Rust for Safety and Security Critical Systems

Malte Munch

2025-10-28

This research is financially supported by the European Regional Development Fund and the Cybersäkerhetsnod Norr-project (no. 20366918) and the Swedish Innovation Agency 2025-00844, CRIT - Certification of RUST in crITical Systems

LULEÅ TEKNISKA UNIVERSITET

About me

- Malte Munch
- Cybersecurity PhD Student at Luleå University of Technology
- Supervisor: Per Lindgren (RTIC)
- `malte.munch@ltu.se`

Safety critical systems

- typical examples: cars, planes, trains
- behavior can be potentially dangerous / disastrous (brakes not working, steering a plane's nose down, ...)

Standards

Common practices to address safety issues are packaged into standards:

- ISO 26262 (Automotive)
- DO-178-C (Aviation)
- IEC 61508 (Industrial)
- IEC 62304 (Medical)
- ...

Safety integrity levels...

define a *safety goal*

- Automotive brake system: ASIL D (highest)
- Car headlights: ASIL B (second to lowest)

The higher the SIL, the more rigour is demanded to reduce risk

Risks

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

- → (un)schedulability
- → stack overflow
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

- → (un)schedulability
- → stack overflow
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

- → (un)schedulability - schedulability analysis, testing
- → stack overflow
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

- → (un)schedulability - schedulability analysis, testing
- → stack overflow - stack-depth analysis, testing
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

→ (un)schedulability	-	schedulability analysis, testing
→ stack overflow	-	stack-depth analysis, testing
→ undefined behaviour	-	guidelines, testing
→ memory (un)safety		
→ wrong implementation		

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

→ (un)schedulability	-	schedulability analysis, testing
→ stack overflow	-	stack-depth analysis, testing
→ undefined behaviour	-	guidelines, testing
→ memory (un)safety	-	guidelines, testing, specialized tooling (e.g. valgrind)
→ wrong implementation		

[1] for example C

Risks and (traditional) countermeasures

Implementation of safety critical systems using computers and (unsafe)[1] programming languages introduces additional risks, for example:

$\rightarrow$	(un)schedulability	-	schedulability analysis, testing
$\rightarrow$	stack overflow	-	stack-depth analysis, testing
$\rightarrow$	undefined behaviour	-	guidelines, testing
$\rightarrow$	memory (un)safety	-	guidelines, testing, specialized tooling (e.g. valgrind)
$\rightarrow$	wrong implementation	-	V-model, tight requirement tracking, tests, formal methods

[1] for example C

Traditional approach

- Use of potentially inadequate tools, like C
- Tests in this case means unit-tests
- counteract shortcomings with tooling, and pinning / freezing of state and lots of testing
- Testing means gathering statistical evidence that the implementation behaves well given a large sample-size (many test executions, long runtime)
- stemming from the "hazard-model" where "bad-things" happen with a certain, static likeliness.

$\Longrightarrow$ All in all this seems to work well enough[2][1]

Connectivity

Adding connectivity into our system [1] means having an entry channel for malicious actors.

→ The *hazard-model* needs to be accompanied by a *threat model.*

[1] Connectivity is required now, e-call for example

Connectivity

Adding connectivity into our system [1] means having an entry channel for malicious actors.

→ The *hazard-model* needs to be accompanied by a *threat model.*

Safety:

- more or less static hazard scenarios

Security:

- highly dynamic, expect the unexpected
- A malicious actor will always poke into the "blind spots"

[1] Connectivity is required now, e-call for example

Security

- dynamic and changing threats require dynamic, updateable systems
- This is reflected in Automotive Security Standards
  - ISO/SAE 21434
  - UNECE R155
- traditional approaches might be too slow to react adequately
- continous updates are in conflict with some safety development practices (testing, pin everything, proven in use)

$\Rightarrow$ New approaches are necessary to deliver fast and continuous updates

Risks

- → (un)schedulability
- → stack overflow
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

Risks, how to address them with Rust

- → (un)schedulability
- → stack overflow
- → undefined behaviour
- → memory (un)safety
- → wrong implementation

Risks, how to address them with Rust

→ (un)schedulability - RTIC / SRP + WCET Analysis, Symex, EASY

→ stack overflow

→ undefined behaviour

→ memory (un)safety

→ wrong implementation

Risks, how to address them with Rust

→	(un)schedulability	-	RTIC / SRP + WCET Analysis, Symex, EASY
→	stack overflow	-	cargo-call-stack, Symex
→	undefined behaviour		
→	memory (un)safety		
→	wrong implementation		

Risks, how to address them with Rust

→ (un)schedulability	-	RTIC / SRP + WCET Analysis, Symex, EASY
→ stack overflow	-	cargo-call-stack, Symex
→ undefined behaviour	-	Rust
→ memory (un)safety		
→ wrong implementation		

Risks, how to address them with Rust

→ (un)schedulability	- RTIC / SRP + WCET Analysis, Symex, EASY
→ stack overflow	- cargo-call-stack, Symex
→ undefined behaviour	- Rust
→ memory (un)safety	- Rust
→ wrong implementation	

Risks, how to address them with Rust

→ (un)schedulability	-	RTIC / SRP + WCET Analysis, Symex, EASY
→ stack overflow	-	cargo-call-stack, Symex
→ undefined behaviour	-	Rust
→ memory (un)safety	-	Rust
→ wrong implementation	-	as hard as before (tools to support formal methods exist)

Further Information

Further aspects mentioned in the Paper:

- Commercial Toolchains
- MISRA Guidelines
- Bare-metal vs. Hosted Systems
- detailed discussion of Rusts memory safety guarantees

References I

[1] Federal Aviation Administration. *Summary of the FAA's Review of the Boeing 737 MAX*. Federal Aviation Administration, Nov. 15, 2020, p. 99. URL: https://www.faa.gov/sites/faa.gov/files/2022-08/737_RTS_Summary.pdf (visited on 10/26/2025).

[2] Prof. Phil Koopman. "A Case Study of Toyota Unintended Acceleration and Software Safety". URL: https://users.ece.cmu.edu/~koopman/lectures/2014_toyota_ua.pdf (visited on 06/28/2025).

LULEÅ
UNIVERSITY
OF TECHNOLOGY