Microsoft

Hardware Trust, Firmware Assurance: A Rust-based Root-of-Trust for Modern Secure Systems

Speakers: Xiling Sun, Parvathi Bhogaraju

Oct. 1st , 2025

Motivation

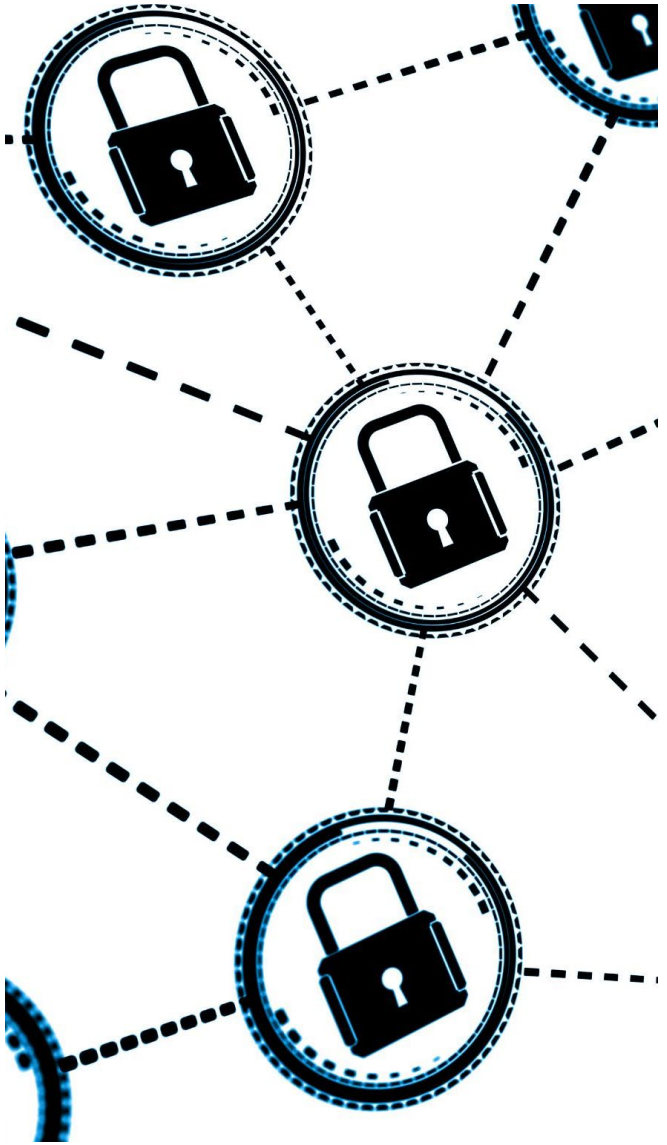
Increasing Firmware Attacks
Firmware is a growing target for sophisticated attackers seeking to bypass higher-level security controls

Secure-by-Design Principle
Modern platforms require robust security foundations starting from the hardware Root-of-Trust (RoT)

Role of Hardware Root-of-Trust
Establishes a secure and verifiable foundation for launching upper-level firmware and software layers

Limitations of Traditional C-based RoT Firmware
Vulnerabilities such as buffer overflows, use-after-free, and undefined behavior are common in C, especially in privileged code

Modern RoT Firmware with Rust
Utilizing Rust and Tock OS enhances firmware security through memory safety and microkernel isolation



Design Goals

Memory Safety
Mitigate vulnerabilities using Rust’s ownership model and type system.

Modularity & Isolation
Tock OS microkernel isolates components, reducing fault propagation.

Scalability
Support diverse platforms with reusable, configurable components.

Open Development
Higher security transparency and implementation visibility.

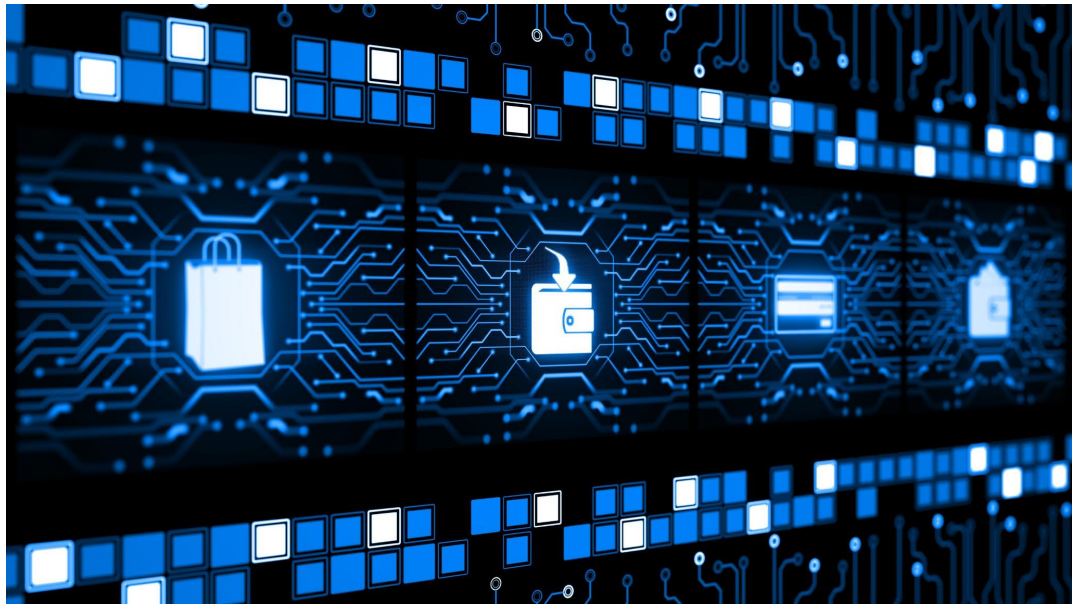
Threat Model

Adversary Capabilities

- Tamper with firmware during manufacturing or supply chain
- Exploit runtime vulnerabilities to gain privileged access
- Attempt to bypass secure boot or measurement mechanisms
- Interfere with cryptographic operations or key management
- Physical attacks (e.g., side-channel, fault injection)

Defensive Focus (Firmware)

- Strict memory safety
- Process isolation
- Integrity from boot through attestation



Caliptra Security Subsystem Overview

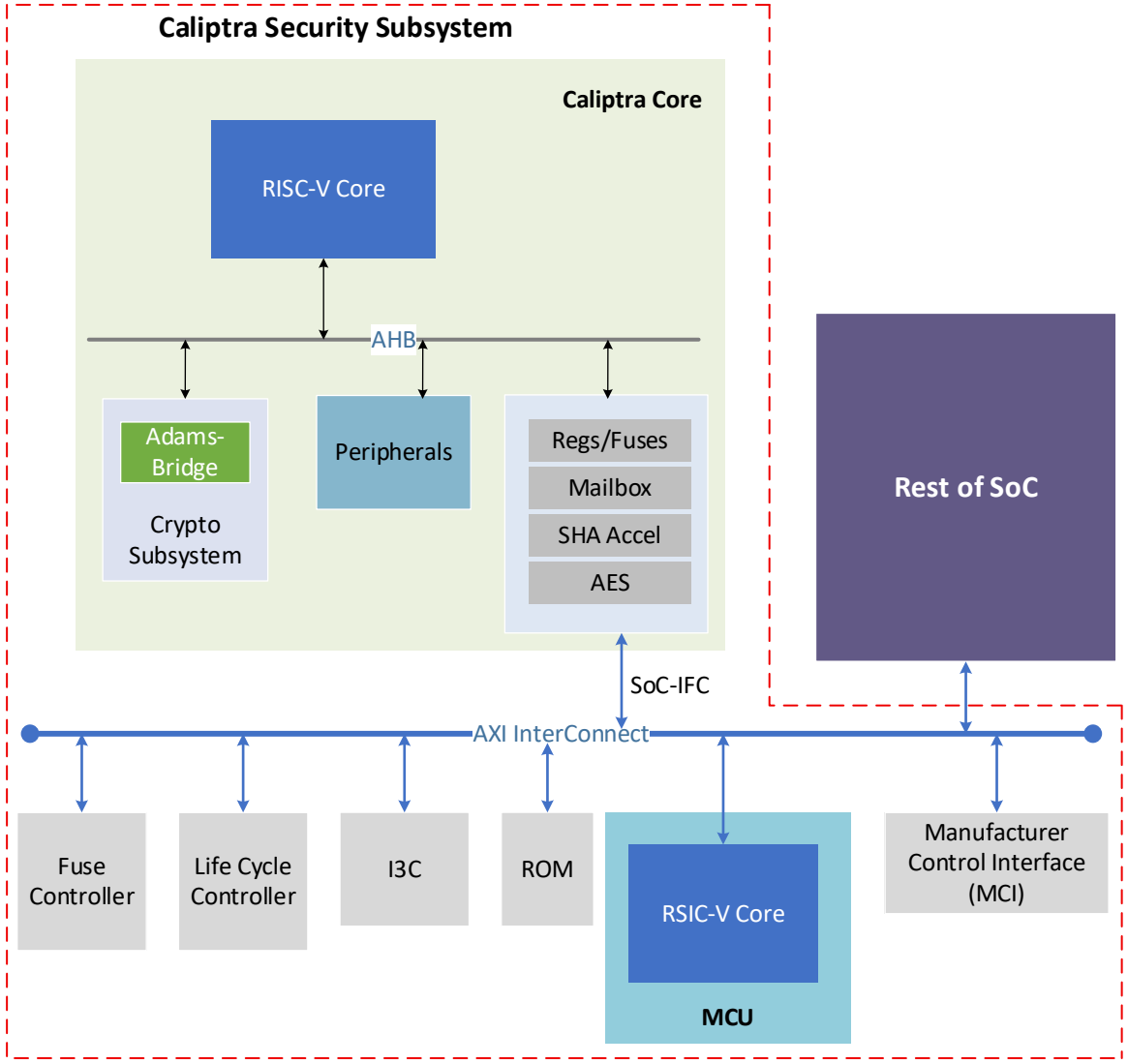


Fig. 1 Simplified Caliptra Security Subsystem Architecture

- Open-source, standardized hardware Root-of-Trust (RoT) for wide range of System-on-Chip(SoC) platforms
- Anchors secure boot, attestation, measurement, and cryptographic firmware updates and secure I/O
- MCU (Manufacturing Control Unit) orchestrates platform-level security and lifecycle management
- Hardware-backed cryptography and measured boot defend against supply chain threats and runtime attacks
- Modular design enables interoperability and scalability across diverse platforms

MCU Design Highlights

Architecture

- Open-source RISC-V
  - VeeR EL2 core, SoC-agnostic design
  - Modular Tock OS drivers/capsules for hardware abstraction
- Dual-stage firmware stack
  - Bare-metal ROM for hardware initialization
  - Runtime firmware for RoT services

Secure RTOS Integration - Tock OS

- Rust-based microkernel
- Enforces process isolation and memory safety
- Peripheral drivers as capsules for clean separation and extensibility

Security Features

- Secure firmware update and attestation
- Hardware-backed cryptographical services
- Anti-rollback protection

Streaming Boot

- Eliminates persistent firmware storage: reducing an attack interface
- Streams and validates firmware at boot dynamically and securely

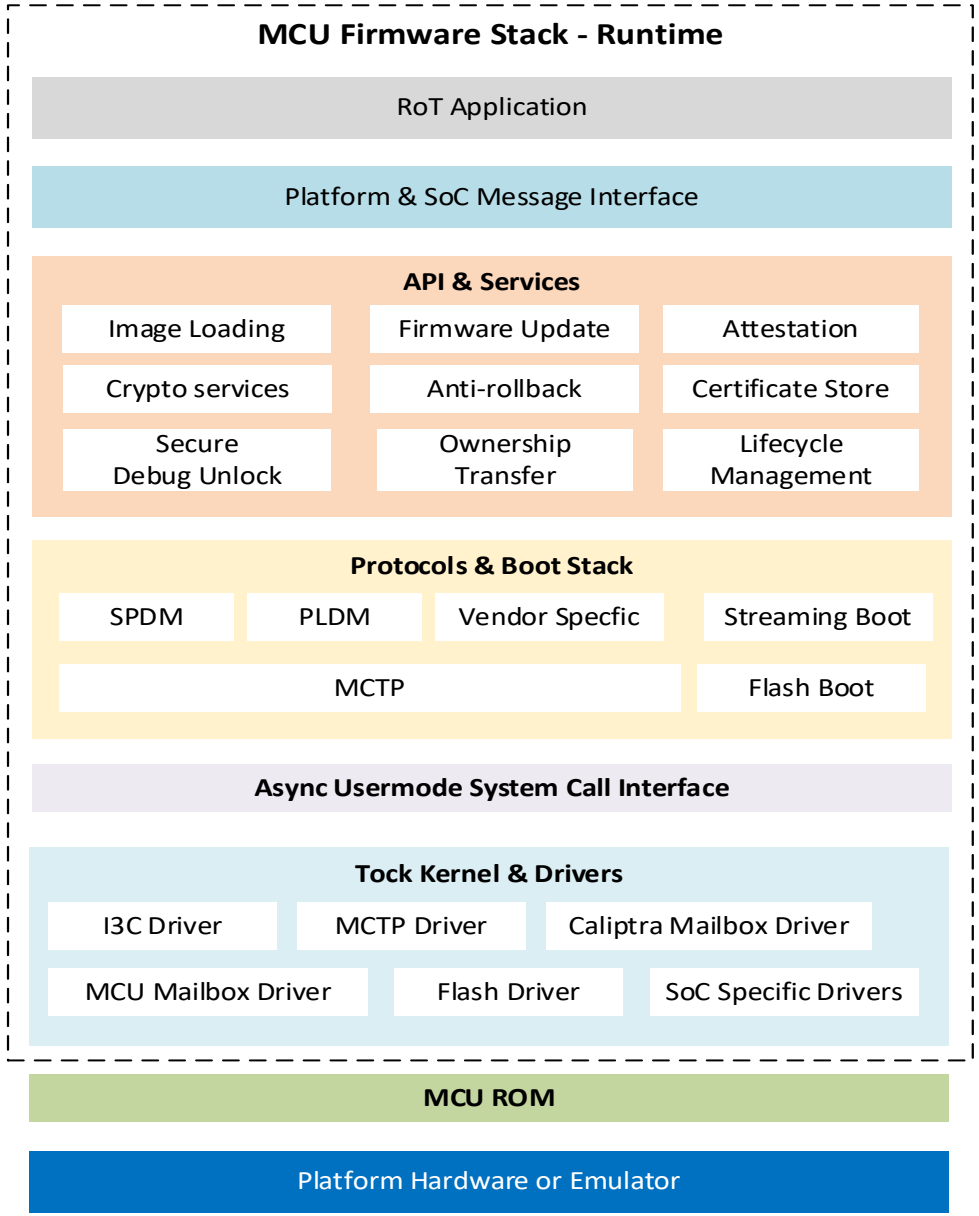


Fig. 2 MCU Firmware Stack

Protocol Stack & Implementation

Supported Protocols

- SPDM, PLDM, Caliptra vendor-defined messages

Async interface

- Provide async Rust APIs for sending/receiving messages from userspace.

Memory-safe isolation with Tock Capsules

- MCTP base and control protocol handling.
- Virtualization: Multiple virtual MCTP drivers per board , each protocol assigned a unique driver number.
- Mux Layer: Centralizes transmission/reception, tags messages, manages outstanding requests
- Transport binding: Adds/removes I3C-specific header/trailer, handles Packet Error Code (PEC).

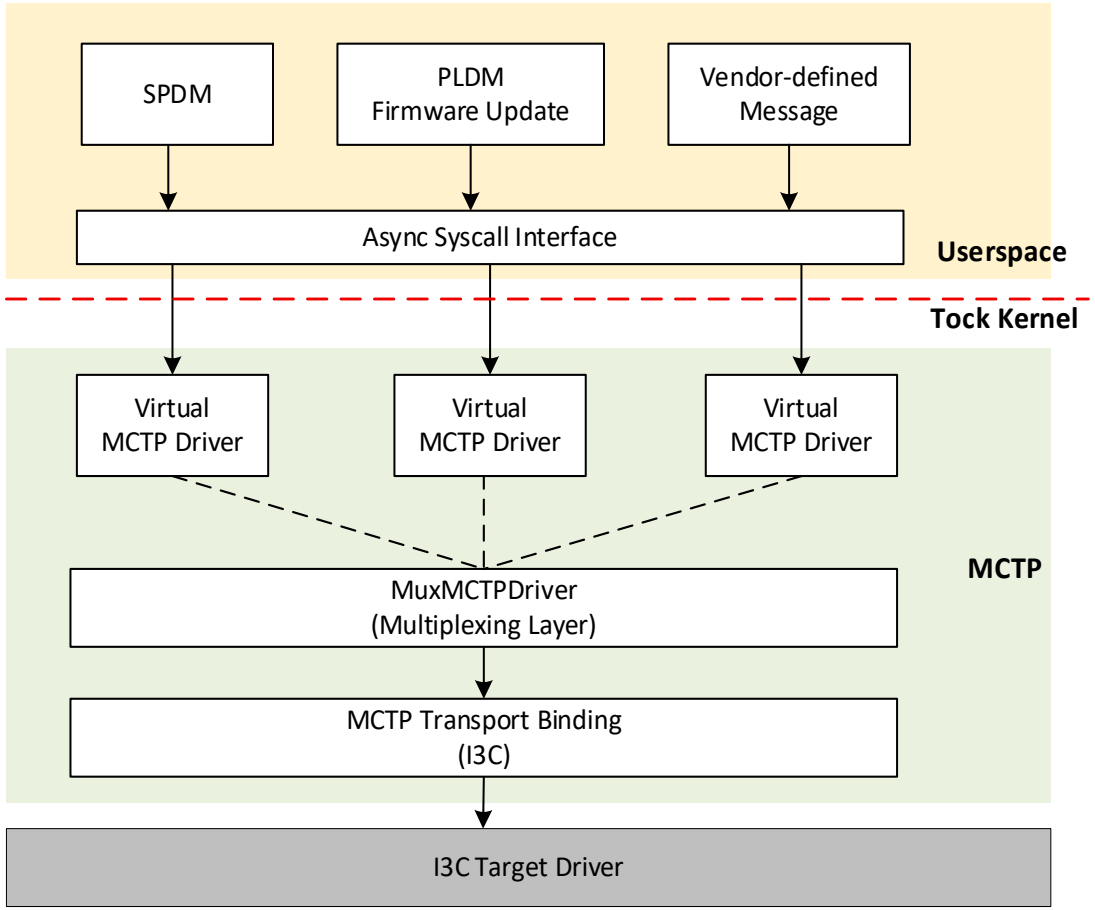


Fig. 3 Protocol Stack

Streaming Boot Flow

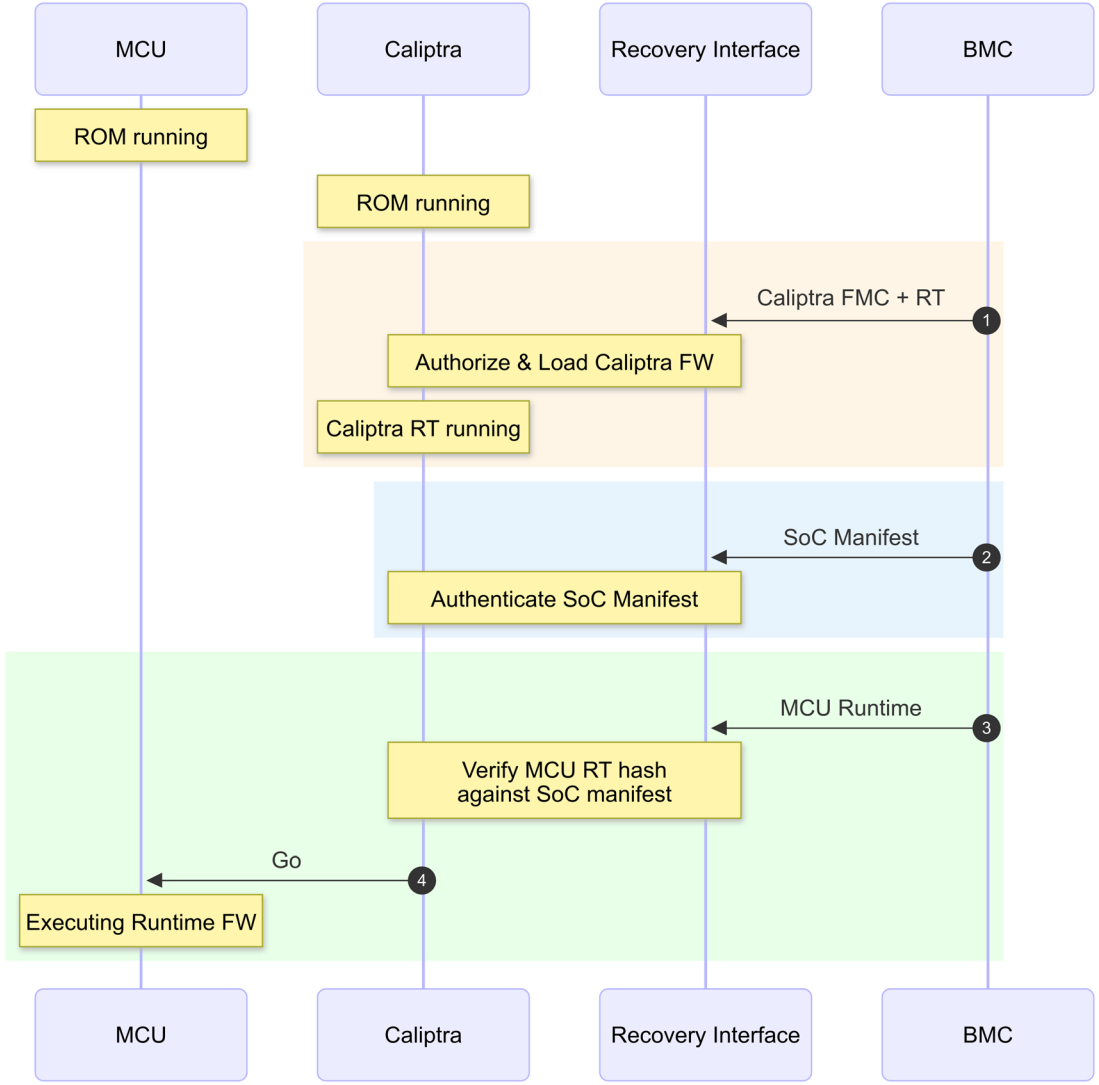


Fig. 4 Simplified streaming boot flow –stage 1

Two-stage boot sequence for secure, scalable firmware delivery.

- **Stage 1: Early Firmware Loading via OCP Recovery Protocol**
  - Caliptra FMC + RT
  - SOC Manifest
  - MCU Runtime

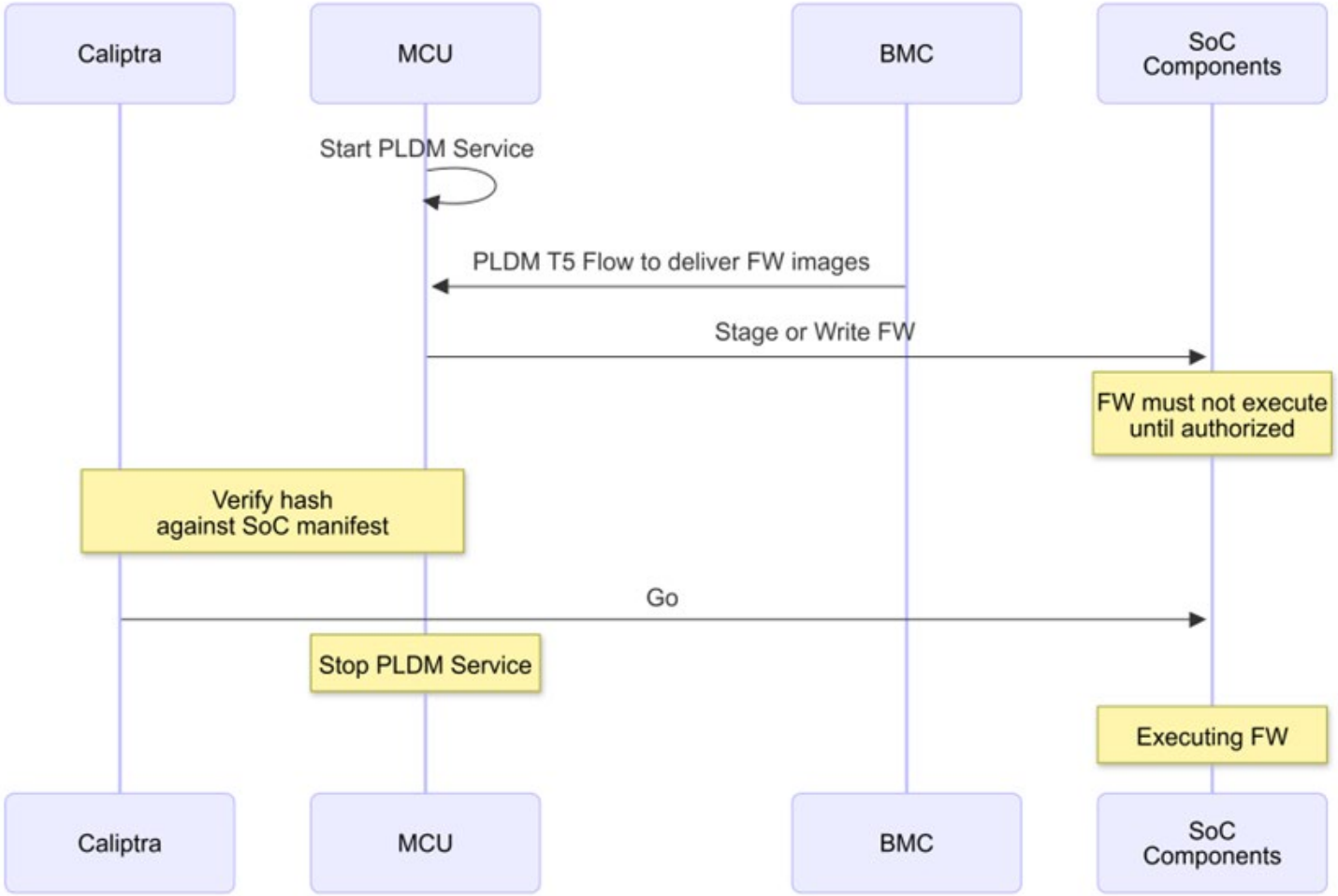


Fig. 5 Simplified streaming boot flow –stage 2

Stage 2: Remainder Firmware Loading via PLDM Firmware Update Protocol

- Enables modular, component-based firmware updates and supports “pull” model for flow control and error recovery.
- Remainder-firmware is loaded directly into device RAM, not persistent storage, enabling secure, impactless updates and rapid recovery.
- Device attests to its boot state via SPDM, ensuring integrity and compliance.

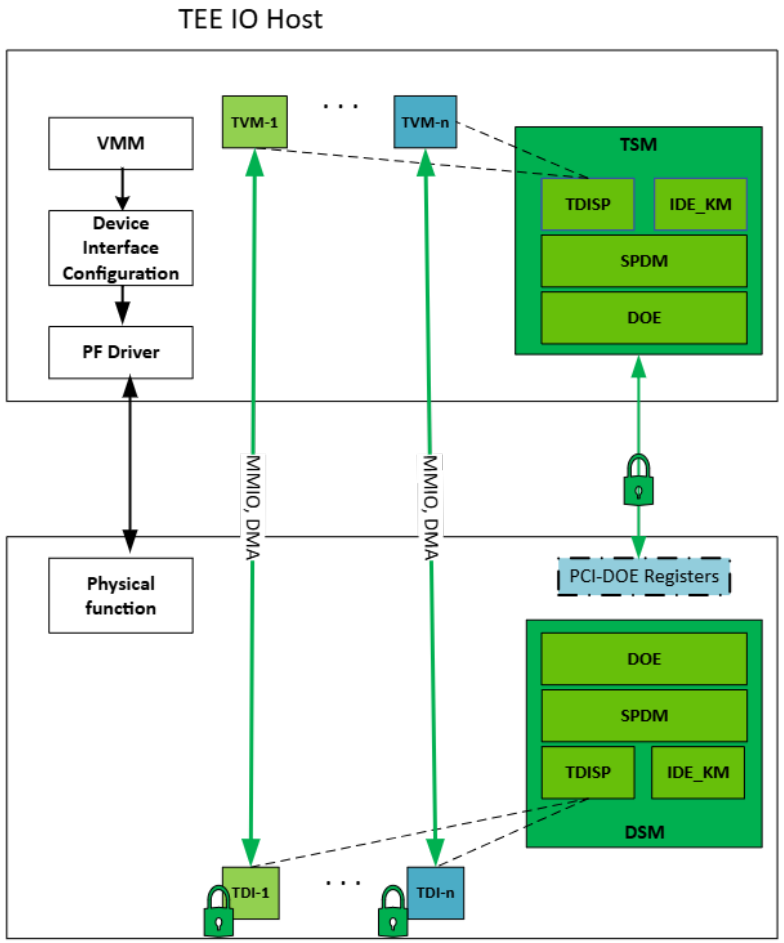
Trusted I/O for Confidential Workloads

SPDM provides cryptographic attestation to verify device identity and integrity, forming the root of trust.

IDE_KM secures PCIe data in transit by managing encryption keys, enabling end-to-end path protection.

TDISP ensures trusted assignment and isolation of device resources to virtual machines, supporting secure I/O virtualization.

Together, they enable **secure communication**, **hardware-enforced isolation**, and form the **foundation for confidential computing** across diverse platforms.



TEE IO Device with Caliptra Security Sybsystem

DOE- Data Object Exchange
DSM - Device Security Manager
IDE-KM - Integrity and Data Encryption Key Management
PF - Physical Function
SPDM - Security Protocol and Data Model
TDISP - TEE Device Interface Security Protocol
TEE - Trusted Execution Environment
TSM - TEE Security Manager
TVM - TEE VM
VM - Virtual Machine
VMM : Virtual Machine Monitor

Evaluation

Fully open-source design enables public auditing and repeatable builds
Adheres to industry standards (OCP, TCG, DMTF, PCI-SIG), ensuring interoperability and easy validation
Rust-based firmware eliminates common memory safety vulnerabilities found in C-based RoT firmware .
Modular architecture supports secure boot, attestation, anti-rollback, and lifecycle management, surpassing legacy RoT flexibility
Integrates post-quantum cryptography, preparing platforms for future threats

Rust Development Experience

Challenges

- Required a fundamental shift in development mindset and architecture due to Rust's ownership model and strict type system.
- Initial learning curve was steep. Explicit lifetimes and borrowing rules demanded careful resource management.
- Embedded debugging and hardware integration posed new workflow challenges.

Opportunities & Advantages

- Eliminated entire classes of C bugs (buffer overflows, memory aliasing, dangling pointers) through strong compile-time checks.
- Rust's async/concurrency model, combined with Tock OS microkernel, enabled safe, isolated, event-driven firmware tasks.
- Possible to achieve near C-level performance, optimization required for dynamic dispatch and heavy async.
- Modern tooling (cargo, integrated testing, LLVM) streamlined development and surfaced issues earlier in the cycle.

Microsoft

Thank you!