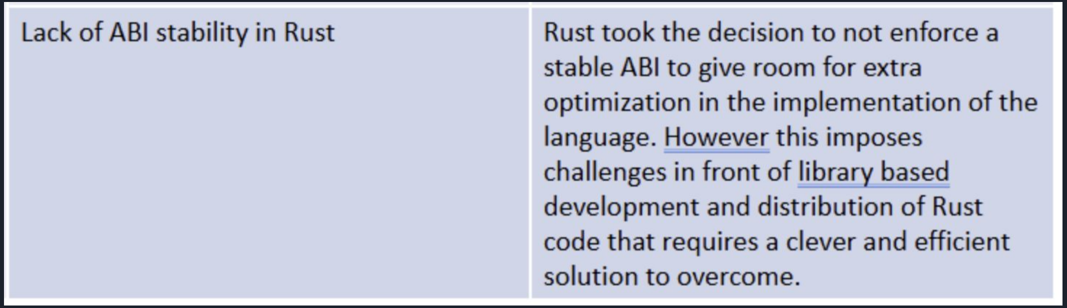
Subcommittee membership gets invites to recurring meetings.

Safety-Critical
Rust Consortium

03

Addressing Topics

Rust Compiler ABI Stability



Lack of ABI stability in Rust	Rust took the decision to not enforce a stable ABI to give room for extra optimization in the implementation of the language. However this imposes challenges in front of library based development and distribution of Rust code that requires a clever and efficient solution to overcome.

Interpreting this as that of wanting expanded ABI stability, beyond C ABI:

- Default ABI of `extern "Rust"` with `#[repr(Rust)]` layout is considered an implementation detail. Compiler's free to change between versions.
    - Stance of t-compiler and t-lang has been consistent
- Most stable ABI to use if you'd like `.so` or `.dll` is `extern “C”` with `#[repr(C)]`
- Some research-level maturity work on-going for `crABI`

ABI Stability: C ABI 1/2

Cargo.toml

```toml
[package]
name = "speed_control"
version = "0.1.0"
edition = "2021"

[lib]
# cdylib for a shared library,
# or "staticlib" for a .a
crate-type = ["cdylib"]

[build-dependencies]
cbindgen = "0.29"
```

src/lib.rs

```rust
#[repr(C)]
pub struct SpeedCommand {
    pub target_kph: f32,
    pub accel_limit: f32,
}

#[no_mangle]
pub extern "C" fn apply_speed_command(cmd: *const SpeedCommand)
{
    assert!(!cmd.is_null());
    let cmd = unsafe { &*cmd };
    // Real implementation is free to be idiomatic Rust.
    // The ABI boundary is just this function
    // and SpeedCommand layout.
    do_apply(cmd.target_kph, cmd.accel_limit);
}

fn do_apply(target_kph: f32, accel_limit: f32) {
    todo!("write logic here: {target_kph}, {accel_limit}")
}
```

ABI Stability: C ABI 2/2

src/lib.rs

```rust
#[repr(C)]
pub struct SpeedCommand {
    pub target_kph: f32,
    pub accel_limit: f32,
}

#[no_mangle]
pub extern "C" fn apply_speed_command(cmd: *const SpeedCommand)
{
    assert!(!cmd.is_null());
    let cmd = unsafe { &*cmd };
    // The ABI boundary is just this function
    // and SpeedCommand layout.
    do_apply(cmd.target_kph, cmd.accel_limit);
}

// Real implementation is free to be idiomatic Rust.
fn do_apply(target_kph: f32, accel_limit: f32) {
    todo!("write logic here: {target_kph}, {accel_limit}")
}
```

via build.rs

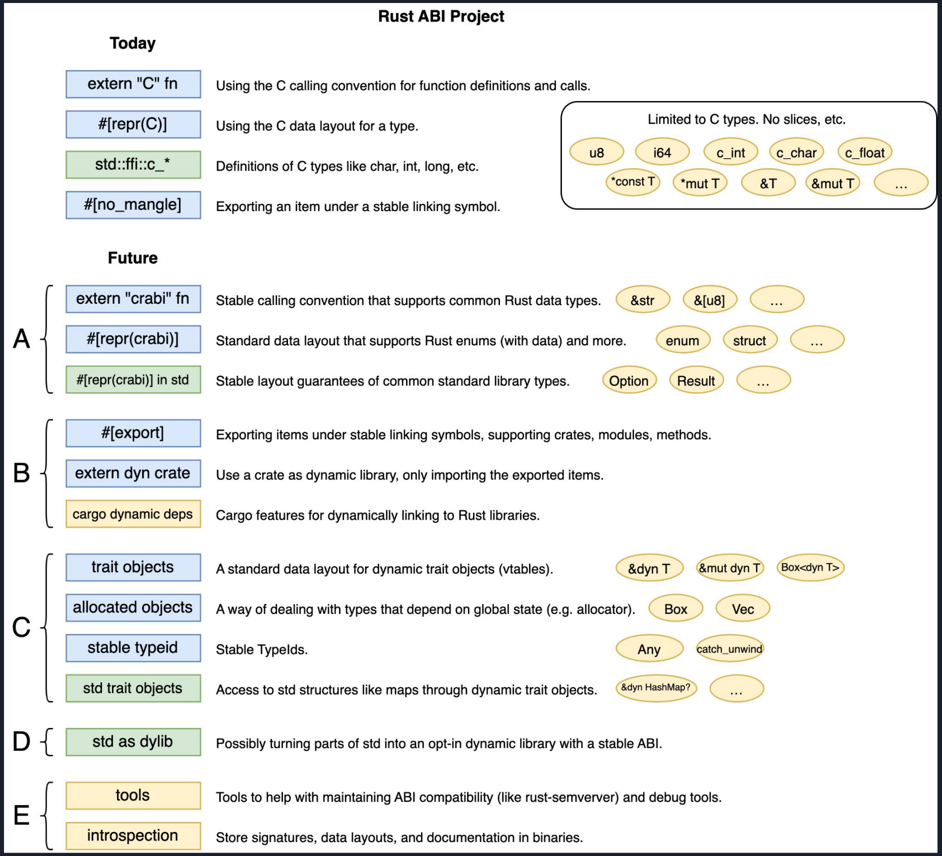
```c
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

typedef struct SpeedCommand {
  float target_kph;
  float accel_limit;
} SpeedCommand;

void apply_speed_command(const struct
SpeedCommand *cmd);
```

target/build/speed-control-xxx/out/speed-control.h

ABI Stability: crABI

Proposed in https://github.com/rust-lang/rust/pull/105586 on 2023-05-10

- Attempt to move above the lowest-common denominator of the C ABI
- Current approach is for each pair of languages, enable interop
  - e.g. C ⇔ Rust, C++ ⇔ Rust, Java ⇔ Rust, Python ⇔ Rust
  - Suffers from a need to have bespoke tooling; multiple languages suffer
- Progress seems unclear since middle of 2024 or so



Rust ABI Project

Today

extern "C" fn	Using the C calling convention for function definitions and calls.	
#[repr(C)]	Using the C data layout for a type.	
std::ffi::c_*	Definitions of C types like char, int, long, etc.	
#[no_mangle]	Exporting an item under a stable linking symbol.	

Limited to C types. No slices, etc.

u8 · i64 · c_int · c_char · c_float · *const T · *mut T · &T · &mut T · ...

Future

A	extern "crabi" fn	Stable calling convention that supports common Rust data types.	&str · &[u8] · ...
	#[repr(crabi)]	Standard data layout that supports Rust enums (with data) and more.	enum · struct · ...
	#[repr(crabi)] in std	Stable layout guarantees of common standard library types.	Option · Result · ...
B	#[export]	Exporting items under stable linking symbols, supporting crates, modules, methods.	
	extern dyn crate	Use a crate as dynamic library, only importing the exported items.	
	cargo dynamic deps	Cargo features for dynamically linking to Rust libraries.	
C	trait objects	A standard data layout for dynamic trait objects (vtables).	&dyn T · &mut dyn T · Box<dyn T>
	allocated objects	A way of dealing with types that depend on global state (e.g. allocator).	Box · Vec
	stable typeid	Stable TypeIds.	Any · catch_unwind
	std trait objects	Access to std structures like maps through dynamic trait objects.	&dyn HashMap? · ...
D	std as dylib	Possibly turning parts of std into an opt-in dynamic library with a stable ABI.	
E	tools	Tools to help with maintaining ABI compatibility (like rust-semverver) and debug tools.	
	introspection	Store signatures, data layouts, and documentation in binaries.	

ABI Stability: Safe Linking

Proposed in https://rust-lang.github.io/rust-project-goals/2025h1/safe-linking.html in January 2025

- The future we're working towards is one where (dynamically) linking separately compiled code (e.g. plugins, libraries, etc.) will feel like a first class Rust feature that is both safe and ergonomic.
- Depending on the outcomes of the research, this can provide input and design requirements for future (stable) ABIs, and potentially pave the way for safe cross-language linking.

- `#[export]` RFC (unmerged)
  - https://github.com/rust-lang/rfcs/pull/3435
  - Implementation PR:
    - https://github.com/rust-lang/rust/pull/134767
- Progress has stalled due to funding shortage

ABI Stability: Near-Term “Workarounds”

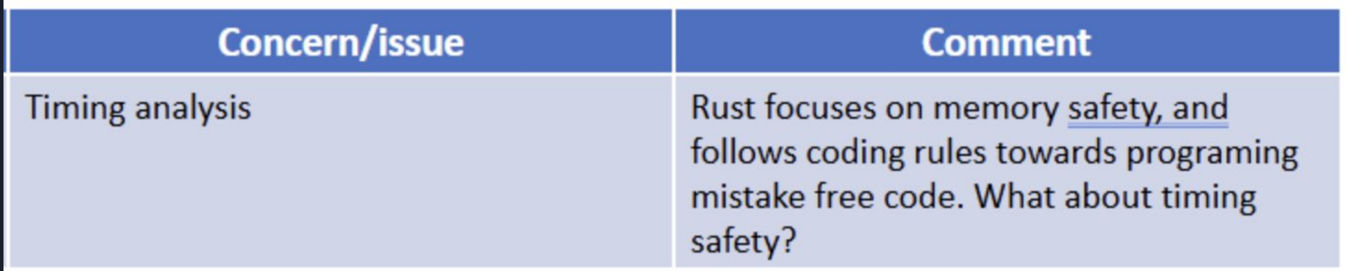
If you need something less powerful than a full stable ABI, but some stability:

- `stabby`
- `abi_stable`
- `safer_ffi`

These each have different trade-offs to consider, so generally

- it’s advisable to stick to the C ABI unless you’ve got a good reason
- if you’ll be doing something like a plugin system where you’ll have many other libraries written and loaded, choose a crate from above
  - For example: the Eclipse Zenoh router (`zenohd`) uses the `zenoh-plugin-trait` crate, backed by `stabby`, to present a stable programming interface for the shared library plugins loaded by the router

Timing Analysis: Safety Breakdown



Concern/issue	Comment
Timing analysis	Rust focuses on memory safety, and follows coding rules towards programing mistake free code. What about timing safety?

Interpreting this as that of the temporal, spatial breakdown of memory safety:

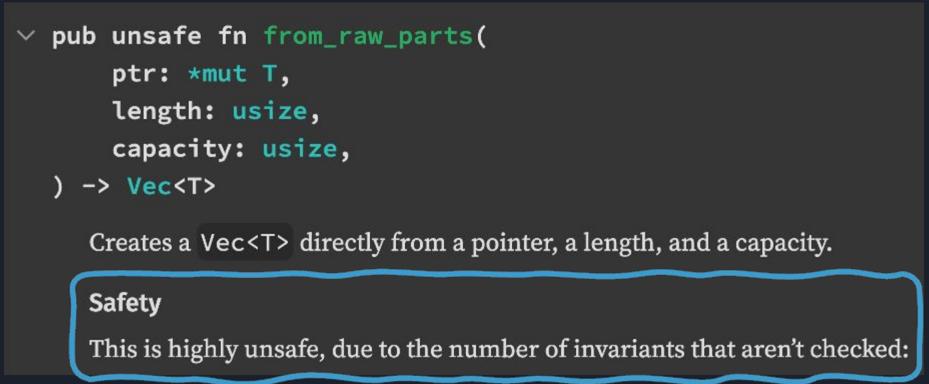
- Rust’s novel and central move is to make temporal safety a static guarantee in the type system
- Spatial safety is handled in more “conventional” ways (slices, bounds checks) once you have that.
- When reaching for the `unsafe` keyword the conversation gets a bit more complicated

unsafe: Additional Considerations

The unsafe keyword is a part of the Rust programming language and does have its uses

- calling unsafe APIs
- interacting with hardware
- performance optimizations

However, it brings new issues to consider.



```
pub unsafe fn from_raw_parts(
    ptr: *mut T,
    length: usize,
    capacity: usize,
) -> Vec<T>
```

Creates a `Vec<T>` directly from a pointer, a length, and a capacity.

Safety

This is highly unsafe, due to the number of invariants that aren't checked:

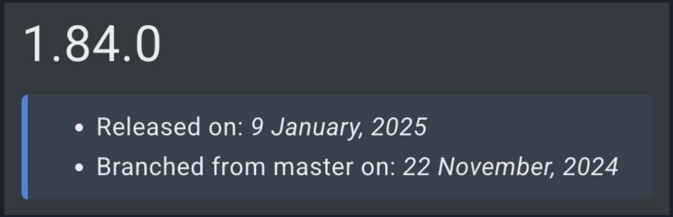
The invariants the compiler would normally check are now up to the engineer.

Pointer Provenance: A Primer

In Rust, pointers are not simply an “integer” or “address”.

A pointer value in Rust semantically contains the following information:

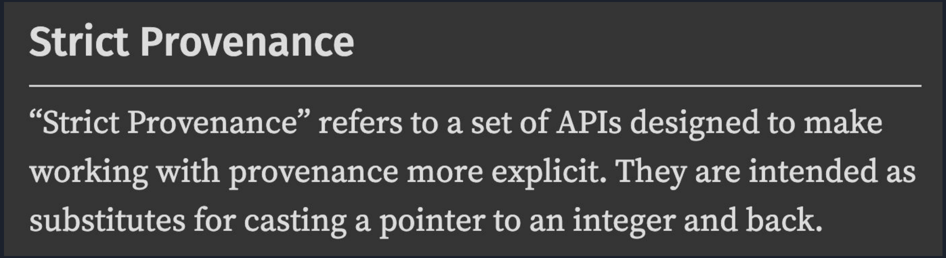
- The **address** it points to, which can be represented by a `usize`.
- The **provenance** it has, defining the memory it has permission to access. Provenance can be absent, in which case the pointer does not have permission to access any memory.



1.84.0

- Released on: *9 January, 2025*
- Branched from master on: *22 November, 2024*

https://releases.rs/docs/1.84.0/



Strict Provenance

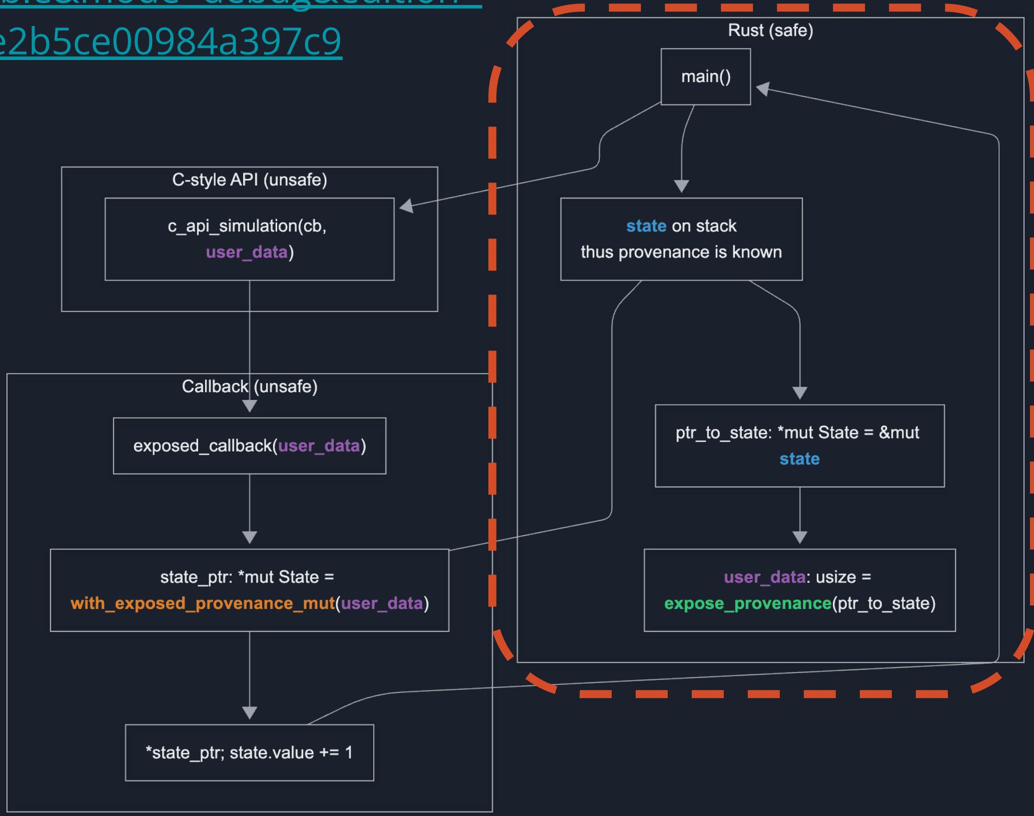
“Strict Provenance” refers to a set of APIs designed to make working with provenance more explicit. They are intended as substitutes for casting a pointer to an integer and back.

https://doc.rust-lang.org/std/ptr/index.html#strict-provenance

Pointer Provenance In Practice 1 / 3

https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=b438711e27aafbde2b5ce00984a397c9

```
fn main() {
  // Local state we want the C API to hand back
  // via `user_data`.
  let mut state = State { value: 41 };
  let ptr_to_state: *mut State = &mut state;

  // Expose the pointer's provenance and pack it
  // into `usize`.
  let user_data: usize = ptr_to_state
    .expose_provenance();

  // Simulate a C API passing that `usize` back
  // into our callback.
  unsafe {
    c_api_simulation(exposed_callback, user_data);
  }
  assert_eq!(state.value, 42);
}
```



Rust (safe)

main()

state on stack
thus provenance is known

ptr_to_state: *mut State = &mut state

user_data: usize = expose_provenance(ptr_to_state)

C-style API (unsafe)

c_api_simulation(cb, user_data)

Callback (unsafe)

exposed_callback(user_data)

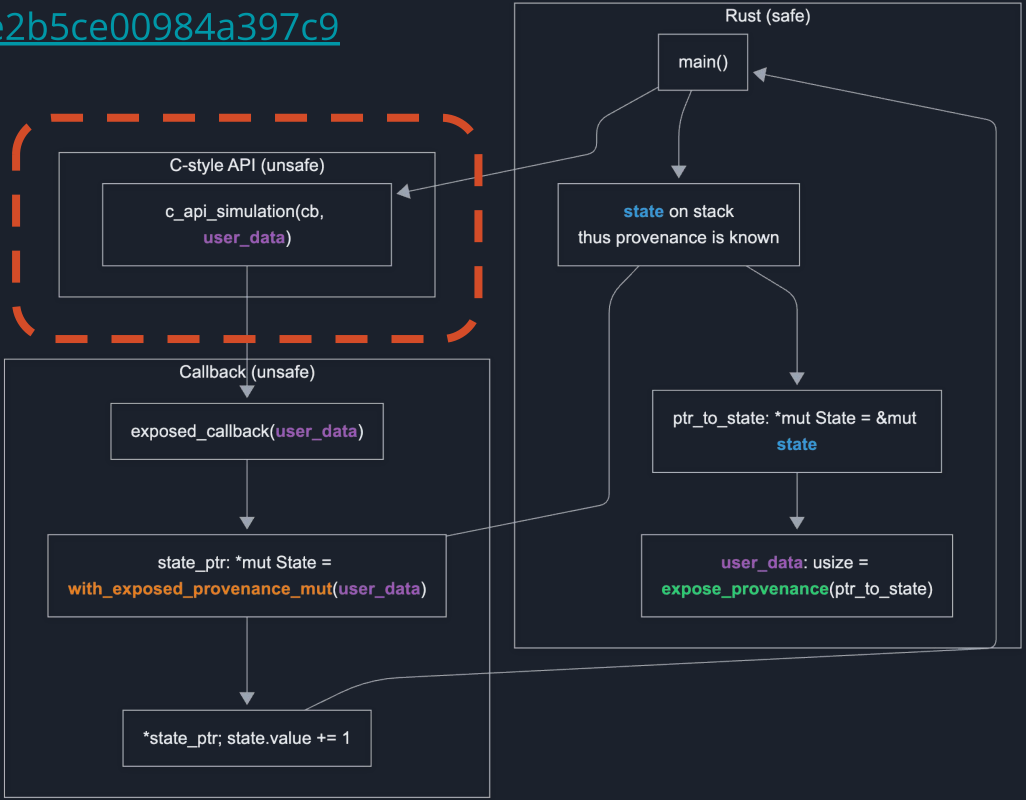
state_ptr: *mut State = with_exposed_provenance_mut(user_data)

*state_ptr; state.value += 1

Pointer Provenance In Practice 2 / 3

https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=b438711e27aafbde2b5ce00984a397c9

```
use std::ptr;
#[derive(Debug)]
struct State {
    value: i32,
}

// Simulate a C-style callback that gets a `usize`
// user data pointer.
type RawCallback =
  unsafe extern "C" fn(user_data: usize);

// A fake C API that just immediately calls
// the callback once.
unsafe fn c_api_simulation(
  cb: RawCallback,
  user_data: usize)
{
  unsafe {
      cb(user_data);
  }
}
```

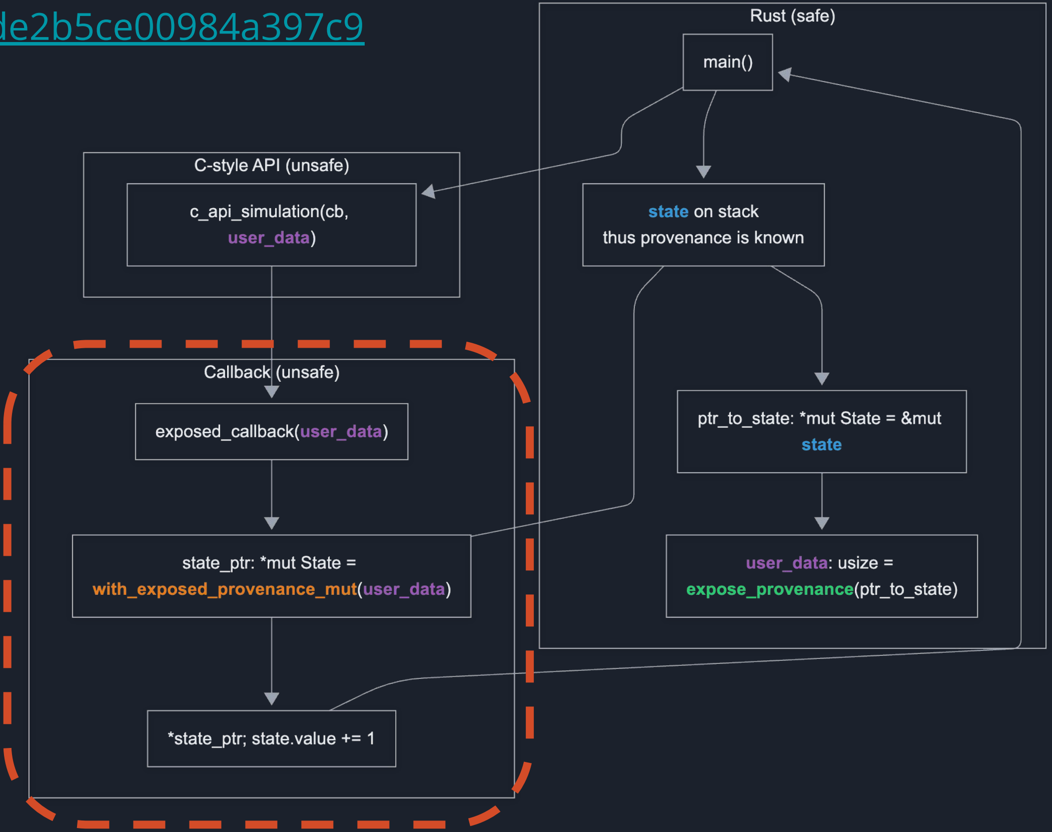


Rust (safe)
main()
state on stack
thus provenance is known
ptr_to_state: *mut State = &mut state
user_data: usize = expose_provenance(ptr_to_state)
C-style API (unsafe)
c_api_simulation(cb, user_data)
Callback (unsafe)
exposed_callback(user_data)
state_ptr: *mut State = with_exposed_provenance_mut(user_data)
*state_ptr; state.value += 1

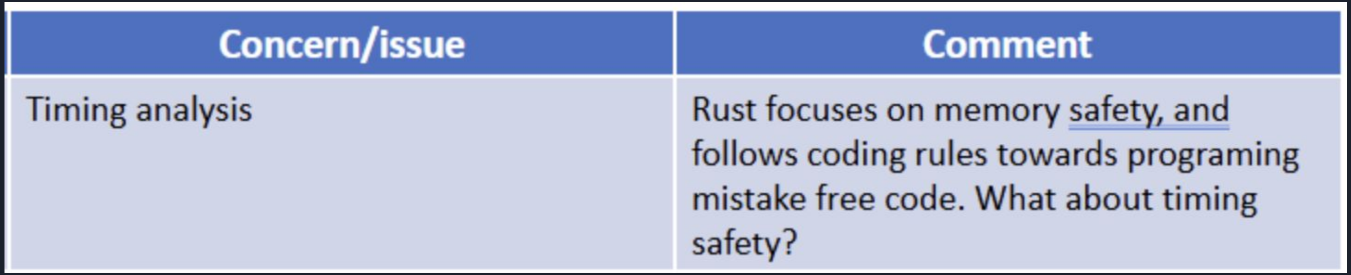
Pointer Provenance In Practice 3 / 3

https://play.rust-lang.org/?version=stable&mode=debug&edition=2024&gist=b438711e27aafbde2b5ce00984a397c9

```
// Exposed provenance: round-tripping a pointer
// through `usize`for FFI-style user data
unsafe extern "C" fn exposed_callback(
user_data: usize) {
  // Recreate the pointer to `State` from the
  // exposed address.
  let state_ptr: *mut State = ptr
    ::with_exposed_provenance_mut
    ::<State>(user_data);
  // SAFETY: In this example the `user_data` came
  // from a live `State` on the stack and the
  // callback is invoked before that `State` goes
  // out of scope.
  unsafe {
    assert_eq!((*state_ptr).value, 41);
    (*state_ptr).value += 1;
    assert_eq!((*state_ptr).value, 42);
  }
}
```


Rust (safe)
main()
state on stack
thus provenance is known
ptr_to_state: *mut State = &mut state
user_data: usize = expose_provenance(ptr_to_state)
C-style API (unsafe)
c_api_simulation(cb, user_data)
Callback (unsafe)
exposed_callback(user_data)
state_ptr: *mut State = with_exposed_provenance_mut(user_data)
*state_ptr; state.value += 1

Timing Analysis: WCET

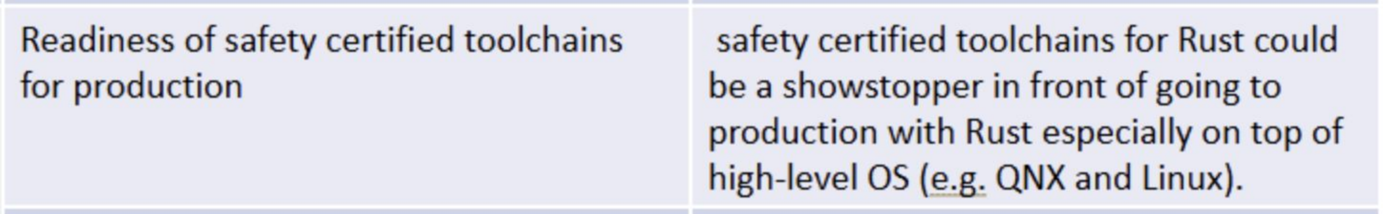


Concern/issue	Comment
Timing analysis	Rust focuses on memory safety, and follows coding rules towards programing mistake free code. What about timing safety?

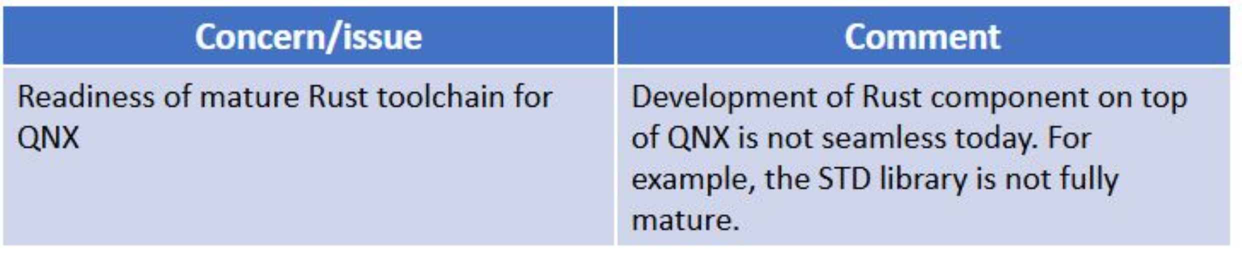
Interpreting this as that of the Worst-Case Execution Time analysis:

- Rapita RVS (RapiTime Zero + RapiCover Zero) + AdaCore GNAT Pro for Rust
  - Codebase scale / complexity supported is not clear
  - https://www.adacore.com/press/rapita-systems-showcases-adacores-gnat-pro-for-rust-at-hisc
- Many tools function on the compiled binary, so could be usable
- Open source efforts, but primarily research in nature or LLVM-focus
  - https://github.com/HEAPLab/es-prj-wcet-tool
  - https://gitlab.cs.uni-saarland.de/reineke/llvmta

Safety-Qualified Rust Toolchains



Readiness of safety certified toolchains for production	safety certified toolchains for Rust could be a showstopper in front of going to production with Rust especially on top of high-level OS (e.g. QNX and Linux).



Concern/issue	Comment
Readiness of mature Rust toolchain for QNX	Development of Rust component on top of QNX is not seamless today. For example, the STD library is not fully mature.

Interpreting this concern that Rust may not be ready for certain platforms:

- Upstream support for QNX 7.0, 7.1 is fairly solid
- Support for parts of `libstd` still in-flight for QNX 8.0
- It's possible to contribute, if the need is there

Safety-Qualified Rust Toolchains

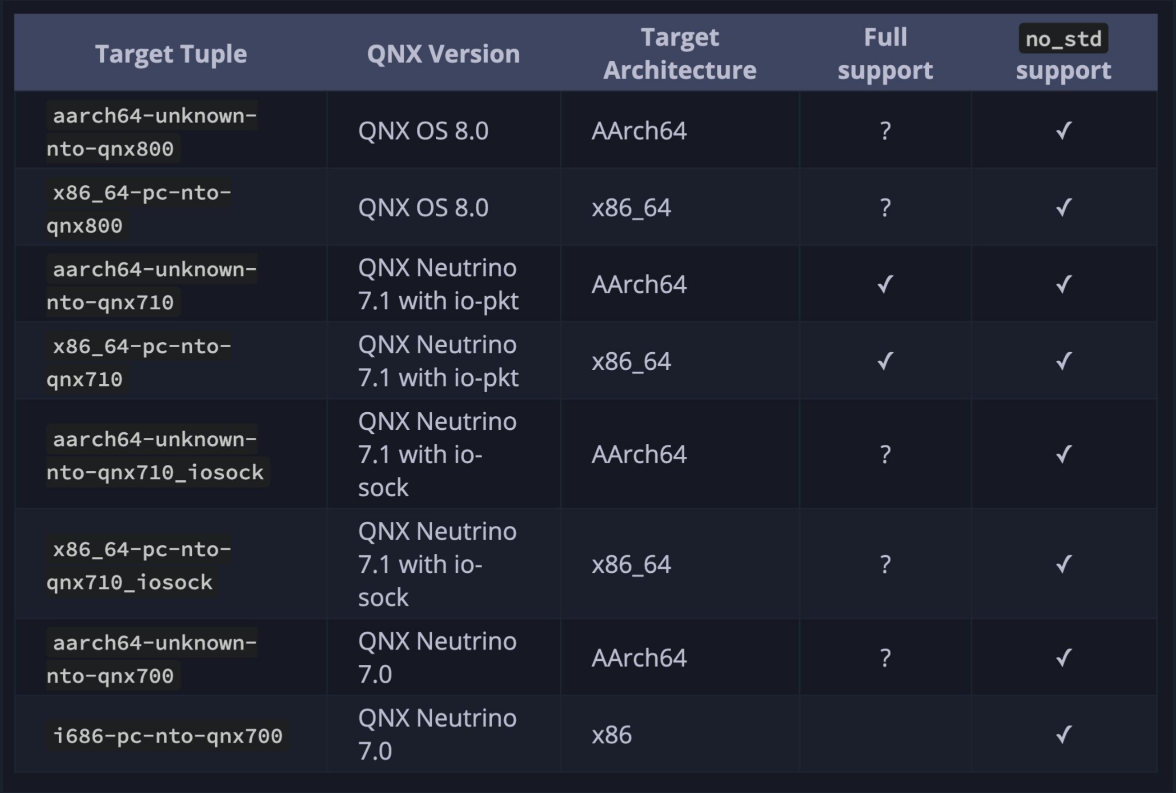
`nto-qnx`

Multiple options exist depending on the need

- Ferrocene
- GNAT Pro for Rust
- HighTec Rust Development Platform

Other platforms also available, e.g. `aarch64-unknown-linux-musl`

https://doc.rust-lang.org/beta/rustc/platform-support.html



Target Tuple	QNX Version	Target Architecture	Full support	`no_std` support
`aarch64-unknown-nto-qnx800`	QNX OS 8.0	AArch64	?	✓
`x86_64-pc-nto-qnx800`	QNX OS 8.0	x86_64	?	✓
`aarch64-unknown-nto-qnx710`	QNX Neutrino 7.1 with io-pkt	AArch64	✓	✓
`x86_64-pc-nto-qnx710`	QNX Neutrino 7.1 with io-pkt	x86_64	✓	✓
`aarch64-unknown-nto-qnx710_iosock`	QNX Neutrino 7.1 with io-sock	AArch64	?	✓
`x86_64-pc-nto-qnx710_iosock`	QNX Neutrino 7.1 with io-sock	x86_64	?	✓
`aarch64-unknown-nto-qnx700`	QNX Neutrino 7.0	AArch64	?	✓
`i686-pc-nto-qnx700`	QNX Neutrino 7.0	x86		✓

https://doc.rust-lang.org/beta/rustc/platform-support/nto-qnx.html

Safety-Qualified Rust Toolchains

- Which platforms are of highest importance to HAL4SDV?
- What level of safety-critical software development is planned for (ASIL)?
- Where in the vehicle is Rust being first considered for deployment?
  - e.g. ADAS or Body or …

04

Close

Thanks!

Pete LeVasseur
plevasseur@gmail.com
https://petelevasseur.com
https://www.linkedin.com/in/pete-levasseur

Safety-Critical Rust Consortium
https://arewesafetycriticalyet.org/

Safety-Critical
Rust Consortium

Eclipse SDV: Rust SIG
https://sdv.eclipse.org/special-interest-groups/rust/

SDV™
Eclipse Software Defined Vehicle

Rust Project: FLS
https://rust-lang.github.io/fls/