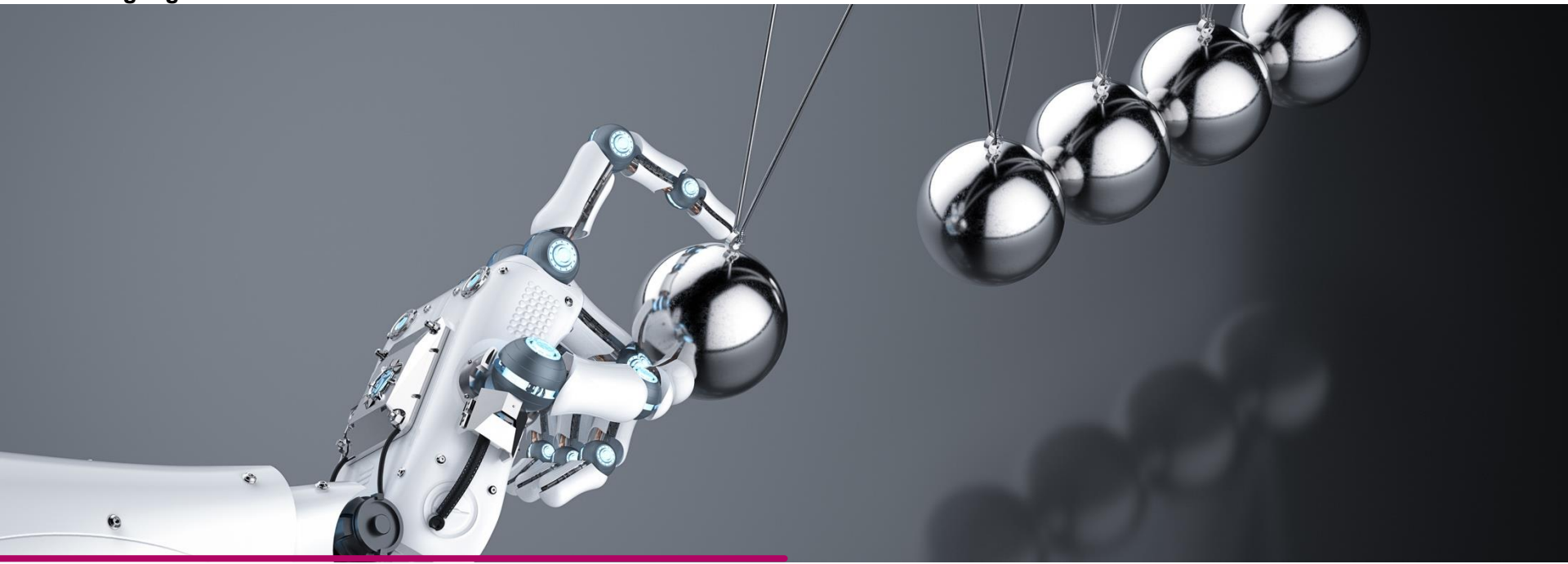
Von Regelprüfung zu Spracherzwingung: MISRA-Compliance in Rust

Florian Werner (VOLKSWAGEN Infotainment), Andreas Wübbeke (FH Südwestfalen)

TAV-Tagung 2026



Wir geben Impulse

VOLKSWAGEN INFOTAINMENT

Fachhochschule Südwestfalen
University of Applied Sciences

Agenda

Von der Regelprüfung zur Spracherzwingung: MISRA-Compliance in Rust

1. Motivation und Problemstellung
2. Forschungsfrage, Methodik und Klassifikation
3. Ergebnisse: Safe Rust, Unsafe Rust und Tool-Unterstützung
4. Diskussion: CI-Pipeline, Paradigmenwechsel und Sicherheitsprofil
5. Implikationen für Test & Verifikation
6. Auswirkungen auf KI-gestützte Softwareentwicklung
7. Fazit und Ausblick

Motivation

Memory-Safety-Bugs: Vom Code-Defekt zur globalen Krise

CrowdStrike - 19.07.2024

1 Out-of-Bounds-Access
in einem C-basierten Windows-Kerneltreiber

- 8,5 Mio. Geräte betroffen
- >5 Mrd. $ geschätzter Schaden
- Flughäfen, Krankenhäuser, Banken

Fehlerklasse: Array-Zugriff außerhalb der definierten Grenzen

Systemisches Muster

~70% aller CVEs
sind Memory-Safety-Fehler
(Microsoft, Google)

- NSA (11/2022): Guidance zu Memory-Safe Languages
- Linux Kernel (12/2025): Rust als permanente Kernsprache anerkannt

Was bedeutet das für Test, Analyse und Verifikation in der Automobilindustrie?

Quellen: [1] CrowdStrike PIR, 24.07.2024 [2] Microsoft Incident Report, 20.07.2024 [3] Parametrix Insurance, 2024

VOLKSWAGEN INFOTAINMENT
Fachhochschule Südwestfalen
University of Applied Sciences

MISRA C++:2023 – Überblick

175 Rules + 4 Directives für sicherheitskritische C++17-Entwicklung

Standard-Architektur und Scope

- **179 Guidelines:** Konsolidierung von MISRA C++:2008 und AUTOSAR C++14 für ISO C++17
- **Kategorisierung**: Mandatory / Required / Advisory – Decidable / Undecidable
- **Compliance-Rahmen**: MISRA Compliance:2020 (GEP, GCS, Deviation Records)

Adressierte Fehlerklassen

- Undefined Behaviour (Dangling Pointers, Data Races, Signed Overflow)
- Implementation-Defined Behaviour (Typ-Repräsentation, Alignment, Endianness)
- Fehleranfällige Muster (implizite Konversionen, Resource Leaks, Exception Safety)

Durchsetzungsmodell und Leitfrage

- **Klassisch**: externe **S**tatic-**A**pplication-**S**ecurity-**T**esting-Tools (Helix QAC, Polyspace, PC-lint) + manuelles Review
- **Grundproblem**: Regeln existieren als Kompensation für C++-Sprachschwächen
- **Leitfrage:** Welcher Anteil wird obsolet, wenn die Sprachschwächen strukturell eliminiert sind?

Rust als Alternative: Ein anderer Ansatz

Sicherheit als Spracheigenschaft, nicht als Nachkontrolle

Retrospektive Kontrolle

C/C++ with MISRA compliance

- Fehler nach dem Schreiben entdeckt
- MISRA kompensiert Sprachdefizite
- Hoher Aufwand: Static Analysis, Reviews
- ISO 26262 erzwingt Konformität

Prevention by Design

Rust compiler-enforced safety

- Ownership erzwingt Speichersicherheit
- Kein Null-Pointer, keine Data Races
- Compiler lehnt unsicheren Code ab
- Von Fehlersuche zu Fehlervermeidung

Was bedeutet das für Test, Analyse und Verifikation?

Forschungsfrage

Empirische Analyse von MISRA C++:2023 in Rust

Zentrale Frage

- Welche MISRA-Regeln werden durch Rust sprachseitig erzwungen – und welche nicht?

Rahmenbedingungen

- Masterarbeit: Kooperation VW Infotainment & FH Südwestfalen
- Kein Plädoyer für Sprachwechsel – sondern Analyse der Verschiebung von Verifikationspflichten
- Zielgruppe: Test- und Verifikationsingenieure in der Automobilindustrie

VOLKSWAGEN INFOTAINMENT
Fachhochschule Südwestfalen
University of Applied Sciences

Methodik: Regelauswahl und Klassifikation

Systematische Analyse von 179 MISRA-C++:2023-Guidelines

Auswahlkriterien

- Konzeptionelle Anwendbarkeit (Sicherheitseigenschaft jenseits von C++-Syntax)
- Relevanz für automobiltechnische Embedded-Software (Memory, Concurrency, Control Flow)

Ergebnis der Filterung (179 → 138 Regeln)

- 135 Regeln (75%) voll anwendbar auf Rust
- 3 Regeln (2%) teilweise anwendbar
- 41 Regeln (23%) nicht anwendbar – C++-spezifische Konstrukte ohne Rust-Äquivalent

Analysebasis: 138 (teil-)anwendbare Regeln für die empirische Evaluierung

Klassifikationsschema

Drei Kategorien für jede Regel

Spracherzwungen

Language-enforced

- Regel wird durch Compiler oder Typsystem von Rust inhärent durchgesetzt
- Verletzung führt zu Compile-Fehler – kein ausführbares Artefakt entsteht

Toolgestützt

Tool-enforced

- Regel ist anwendbar, aber nicht automatisch garantiert – externe Werkzeuge nötig

Nicht anwendbar

Non-applicable

- Regel bezieht sich auf C++-Konstrukte, die in Safe Rust nicht existieren

Experimentelles Setup: Vorgehen

Evaluierung von 138 MISRA-C:2023-Regeln gegen rustc und Clippy

Untersuchungsgegenstand

- **Basis**: MISRA C:2023 – 221 Guidelines (Mandatory / Required / Advisory)
- **Filterung**: 138 (teil-)anwendbare Regeln nach Ausschluss C-spezifischer Konstrukte
- **Decidability-Klassifikation**: statisch analysierbare vs. manuell prüfpflichtige Regeln

Experimentelles Design

- **Regel**: minimales Rust-Codebeispiel zur isolierten Regelverletzung (Negativtest)
- **Kompilierung**: stabiler rustc – Safe- und Unsafe-Varianten je Regel
- **Ergänzend**: Clippy-Lints (pedantic, restriction) als SAST-Erweiterung des Compiler-Checks

Klassifikationsschema der Compiler-Diagnosen

- **Enforced:** Kompilierung abgelehnt (hard error) – Verletzung strukturell unmöglich
- **Detectable:** Warnung / Lint (warn, deny) – toolgestützt identifizierbar
- **Residual:** Still akzeptiert – externes Review erforderlich

Experimentelles Setup: Eingesetzte Werkzeuge

Static-Analysis-Toolchain für Rust-MISRA-Compliance

Primäranalyse: Compiler-integrierte Prüfung

- **rustc**: Typ-, Ownership- und Borrow-Checker als Enforcement-Instanz
- **Clippy** (pedantic, restriction, nursery): >700 Lints für idiomatische/sicherheitsrelevante Muster

Sekundäranalyse: Ergänzende SAST-Werkzeuge

- **Semgrep**: regelbasierte AST-Mustererkennung für projektspezifische Compliance-Checks
- **Cargo-Geiger**: Quantifizierung der unsafe-Codedichte – Metrik für V&V-Aufwandsallokation

Automatisierung und Reproduzierbarkeit

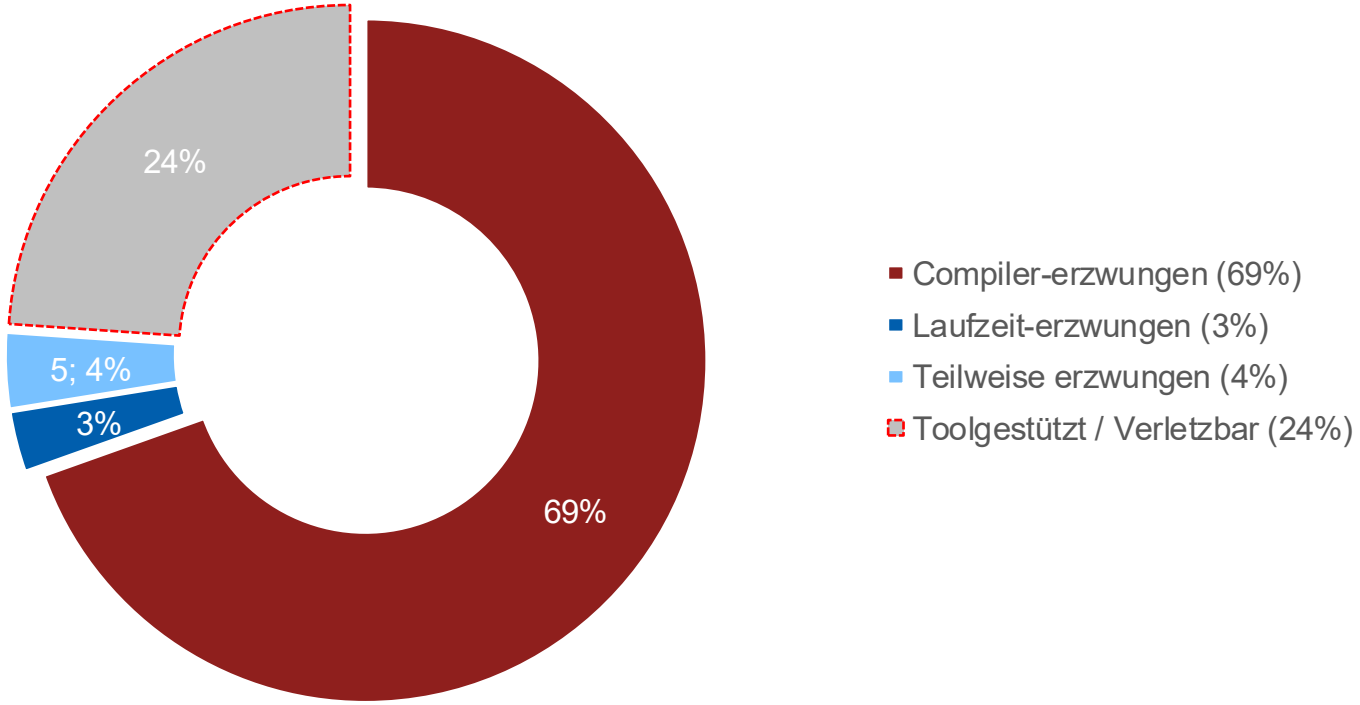
- **CI-Pipeline (GitHub Actions)**: automatisierte Compliance-Prüfung bei jedem Commit
- **Deterministische Umgebung**: fixierte Toolchain-Version, kontrollierte Compiler-Flags
- **Konform mit MISRA Compliance**:2020 § 5.3 (Guideline Enforcement Plan)

Ergebnisse

Safe Rust, Unsafe Rust, Tool-Unterstützung und CI-Integration

Ergebnis: Klassifikation unter Safe Rust (Visualisierung)

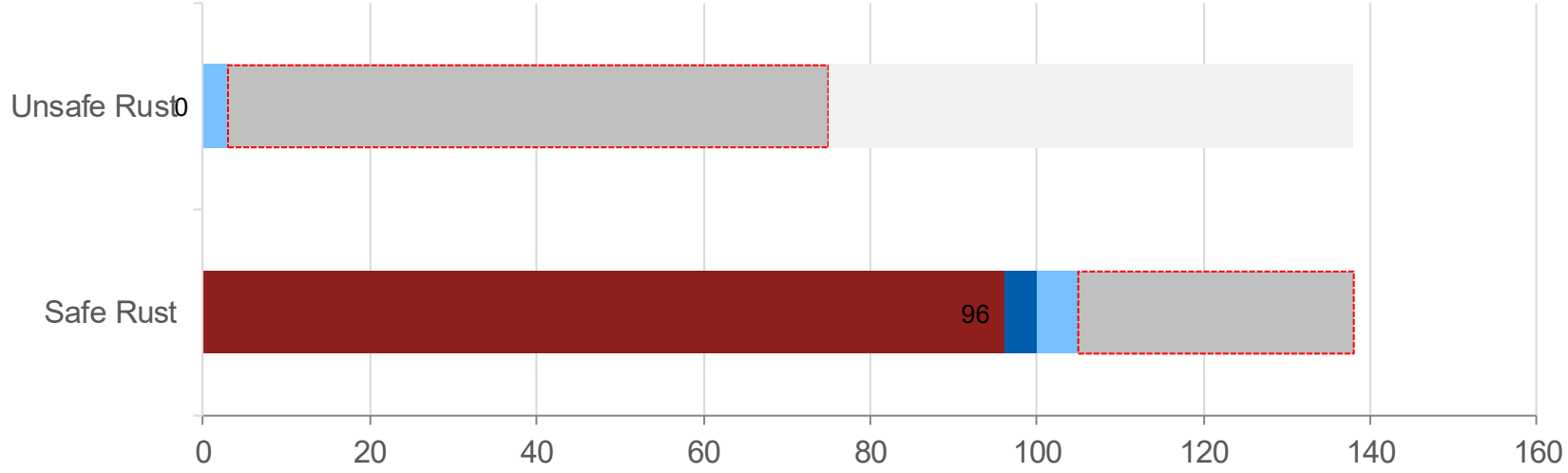
Verteilung der 138 anwendbaren MISRA-C++:2023-Regeln in Safe Rust



VOLKSWAGEN INFOTAINMENT
Fachhochschule Südwestfalen
University of Applied Sciences

Ergebnis: Safe vs. Unsafe Rust – Compliance-Vergleich

Verschiebung des Sicherheitsprofils durch unsafe-Kontexte



	Safe Rust	Unsafe Rust
Compiler-erzwungen	96	0
Laufzeit-erzwungen	4	0
Teilweise erzwungen	5	3
Toolgestützt / Verletzbar	33	72
Verletzung nicht möglich	0	63

VOLKSWAGEN INFOTAINMENT
Fachhochschule Südwestfalen
University of Applied Sciences

Ergebnis 3: Tool-Unterstützung in der Praxis

Fragmentierte Abdeckung – erhebliche Lücken bei Unsafe

Safe Rust (43 nicht vollständig erzwungene Regeln, davon 33 tool-prüfbar)

- Clippy und Semgrep decken 33 Regeln zuverlässig ab (geringe False-Positive-Rate)
- 10 Regeln teilweise erzwungen – partielle Compiler-Abdeckung, ergänzende Prüfung empfohlen

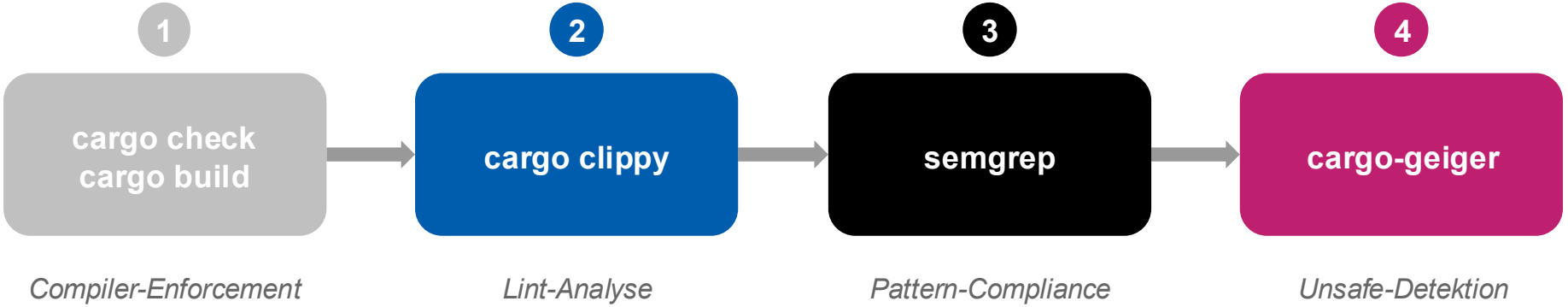
Unsafe Rust (72 Regeln zu prüfen)

- 34 Regeln: durch Semgrep (Custom-Regeldefinitionen) abgedeckt
- 3 Regeln: durch Clippy erkannt
- 1 Regel: durch einfaches grep-Matching erkennbar
- **34 Regeln (47%): KEIN verfügbares Tool** – semantische Komplexität übersteigt heutige Analysen

Fazit: Rust-Tooling-Ökosystem für Embedded noch nicht vollständig ausgereift

CI-Pipeline: Automatisierte MISRA-Compliance-Prüfung

Prototyp-Architektur der automatisierten Prüfkette (GitHub Actions)



Build-Gate: Pass / Fail bei Regelverletzung

VOLKSWAGEN INFOTAINMENT
Fachhochschule Südwestfalen
University of Applied Sciences

Diskussion & CI-Pipeline

CI-Integration, Paradigmenwechsel und Sicherheitsprofil

Diskussion: Paradigmenwechsel im Compliance-Modell

Von retrospektiver Regelprüfung zu sprachseitiger Enforcement

Traditionelles Modell (C/C++ + MISRA)

- Regelwerk kompensiert Sprachdefizite *post hoc* – Compliance durch externe SAST-Tools (Polyspace, Helix QAC)
- Retroaktiv, unvollständig, hohe False-Positive-Raten – V&V-Kosten skalieren überproportional

Neues Modell (Rust + Compiler-Enforcement)

- Sicherheitseigenschaften *a priori* durch Typsystem erzwungen – Use-after-free, Data Races, Buffer Overflows strukturell eliminiert
- V&V-Budget-Reallokation: Memory-Safety-Tests obsolet; Fokus auf Systemverhalten, Domänenlogik, Security

Differenzierung: Grenzen des Modells

- Unsafe-Code (MMIO, FFI, DMA), Logikfehler und domänenspezifische Invarianten bleiben außerhalb der Compiler-Reichweite
- Normative Lücke: ISO 26262 / IEC 61508 adressieren Rust nicht explizit – Ferrocene-Qualifikation notwendig, aber nicht hinreichend

Diskussion: Inverses Sicherheitsprofil

Strukturelle Asymmetrie zwischen Sprachgarantien und Risikoprofil

Beobachtung: Strukturelle Asymmetrie

- Compiler-Garantien maximal im Application Layer
- Höchstes Risiko an HAL/Driver-Ebene – empirisch: >50% unsafe in svd2rust, embedded-hal
- 20% aller Rust-Crates nutzen unsafe – in Embedded-Domänen signifikant höher

Implikation: Risikoadaptive V&V-Strategie

- V&V-Intensität muss invers zur Compiler-Abdeckung skalieren
- **Architekturprinzip**: „unsafe as thin layer“
- **Safe Rust**: funktionale Tests, SAST (Clippy)
- **Unsafe Rust**: formale Verifikation (Miri, Kani), manuelles Review, Boundary-Tests erforderlich
- ➢ **Offene Frage**: Reicht formale Verifikation für ASIL-D-Zulassung von unsafe-Boundaries?

Implikationen für Test & Verifikation (1/2)

Neue Kompetenzprofile und Test-Portfolio-Umstrukturierung

Test-Portfolio-Umstrukturierung

- Redundant werden: Null-Pointer-Tests, Use-after-free-Tests, Buffer-Overflow-Grenzwerte, Data-Race-Stresstests
- Ressourcen können auf funktionale Korrektheit und Domänen-Invarianten umgelenkt werden

Kompetenzverschiebung bei Verifikationsingenieuren

- Weniger relevant: Valgrind, AddressSanitizer, ThreadSanitizer für Safe-Code
- Wichtiger: Rust-Typsystem-Semantik, Ownership-Reasoning, Unsafe-Boundary-Auditing
- Neue Kernkompetenz: formales Reasoning über Unsafe-Block-Invarianten

Implikationen für Test & Verifikation (2/2)

Stratifiziertes Verifikationsmodell für Mixed Safe/Unsafe Code

Tier 1 – Application Layer (Safe Rust)
Primär: funktionale Tests und Integrationstests

Funktional getestet

Tier 2 – Abstraktionsgrenzen (Unsafe-Wrapping APIs)
Vertragsbasierte Tests, Invarianten-Dokumentation, Contract-Testing

Contract Testing

Tier 3 – Hardware-Interface (Unsafe Rust)
MISRA-äquivalente Prüfung: Manuelle Reviews, formale Verifikation, Grenzwert-Tests

Formale V&V

Implikationen für Test & Verifikation

Konsequenzen für Verifikation und Validierung

KI-Assistenz und Rust: Ein Synergieeffekt

Auswirkungen auf KI-gestützte Softwareentwicklung

Problemstellung: KI-generierter Code

- LLMs generieren zunehmend Produktionscode in C/C++ – mit inhärenten Memory-Safety-Fehlern [1]
- Hoher Verifikationsaufwand für KI-generierten C++-Code [2]

Rust als Sicherheitsnetz

- Compiler-Enforcement ermöglicht inhärente Korrektheitssicherung für KI-generierten Code
- Reduzierter Verifikationsaufwand gegenüber KI-C++-Code
- Strategische Rolle als *Safety Net* in KI-augmentierten Entwicklungsprozessen

Quellen: [1] Pearce et al., IEEE S&P 2022 [2] Perry et al., IEEE S&P 2023

Fazit und Ausblick

Rust verändert die Natur der MISRA-Compliance strukturell

Empirische Kernergebnisse

- **Compiler-erzwungene Compliance:** 69 % der MISRA-C:2023-Regeln durch Ownership, Borrowing & Typsystem statisch erzwungen → externe SAST wird für diese Klasse redundant
- **Residuale Regelklassen (31 %):** Codierungsstil, zyklomatische Komplexität, Laufzeitarithmetik
- **Unsafe-Grenzschicht:** ~20 % aller Crates nutzen unsafe, in Embedded signifikant höher für MMIO, FFI, DMA
- **Compliance-Modellwechsel:** Von retroaktiver Regelprüfung zu sprachseitiger Enforcement auf Compiler-Ebene

Implikationen & Ausblick

- **Stratifizierte V&V:** Compiler-Garantien (Safe Rust) + Clippy/SAST (Stilregeln) + formale Verifikation Miri/Kani (Unsafe)
- **Qualifizierte Toolchain:** Ferrocene, TÜV-SÜD-zertifiziert: ISO 26262 ASIL D, IEC 61508 SIL 4,
- **V&V-Budget-Reallokation:** Weniger manuelle Reviews für Speicherdefekte; Fokus auf Systemverhalten, Domänenlogik & Security
- **Normative Perspektive:** MISRA Add. 6 ist Grundlage künftiger MISRA-Rust-Richtlinien; Community entwickelt eigene Safety-Standards

Paradigmenwechsel: Von Tool-getriebener Regelprüfung zu sprachseitig erzwungener Sicherheit

Ausblick: Offene Forschungsfragen

Identifizierte Forschungsdesiderate und methodische Anschlussmöglichkeiten

a) Quantitative Defektdichtenanalyse Rust vs. C/C++

Empirische Erhebung von Defektraten (Defekte/kLOC) in Serienprojekten unter Kontrolle für Confounders (Teamreife, Codebasis-Alter, Testabdeckung). Kosten MISRA-C-Tooling vs. sprachseitige Enforcement.

b) Formale Verifikation der Unsafe-Grenzschicht

Verifikationsstrategien für Rust–C FFI-Boundary (MMIO, DMA, HAL). Evaluation der Reichweite und Skalierbarkeit von Miri, Kani und Creusot auf industrielle Codebases.

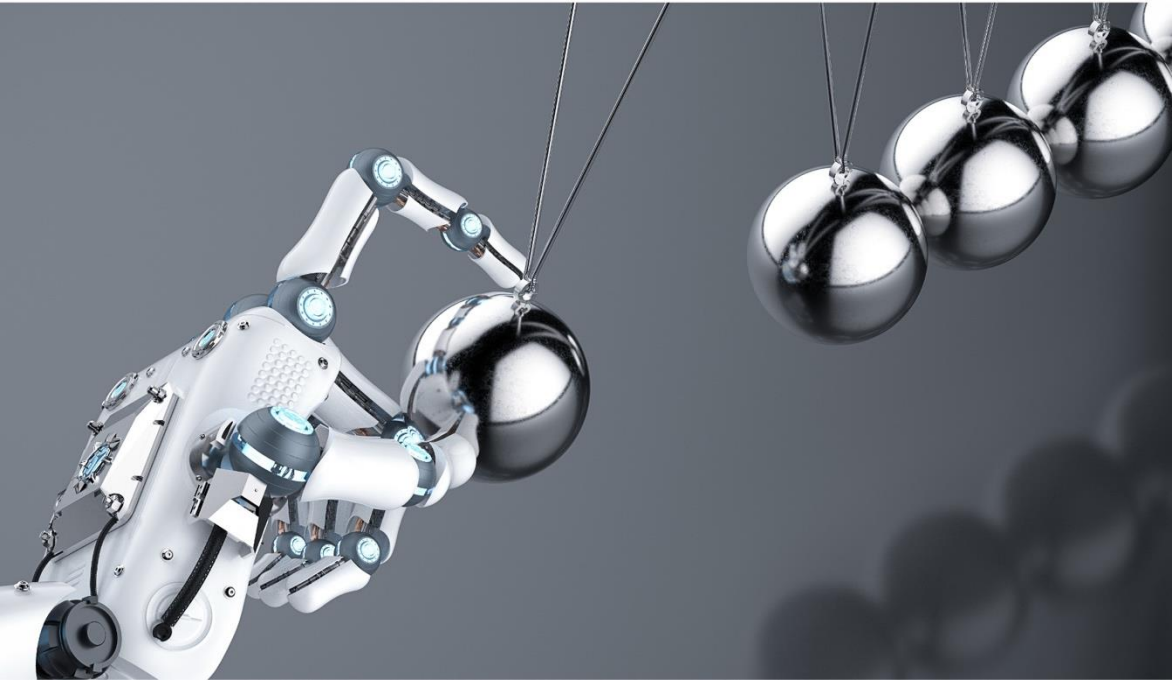
c) Hybride C++/Rust-Architekturen in der automobilen Serienentwicklung

Koexistenzmodelle für C++ und Rust auf Automotive-Plattformen: Interoperabilitätsmuster (FFI-Wrapper, C-ABI, cxx-Bridge), Safety-Island-Partitionierung (sicherheitskritische Module in Rust, Applikationslogik in C++) sowie Auswirkungen auf Build-Systeme und Toolqualifikation. Inkrementelle Migrationspfade bei bestehender MISRA-C++-Compliance.

d) Industrielle Transferstudie: Rust in ASIL-B–D-Projekten

Longitudinale Fallstudie zur C→Rust-Migration (≥3 OEM/Tier-1). Erhebung von Schulungsaufwand, Tooling-Kosten, Qualitätsmetriken und ISO-26262-Prozessintegration.

TAV-Tagung 2026



Florian Werner
florian.werner@vwif.com

Andreas Wübbeke
wuebbeke.andreas@fh-swf.de

Wir geben Impulse

VOLKSWAGEN INFOTAINMENT

Fachhochschule Südwestfalen
University of Applied Sciences