Scale4Edge

Scale4Edge

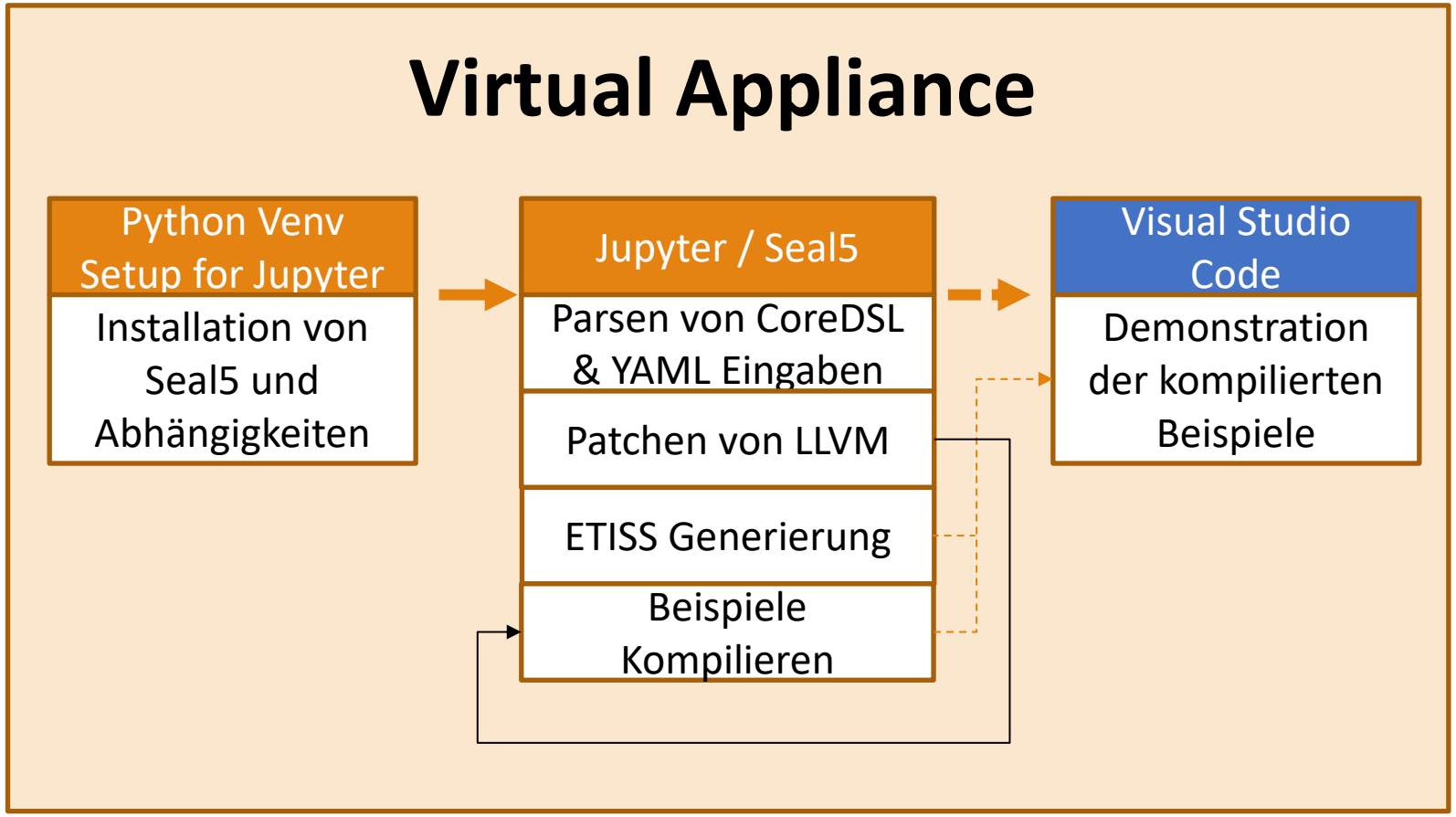
2. Statusseminar
Deutsches Zentrum für Luft- und Raumfahrt
Erweiterbare RISC-V Compiler Toolchain (Extensible Compiler) and Road Show

Jan Schlamelcher, Bewoayia Kebianiyor, Thomas Goodfellow, Kim Grüttner

This work has been developed in the ZuSE project Scale4Edge. Scale4Edge is funded by the German ministry of education and research (BMBF) (reference numbers: 16ME0122K-16ME0140+16ME0465). The authors are responsible for the content of this publication.

SPONSORED BY THE
Federal Ministry of Education and Research

B1.6.2 Scale4Edge Technologiedemonstration (Roadshow)



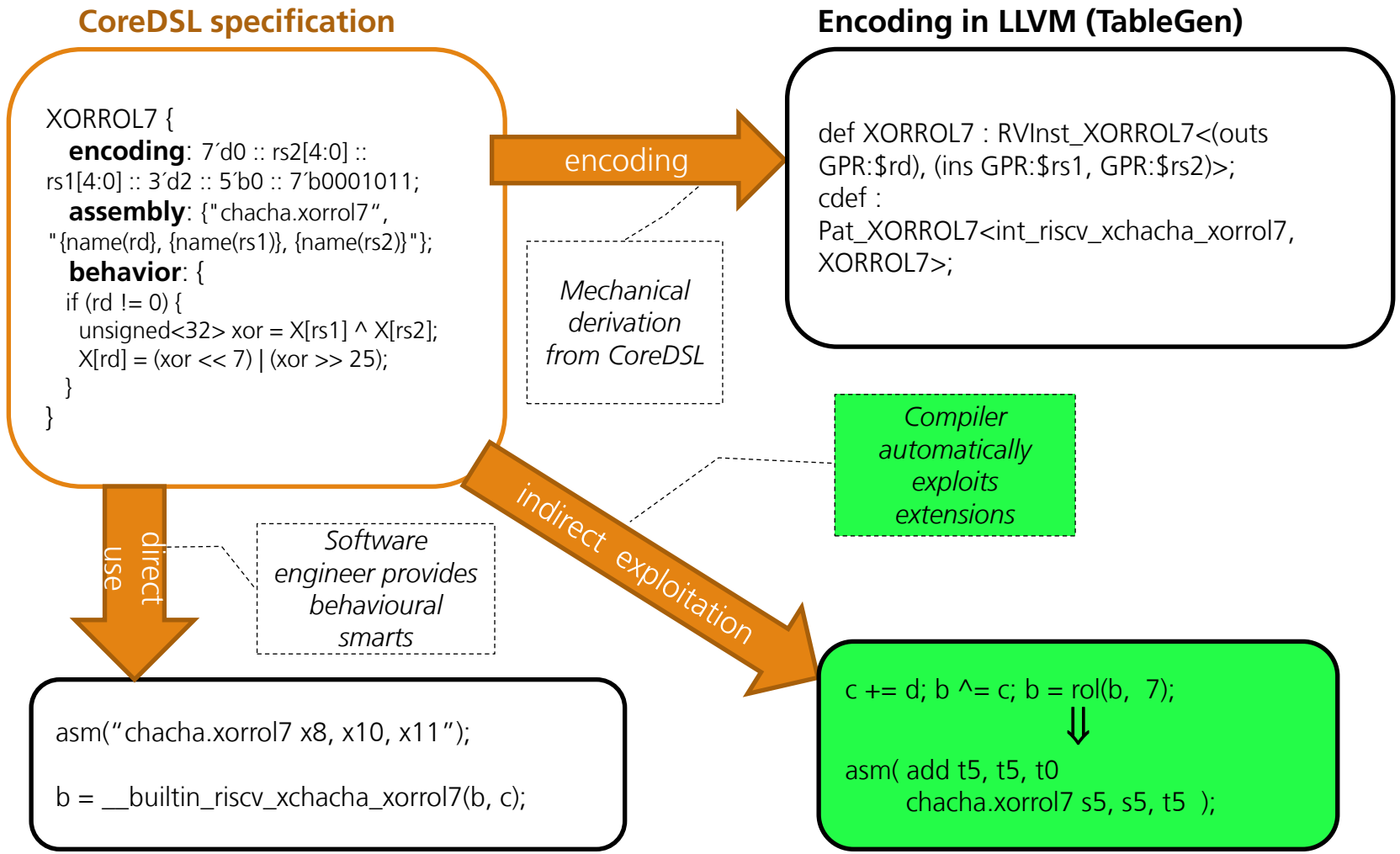
- Virtual Appliance für zuverlässige Demo Verfügbarkeit
- Jupyter Notebook mit Erläuterungen zu den einzelnen Schritten der Demo
- Öffentlich vorgeführt
- Demo für ISS (ETISS via Visual Studio Code)
- Veröffentlicht unter: https://github.com/tum-ei-eda/seal5/wiki/ChaCha20-Demonstrator

B3.3.1 DSL Support in Bezug auf SW and SDK Unterstützung

- **Ziel: Erzeugen von Libraries / SDKs mit Unterstützung für Custom Instructions**
- **Umsetzung fertiggestellt:**
  - Anwendungsfall: Erzeugen der passenden Implementierung anhand verfügbarer Intrinsics / Instructions
  - Eingabe: CoreDSL + DSL mit Function Call Interface
    - Beschreibung unterschiedlicher Implementierungen für als Alternativen für unterschiedliche RISC-V Extensions
  - Automatische Auswahl der passenden Implementierung Anhang der tatsächlich verfügbaren Instructions
  - Erfolgreich an Beispielen aus dem Projekt erprobt
    - Alternative Implementierungen von „dot“ mit automatischer Auswahl
    - Alternative Implementierungen von „subincacc“ mit überschreiben der Implementierung einer existierenden Bibliothek, falls von der Zielhardware unterstützt
  - Masterarbeit: „Automated Generation of C++ Code for Custom RISC-V ISA using SDK Descriptions“, von Christian Pfefferkorn, Betreut durch Jan Schlamelcher (DLR), an der Universität Oldenburg
- **Offene Arbeiten**
  - Anwendung bisher auf kleiner Beispiele aus dem Projektkontext.
  - Anwendung der Ergebnisse auf ein komplexes Beispiel, etwa eine größere vorhandene Software Bibliothek, die für einen Chip optimiert werden soll.

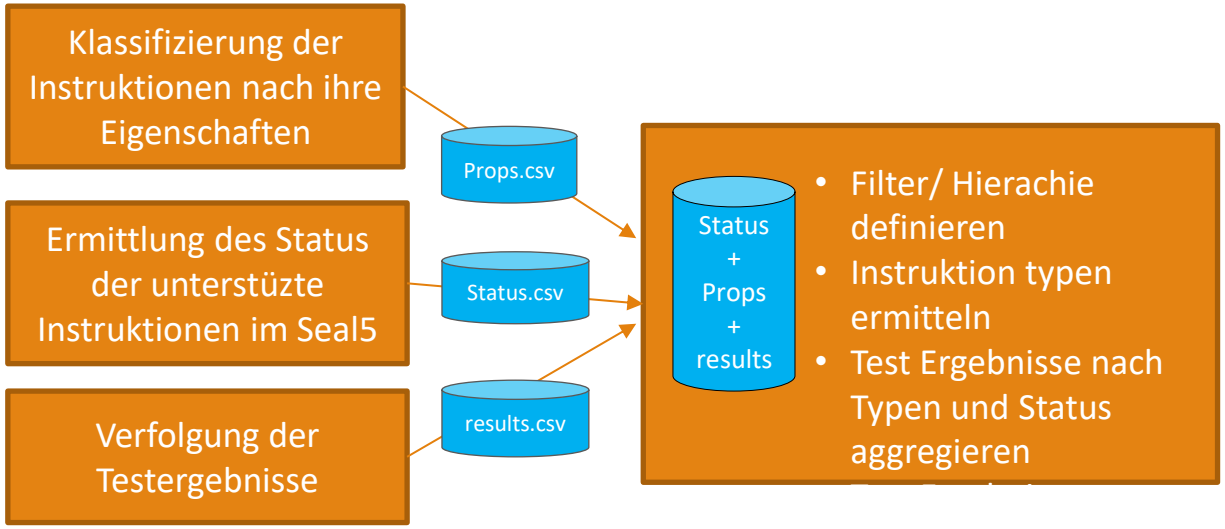
B3.3.2 Erweiterung Extensible Compiler bzgl. automatischer High-Level Code Inferenz von Custom Instruction Behavior

- **Ziel: Automatische Nutzung von benutzerdefinierten Instruktionen aus High-Level-Code**
  - Generierung von Pattern aus CoreDSL-Verhaltensblöcken
    - Werden vom Compiler verwendet, um die effizientesten Instruktionen für eine bestimmte Codesequenz auszuwählen
    - basierend auf dem SEAL5-Tooling, das in Kooperation mit der TUM derzeit entwickelt wird
- **Status**
  - Funktioniert für kombinierte arithmetische und logische Operationen
  - Systematische Tests mit umfangreicheren Verhaltensweisen

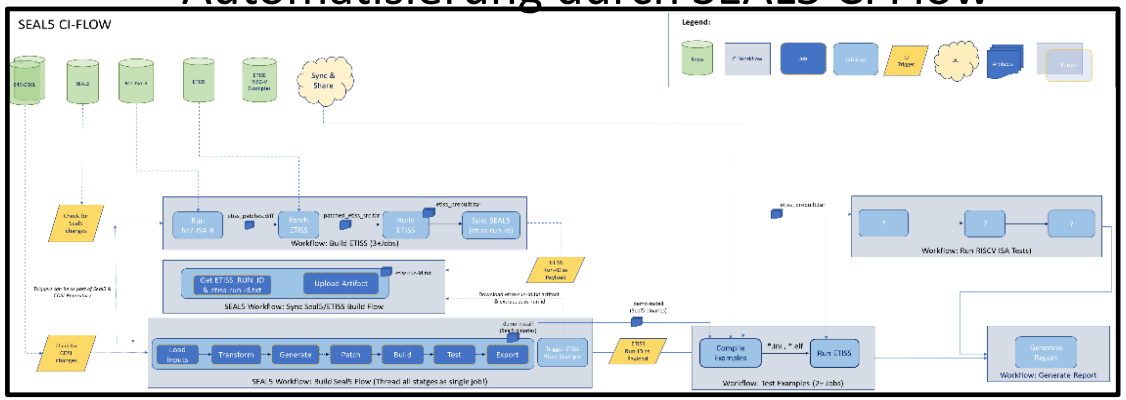


B3.3.3 Vollständiger Custom ISAX Support auf Basis von Instruction Types (1/2)

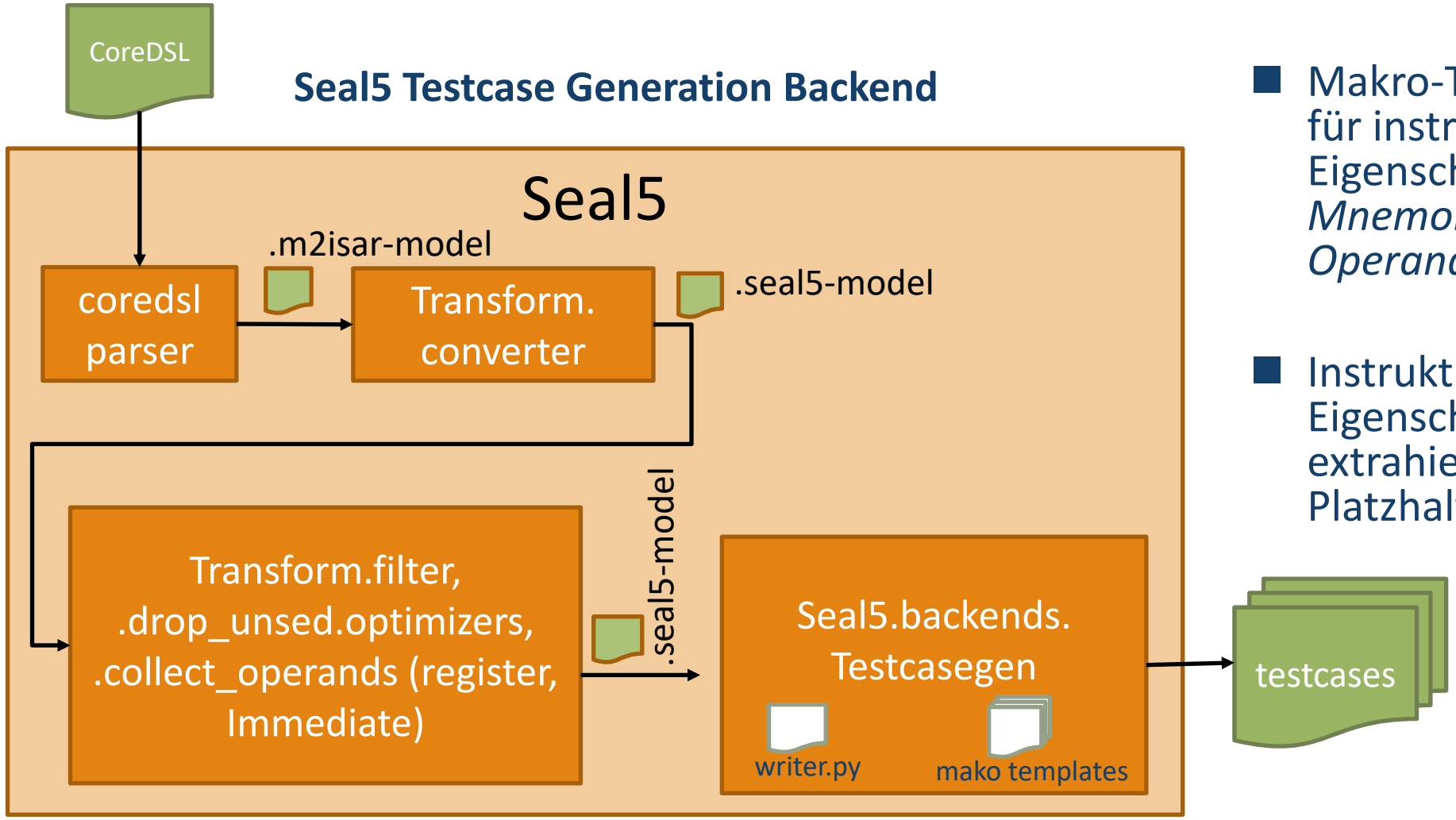
- **Ermittlung von benutzerdefinierten Instruktionen in Instruktionsklassen**
  - Identifikation und Klassifizierung basierend auf Kriterien wie:
    - Zielarchitektur (RV32I, RV64I und RV128I),
    - Opcode-Struktur
    - Funktionscodes
    - Argumenttypen (ob Register-Operationen, Speicherzugriffe oder direkte Konstantenverarbeitung benötigt werden)
  - Kategorisierung der Instruktionen in unterschiedliche Klassen, z.B.:
    - Arithmetische Erweiterungen
    - Vektoroperationen
    - Speicherzugriffe
    - Spezielle Steueroperationen
- **Analyse der unterstützten Instruktionen im Seal5**
- **Test Coverage mit Hilfe existierender Tests und GitHub CI Workflows**
  - Automatische Generierung der Testfälle mit Hilfe neu entwickelte Seal5 Backend (auf der nächsten Folie)



Automatisierung durch SEAL5 CI Flow



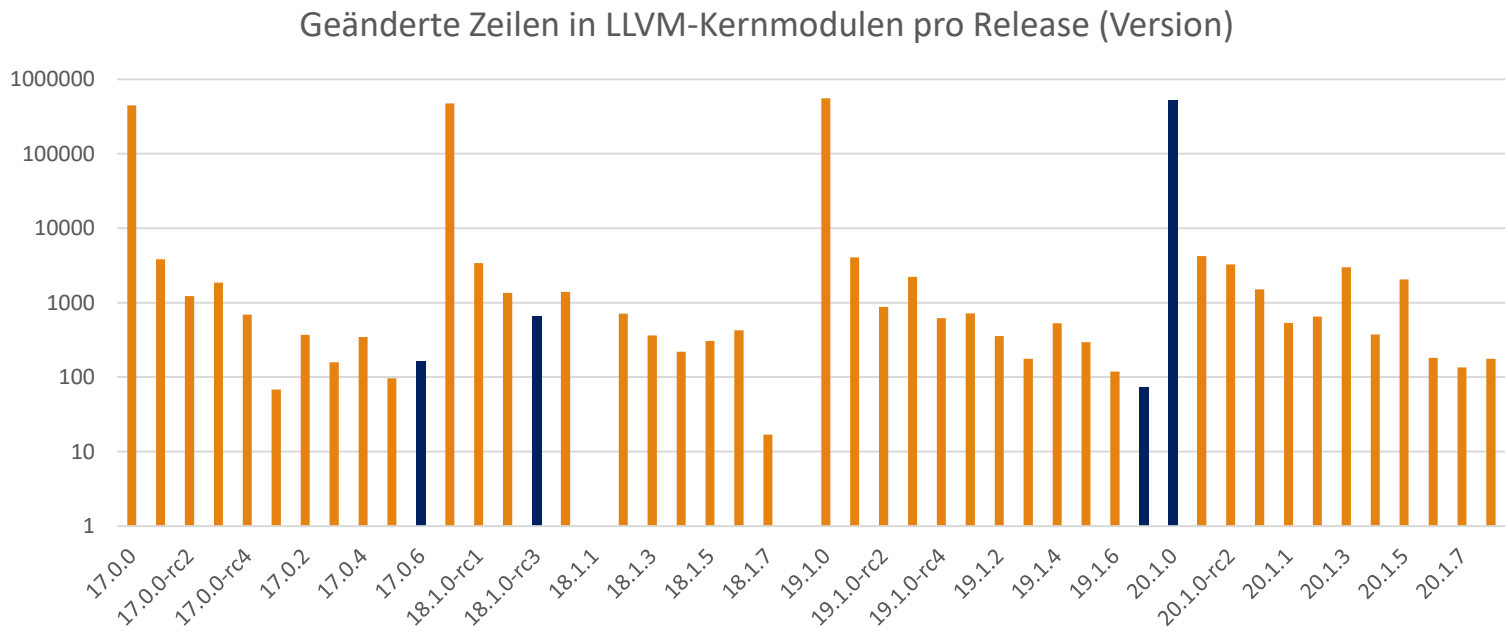
B3.3.3 Vollständiger Custom ISAX Support auf Basis von Instruction Types (2/2)



- Makro-Templates mit Platzhaltern für instruktionsspezifische Eigenschaften wie *Name, Opcode, Mnemonic, Encoding und Operanden*
- Instruktionsspezifische Eigenschaften aus Seal5model extrahieren und in den Platzhaltern ergänzen

B3.3.4 Pflege bzgl. Kompatibilität zu neuen Versionen von Clang/LLVM (1/2)

- **Instruktionen durch Quellcode-Patches zu LLVM hinzugefügt.**
- **Veränderung des Inhalts und des Speicherorts zwischen den verschiedenen LLVM-Versionen**
  - Die LLVM-Codebasis unterliegt einem schnellen Wandel: Die Zieldateien der Patches ändern sich in jeder Version mehrfach mit zusätzlicher tiefgreifenden Refactorings
  - Fortlaufende Wartungsarbeiten benötigt, um das Patchen an neue LLVM-Releases anzupassen



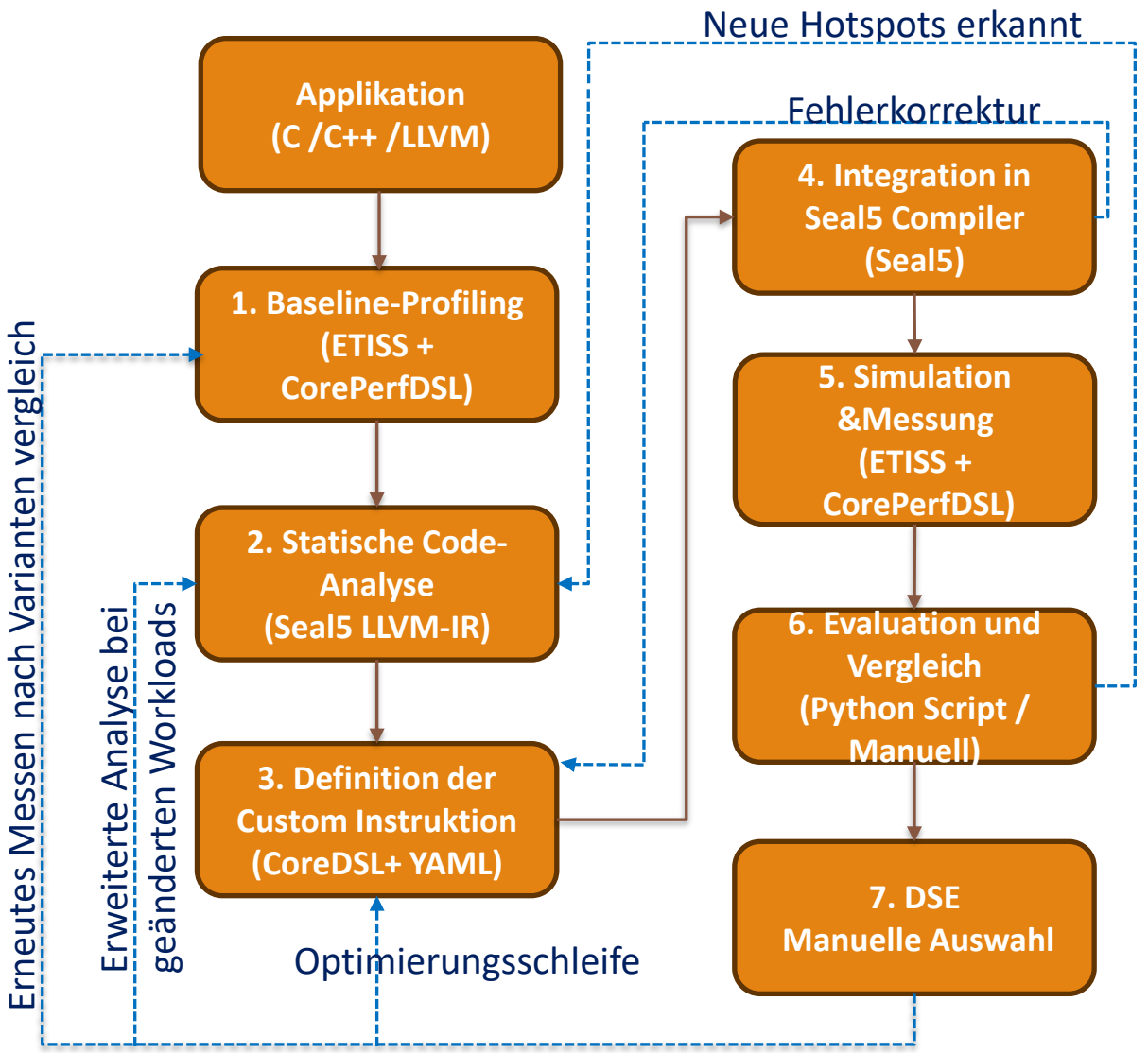
Änderungen in Minor-Releases sind < 1% des Codes im Vergleich zu den Major-Releases. Das Patchen ist vollständig unterstützt bei den blau markierten Releases. Diese sind meistens auch kompatible zu den nachfolgenden Minor-Releases

B3.3.4 Pflege bzgl. Kompatibilität zu neuen Versionen von Clang/LLVM (2/2)

- **Geplante Verbesserung war ein Erweiterungsmechanismus für LLVM:**
  - neue API – diese verbergen die Instabilität der LLVM-internen APIs durch Adapter
  - hat sich nach eingehender Untersuchung als nicht praktikabel herausgestellt
    - die Anforderungen für die Adapter wurde immer komplexer je mehr Abhängigkeiten der Extensibel Compiler zu internen LLVM APIs hinzubekommen hat
    - aufgrund umfangreicher API-Anpassungen/Änderungen in schnell aufeinanderfolgenden LLVM Releases weitere Erhöhung der Komplexität der Pflege und Anpassung von Adaptern
    - Übernahme der Adapter in LLVM nur dann vom Aufwand her vertretbar, wenn diese in dem Hauptbaum (in-tree) von LLVM genutzt werden, also z. B. zumindest von RISC-V verwendet wird und die Verpflichtung zur Weiterpflege nach dem Scale4Edge Projekt garantiert werden kann
  - Stattdessen wurde der bereits genutzte Patching-Mechanismus weiter optimiert
    - Der Patching-Mechanismus wurde weniger ortsabhängig gemacht
    - Code-Refactoring, das Patch-Ziele umstrukturiert, ohne deren Inhalt zu verändern

B3.3.5 Nutzung des Extensible Compilers zur Identifikation und Auswahl von Custom Instructions

1. **Baseline-Profiling** (ETISS + CorePerfDSL)
   - Erfasst Metriken (Ausführungszeit, Codegröße, Datengröße)
2. **Statische Code-Analyse (Seal5 LLVM-IR)**
   - Hotspot- und Mustererkennung, Kandidatenextraktion für Custom Instruktionen
3. **Definition (CoreDSL + YAML)**
   - Funktionale und syntaktische Beschreibung der Kandidaten der Custom Instruktionen aus der Analyse
4. **Integration in Seal5 Compiler**
   - Erweiterung des Toolchains mit Hilfe der CoreDSL und YAML Dateien
5. **Simulation & Analyse (ETISS + CorePerfDSL)**
   - Testen und Messung der Performance mit neuen Custom Instruktionen.
6. **Evaluation & Vergleich**
   - Performance Messung und Vergleich zur Baseline
7. **Design Space Exploration**
   - Manuelle Bewertung verschiedener Instruktions-kombinationen und Auswahl der optimalen Konfiguration



B3.7.2 Integration von TVM-ML-Compiler mit Extensible Compiler

- **Verknüpfung über intrinsische Funktionen (intrinsics)**
  1. Intrinsics werden über YAML beschrieben und auf Instruktionen abgebildet.
  2. LLVM mit Support für Intrinsics wird mittels Seal5 erzeugt
  3. TVM erzeugt optimierten Code unter Verwendung der Intrinsics
- **Ansatz kann mit automatischer Selektion von Instruktionen kombiniert werden**
- **Dieses Arbeiten wurden TVM seitig von der TUM durchgeführt**