RIOT im RISC-V User-Mode

Lars Pfau

INET-Seminar, Grundprojekt

Inhalt

- Motivation
  - IoT Security
  - Trusted Execution Environments
- Analyse
  - RISC-V Privileged Architecture
- Problemstellung
- Implementierung
  - RISC-V Secure Firmware
- Evaluation
- Ausblick

Motivation: IoT Security

- Anbindung einer hohen Anzahl von Sensoren und Geräten an das Internet
- Eigenschaften von IoT Geräten
  - Stückkostenoptimiert
  - Mikrocontroller ohne Speicherisolation
  - Software hat Zugriff auf gesamten physischen Adressraum
- Sicherheitsprobleme
  - Remote Attacker nutzen Sicherheitslücken aus (z.B. Buffer Overflows)
  - Private Schlüssel auslesen
  - Malware ausführen, installieren
- Auswirkungen auf das Internet
  - Botnetze, DDoS-Angriffe

Trusted Execution Environments

- TEE = Manipulationssichere Ausführungsumgebung
- Software wird auf zwei Ausführungsumgebungen aufgeteilt:
  - „Normale“ Umgebung
    - **RTOS + Anwendung**
  - „Sichere“ Umgebung
    - **Secure Monitor / Firmware**
- Isolation begrenzt die Auswirkungen von Angriffen
  - Normale Umgebung hat keinen Zugriff auf die Ressourcen der sicheren Umgebung
  - Ein Angriff hat keine Auswirkung auf die sichere Umgebung

Use Cases / Features von TEEs

- **Software Updates + Secure Boot**
  Nur signierte Software wird ausgeführt
- **Keystores**
  TEE speichert Private Schlüssel und Session Keys
- **Remote Attestation**
  Drittsysteme können Integrität der IoT-Nodes verifizieren
- **Secure Storage**
- **Enclaves**

RISC-V ISA

- Neue, offene Befehlssatzarchitektur
  - Modular aufgebaut und erweiterbar durch Extensions, Profiles
  - 32-bit Mikrocontroller – 64-bit Datacenter CPUs
- RISC-V Privileged Architecture
  - Privilege Modes: Machine (M), Hypervisor (HS), Supervisor (S), User (U)
  - Physical Memory Protection (PMP)
  - „Secure Embedded Systems“: M+U Mode + PMP
- Trusted Execution Environments auf RISC-V Secure Embedded Systems möglich
  - User-mode: ■ **RTOS + Anwendung**
  - Machine-Mode: ■ **Secure Monitor**

RIOT + RISC-V

- RIOT unterstützt zurzeit drei 32-bit RISC-V Mikrocontroller:
  - Espressif ESP32-C3
  - GigaDevice GD32VF103
  - SiFive FE310

- SiFive FE310 ist ein RISC-V Secure Embedded System
  - Unterstützt Machine mode + User mode
  - 8 PMP Regions

- RIOT ist entwickelt für RISC-V Simple Embedded Systems
  - RIOT + Anwendung laufen im Machine mode
  - PMP für Data Execution Prevention

Ziel:

Trusted Execution Environments auf RISC-V Mikrocontrollern mit RIOT implementieren

Fragestellungen:

1. Wie kann RIOT im RISC-V User-mode ausgeführt werden?
2. Was sind die technischen Hindernisse für ein User-mode RTOS?
3. Welche Auswirkungen hat der User-mode auf ein RTOS?
   (z.B.: Performance, Speicherverbrauch, etc.)

Grundlagen: RISC-V Privileged Architecture

- Traps = Interrupts + Exceptions
- Interrupts
  - External, Timer, Software Interrupts
  - Konfiguration durch externe Interrupt Controller (Memory mapped)
- Exceptions
  - Zugriffsfehler
  - Illegal instructions
  - Systemaufrufe (ECALL, EBREAK)
- Konfiguration und Behandlung von Traps erfolgt mit „Control and Status Registers“ (CSRs)
  - mstatus: Interrupts global aktivieren, deaktivieren
  - mie, mip: Interrupts maskieren, behandeln
  - mtvec: Adresse des Trap handlers
  - mcause: Ursache der Trap
  - mepc: Program counter zum Zeitpunkt der Unterbrechung
- Trap handler werden in Machine-mode ausgeführt

Grundlagen: RISC-V Traps

(1) Eine Trap wird ausgelöst

1. Ausführungskontext wird gespeichert (Interrupt Enable, Privilege Mode, PC)
   - mstatus.MPIE = mstatus.MIE
   - mstatus.MPP = Current Privilege Mode
   - mepc = pc
2. Interrupts werden deaktiviert, Mode switch
   - mstatus.MIE = 0
   - Privilege Mode = 3 (Machine)
3. Ausführung des Trap handlers
   - pc = mtvec

(2) Trap handler wird ausgeführt

- Liest mcause
- …

(3) Trap handler endet mit „mret“ Instruction

- mstatus, pc werden wiederhergestellt:
  1. mstatus.MIE = mstatus.MPIE
  2. Privilege Mode = mstatus.MPP
  3. pc = mepc

RIOT cpu/riscv_common

- Zugriff auf Machine-mode CSRs und Instructions in RIOT:
  - **irq_arch.c**
    - mcause (ro)
    - mepc (rw)
    - mret

    (für trap_entry, handle_trap)
  - **coretimer.c**
    - mie (wo)
  - **plic.c**
    - mhartid (ro)
  - **irq_arch.h**
    - mstatus (rw)

    (für irq_disable, irq_restore, …)
  - **thread_arch.h**

    (in thread_yield_higher)

Problemstellung

Ziel:

RIOT soll im User-mode ausgeführt werden.

Probleme:

- RIOT hat im User-mode keinen Zugriff auf Machine-mode CSRs
  - M-mode CSRs für Behandlung von Interrupts
    - mcause, mepc, mie, mhartid
  - M-mode CSRs für aktivieren, deaktivieren von Interrupts
    - mstatus, mie

- Trap handler werden im Machine-mode ausgeführt

Lösung

- Entwicklung einer minimalen RISC-V M-mode Secure Firmware
  - Startet RIOT im User-mode
  - Leitet Interrupts an User-mode weiter
  - Systemaufrufe für Zugriff auf Machine-mode CSRs

- Neues RIOT CPU target: „cpu/riscv_common_user“
  - Fork von cpu/riscv_common
  - Ausführbar im User-mode
  - Trap handler interagiert mit der Secure Firmware
  - Zugriffe auf Machine-mode CSRs entfernt oder ersetzt

System Reset

- PC springt zu vordefinierter Adresse in Machine-mode

1. Secure Firmware startet
   - RAM initialisieren
     - Kopiert .data, .sdata von LMA → VMA region
     - Clear .bss, .sbss
   - PMP initialisieren
     - Code, Daten der Secure Firmware werden geschützt
   - Trap Vector mtvec initialisieren
   - Mode switch in User-mode
   - Jump in RIOT‘s _start
2. RIOT startet
   - libc, ctors initialisieren (__libc_init_array)
   - RIOT initialisieren (cpu_init, board_init, kernel_init)

Table 1: PMP Region configuration

PMP Region	PMP Type	Privilege Mode	Description	M-Mode Sections	R	W	X
0	TOR	M+U			1	1	1
1	TOR	M	M-Mode ROM	.init .text .rodata .srodata	0	0	0
2	TOR	M+U			1	1	1
3	TOR	M	M-Mode RAM (LMA)	.data .sdata	0	0	0
4	TOR	M+U			1	1	1
5	TOR	M	M-Mode RAM (VMA)	.data .sdata .bss .sbss	0	0	0
6	TOR	M+U			1	1	1

Systemaufrufe

- Realisierung des Zugriffs auf Machine-mode CSRs im User-mode
  - CSR-Befehle werden mit Systemaufrufen ersetzt
  - Änderungen an RIOT‘s RISC-V Implementierung nötig

- Systemcall Interface:
  - Kompatibel mit RIOT‘s irq_arch.h + thread_arch.h

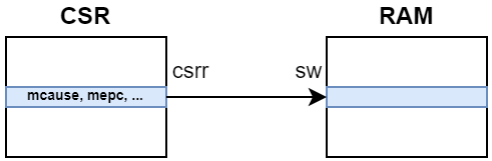
ECALL#	Funktion
0	irq_is_enabled
1	irq_disable
2	irq_restore
3	thread_yield_higher

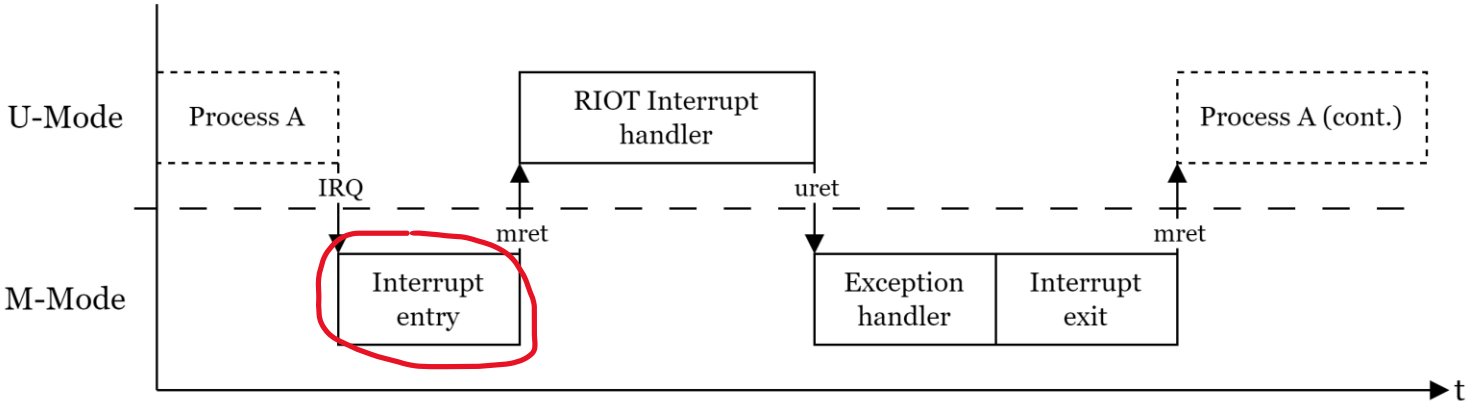
  - Aktivieren, Deaktivieren von Interrupts:
    irq_is_enabled, irq_disable, irq_restore / irq_enable
  - Auslösen von Software Interrupts:
    thread_yield_higher

Interrupt Behandlung

- Secure Firmware leitet Interrupts an den User-mode Interrupt handler von RIOT weiter
- User-mode erhält Zugriff auf relevante Machine-mode CSRs über Shared Memory

1. M-mode „Interrupt entry“:
   - Lädt CSRs mcause, mepc, mie, hartid in Shared Memory
   - Springt zu RIOTs Interrupt handler in User-mode

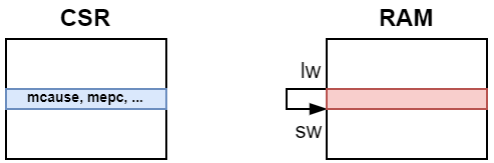


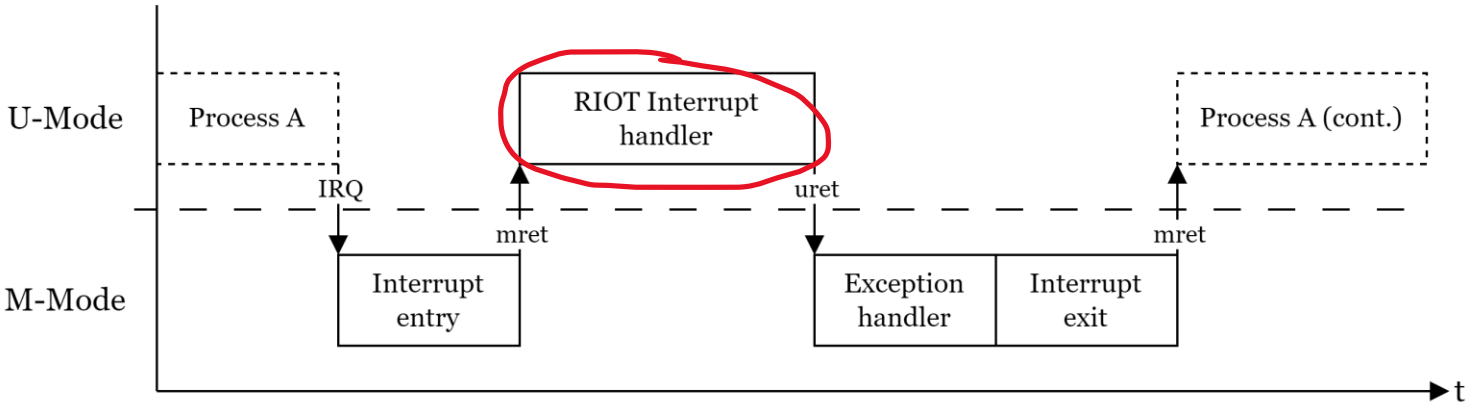


Interrupt Behandlung

- Secure Firmware leitet Interrupts an den User-mode Interrupt handler von RIOT weiter
- User-mode erhält Zugriff auf relevante Machine-mode CSRs über Shared Memory

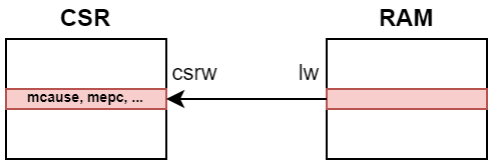
1. M-mode „Interrupt entry“:
   - Lädt CSRs mcause, mepc, mie, hartid in Shared Memory
   - Springt zu RIOTs Interrupt handler in User-mode
2. U-mode „RIOT Interrupt handler“:
   - Behandelt Interrupts: Liest, modifiziert CSRs im Shared Memory
   - Terminiert mit illegal instruction



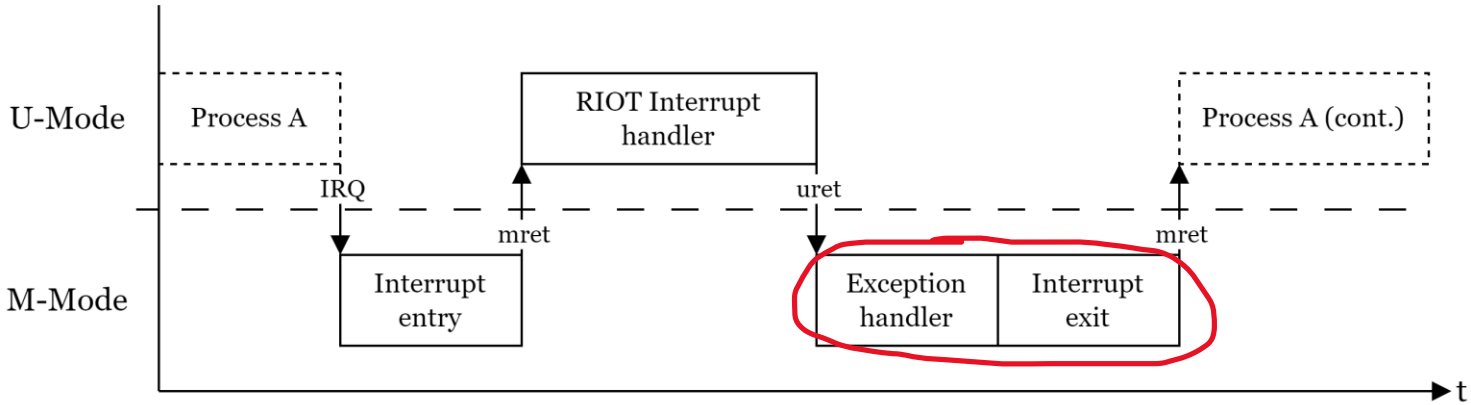


Interrupt Behandlung

- Secure Firmware leitet Interrupts an den User-mode Interrupt handler von RIOT weiter
- User-mode erhält Zugriff auf relevante Machine-mode CSRs über Shared Memory



1. M-mode „Interrupt entry“:
   - Lädt CSRs mcause, mepc, mie, hartid in Shared Memory
   - Springt zu RIOTs Interrupt handler in User-mode
2. U-mode „RIOT Interrupt handler“:
   - Behandelt Interrupts: Liest, modifiziert CSRs im Shared Memory
   - Terminiert mit illegal instruction
3. M-mode „Interrupt exit“:
   - Beschreibt CSRs mit Werten aus Shared Memory
   - Führt „mret“-Befehl aus

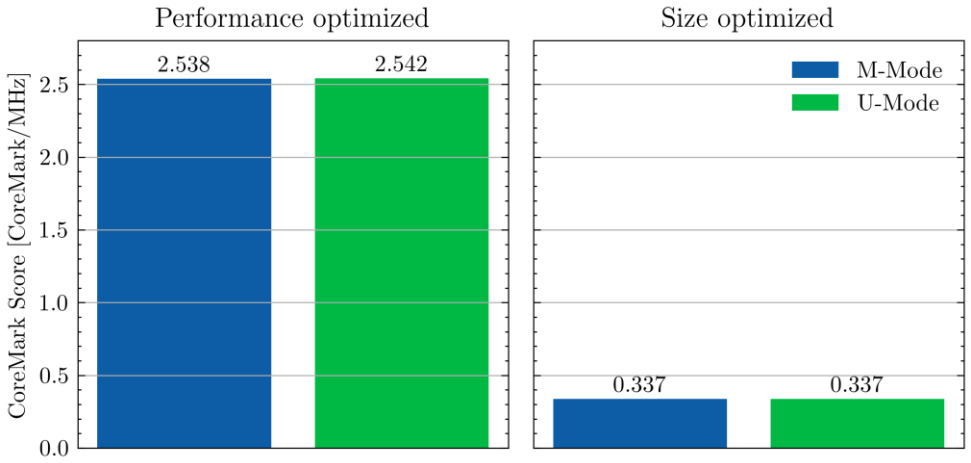


Evaluation

- Secure Firmware + RIOT funktionieren …
- … aber Systemaufrufe und Interrupt Behandlung verursachen Performance-Overhead
- Messung des Performance-Overheads
  - Performance Benchmarks mit/ohne Secure Firmware
  - Vergleich der Benchmark Scores

CoreMark

- Benchmark misst die Performance von typischen Algorithmen und Datenstrukturen in Embedded Systems
- Zwei Versuchsdurchläufe:
  - gcc –O3 (SiFive default)
  - gcc –Os (RIOT default)
- Messungen wurden 10x wiederholt. Die Ergebnisse sind exakt wiederholbar.
- U-mode + PMP haben keinen Einfluss auf die Rechenleistung des Mikrocontrollers



Microbenchmarks (Systemaufrufe)

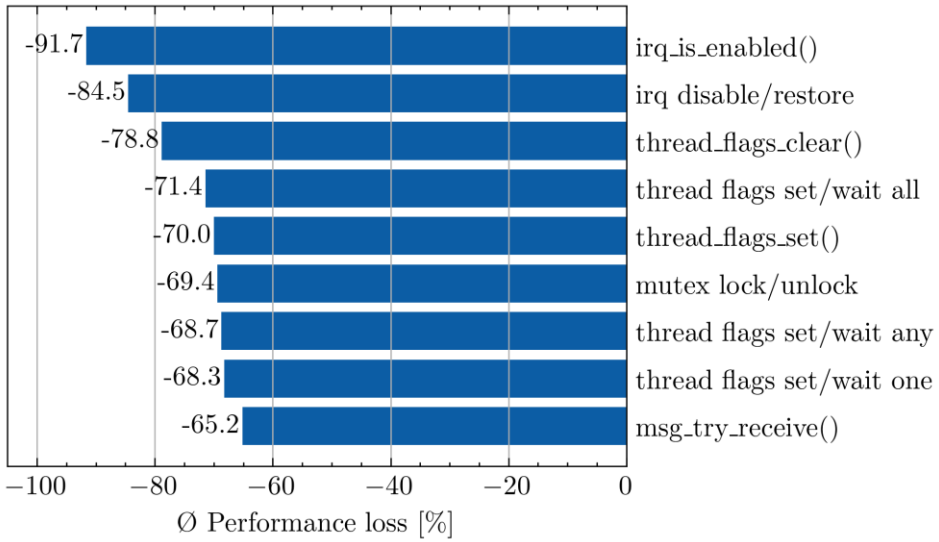
- Angepasste Version von RIOT‘s **runtime_coreapis** Benchmark
  - irq_is_enabled, irq_disable, irq_restore hinzugefügt
  - Laufzeit der Benchmarks erhöht auf min. 1 Sekunde

- „runtime_coreapis“ misst die Laufzeit von Funktionen der core-API von RIOT
  - Führt Funktionen in einer Schleife aus
  - Bestimmt die Iterationen pro Sekunde

- Messungen wurden 10x wiederholt
  - Ergebnisse wiederholbar < 0,1%

Microbenchmarks (Systemaufrufe)

- Performance im User-mode zwischen 91,7% und 65,2% geringer
- irq_arch-API am stärksten betroffen (= Systemaufrufe)
- IPC-Mechanismen weniger betroffen
- thread_flags, mutex, msg nutzen irq_disable/restore für gegenseitigen Ausschluss

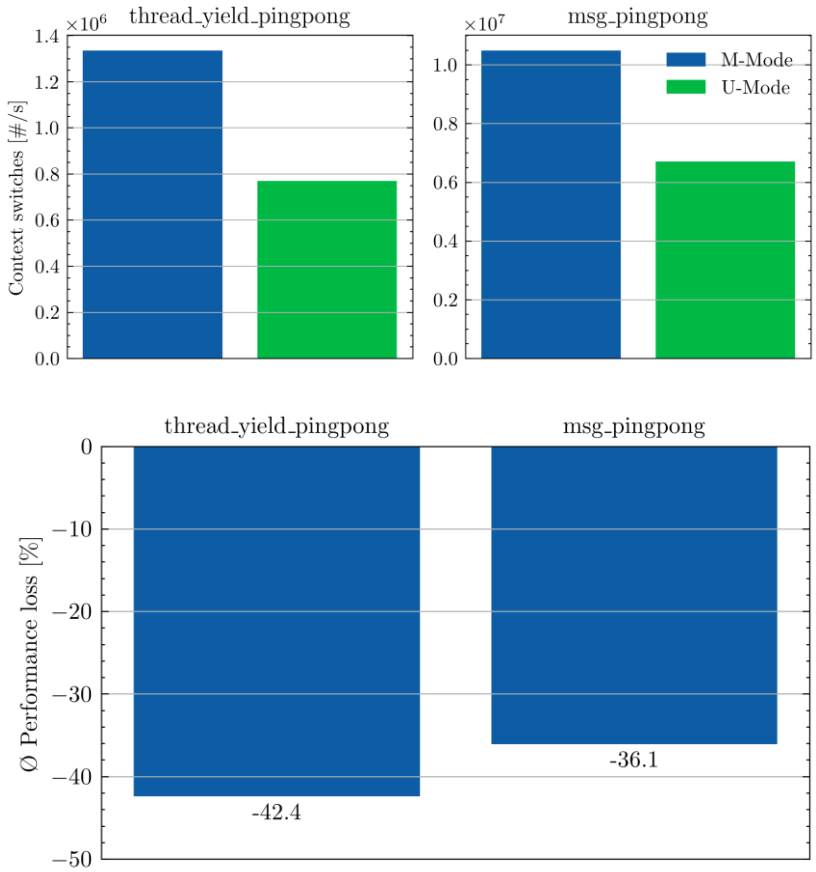


Kontextswitches (Interrupts)

Benchmarks:

1. thread_yield_pingpong:
   - Zwei Threads rufen für 1s thread_yield_higher auf
   - Messung: Anzahl der Kontextswitches pro Sekunde
2. msg_pingpong:
   - Sender-, Receiverthread senden Messages (msg_send, msg_receive)
   - Messung: Anzahl der Kontextswitches pro Sekunde

- Versuchsdurchführung:
  - Messungen 10x wiederholt
  - Abweichungen < 0,1%
- Ergebnisse:
  - Performance um 42,4% und 36,1% geringer



Schlussfolgerungen

- User-mode hat keinen Einfluss auf die Compute-Performance des Mikrocontrollers
  - CoreMark Score unverändert
- User-mode Systemaufrufe verursachen signifikanten Overhead
  - irq_arch-API bis zu 92% langsamer
  - IPC-Mechanismen bis zu 80% langsamer
- User-mode Interrupts sind deutlich langsamer
  - Bis zu 43% weniger Kontextswitches

- Eine normale Anwendung ist ein Mix aus Compute und IPC
- Performanceverlust einer normalen Anwendung erwartbar im unteren zweistelligen Prozentbereich

Zusammenfassung

- Mit Trusted Execution Environments können Sicherheitsprobleme im Internet of Things adressiert werden
- RISC-V Secure Embedded Systems unterstützen prinzipiell TEEs
  - Dafür muss RTOS im User-mode laufen
  - Problem: Interrupt Behandlung / CSR-Zugriffe müssen emuliert werden → Einfluss auf Performance?
- Portierung von RIOT für RISC-V User-mode
- Implementierung einer minimalen Secure Firmware
  - Startet RIOT in User-mode
  - Delegiert Interrupts an User-mode
  - Stellt Systemaufrufe für CSR-Zugriffe bereit
- Ergebnis: Trusted Execution Environments auf RISC-V Secure Embedded Systems sind feasible

Ausblick

1. Weiterentwicklung der Secure Firmware
   - Implementierung von Keystores, Secureboot, …
   - Packaging, Integration in RIOT Buildsystem

2. Sicherheitsanalyse
   - Kann RIOT Daten der Firmware auslesen?
   - Kann RIOT aus dem User-mode ausbrechen?

3. Verbesserung der Metriken
   - Performance, Latenzen
   - Speicherverbrauch, Energieverbrauch

Diskussion