POLITECNICO DI TORINO

Master degree course in Cybersecurity Engineering

Master Degree Thesis

Design and Implementation of device emulation for the TCG MARS specification

Supervisor
Prof. Antonio Lioy
Doc. Enrico Bravi

Candidate
Antonio Capece

Academic Year 2025-2026

Abstract

The rapid expansion of the Internet of Things (IoT) has introduced critical security challenges that conventional software-based defenses are often insufficient to mitigate. While Hardware Roots of Trust (HRoT) mechanisms-such as the Trusted Platform Module (TPM)-effectively secure general-purpose computing platforms, their complexity and resource demands render them impractical for many constrained IoT devices. To bridge this gap, the Trusted Computing Group (TCG) introduced the Measurement and Attestation Root of Trust (MARS), a lightweight specification designed to deliver core security capabilities to resource-limited environments.

Despite its potential, MARS adoption is currently hindered by a lack of physical silicon and mature supporting software, limiting both research and practical experimentation. This thesis addresses these barriers by presenting the design and development of a modular MARS emulator implementing the specification's core command set. Implemented on Linux, the emulator features a comprehensive hardware abstraction layer that decouples application logic from underlying security mechanisms. This architecture preserves interface transparency, allowing applications to interact with the emulated MARS instance indistinguishably from actual hardware. By delivering a robust and extensible virtual testing environment, this project enables the rapid prototyping of secure IoT solutions and facilitates early validation of essential MARS capabilities prior to hardware availability.

Summary

As hardware-backed security and cryptographic Roots of Trust become increasingly central to modern computing, the MARS (Measurement and Attestation Root of Trust) specification provides a standardized interface for interacting with these secure elements. However, at present, there are no commercially available physical devices that implement the MARS specification. While the Trusted Computing Group (TCG) provides a basic reference emulator, it operates entirely in user space and fails to abstract the hardware at the system level.

This lack of a true system-level abstraction creates a significant bottleneck for the software development lifecycle. Developers require a faithful emulation environment that not only replicates the cryptographic operations of a MARS device but also accurately mimics the OS-level interactions, access controls, and concurrency constraints expected of future physical hardware. Relying purely on existing user-space abstractions is insufficient, as it fails to capture the strict mutual exclusion and synchronization required by the specification. Bridging this gap serves as the primary motivation for this thesis.

To address this, this thesis presents the design and implementation of a comprehensive, multi-layered emulation stack. The system architecture is cleanly separated into four distinct components spanning both user and kernel space: a high-level client library (`mars-api`), a kernel-level synchronization proxy (`vmars-driver`), a background execution daemon (`mars-emulator`), and a core emulation library (`libmars`).

Operating at the top layer, the `mars-api` provides a transparent interface for developers, abstracting data serialization through a flexible, CBOR-backed Marshal/Unmarshal pipeline. When an application issues a command, it is routed down to the `vmars-driver`, which acts as the secure bridge of the architecture. To guarantee thread safety and prevent device corruption, this kernel module enforces strict hardware-like mutual exclusion using atomic operations, ensuring only one application can acquire the virtual device at a time. The kernel then orchestrates a synchronous request-response cycle via wait queues, safely passing the marshaled payload up to the `mars-emulator` daemon. Finally, this background daemon leverages the `libmars` foundational library to execute the core hardware logic, manage volatile registers, and route operations through a flexible cryptographic abstraction layer.

Ultimately, this architecture provides a robust, thread-safe, and architecturally faithful virtualized environment. It allows developers to confidently build, test, and debug MARS-compliant applications with the exact same system call flow and exclusivity guarantees as physical hardware, paving the way for software readiness long before physical MARS devices reach the market.

Acknowledgements

I would like to express my deepest gratitude to my family and friends for their unwavering support throughout this journey. Furthermore, I extend my sincere thanks to Professor Antonio Lioy and Dr. Enrico Bravi for providing me with the opportunity to conduct this thesis research.

Contents

1 Introduction 6

2 Trusted Computing 7

2.1 Trusted Computing Concepts . . . 7

2.2 Root of Trust (RoT) . . . 7

2.2.1 Trusted Platform Module (TPM) . . . 8

2.2.2 Trusted Execution Environment (TEE) . . . 10

2.3 Hardware RoT for IoT . . . 10

2.3.1 Device Identifier Composition Engine (DICE) . . . 10

2.3.2 ARM TrustZone . . . 11

2.3.3 Caliptra . . . 11

2.3.4 OpenTitan . . . 12

3 Secure Device Software Implementation 14

3.1 TCG TPM Software Implementations . . . 14

3.1.1 IBM Software TPM . . . 14

3.1.2 Microsoft TPM 2.0 Reference Implementation . . . 15

3.2 TEE Software Implementations . . . 16

3.2.1 Open-TEE . . . 16

4 Measurement and Attestation RootS (MARS) 18

4.1 Architecture . . . 18

4.1.1 Serialized Architecture . . . 18

4.1.2 Memory Mapped Architecture . . . 19

4.1.3 Cryptographic Support Functions . . . 20

4.2 Root of Trust for Storage . . . 20

4.2.1 Platform Configuration Registers (PCRs) . . . 21

4.2.2 Trusted Sensor Registers (TSRs) . . . 21

4.3 Root of Trust for Reporting . . . 21

4.3.1 Key Hierarchy . . . 22

4.4 MARS API . . . 22

4.4.1 Mandatory Commands . . . 22

4.4.2 Extended Commands . . . 23

4.4.3 Initialization and Concurrency Control . . . 23

4.5 Command Serialization . . . 24

5 System Design 25
5.1 General Overview . . . 25
5.2 Architecture . . . 25
5.2.1 vmars-driver . . . 26
5.2.2 mars-api . . . 26
5.2.3 libmars . . . 27
5.2.4 mars-emulator . . . 28

6 Implementation 29
6.1 vmars-driver . . . 29
6.1.1 Character Devices and Synchronization . . . 29
6.1.2 Virtual Device Instantiation and ioctl Handling . . . 30
6.1.3 Device Exclusivity and Access Control . . . 31
6.1.4 Read/Write Operations and Execution Flow . . . 31
6.2 mars-api . . . 32
6.2.1 Initialization and Concurrency Control . . . 32
6.2.2 Marshal-Unmarshal Architecture . . . 33
6.3 libmars . . . 33
6.3.1 State and Profile Initialization . . . 34
6.3.2 Cryptographic Abstraction . . . 35
6.3.3 Sequence Management . . . 36
6.3.4 Command Dispatching and Implementation . . . 37
6.4 mars-emulator . . . 38
6.4.1 Virtual Device Instantiation . . . 38
6.4.2 mars-setup . . . 38
6.4.3 Execution Loop . . . 39

7 Testing and Evaluation 41
7.1 Experimental Setup . . . 41
7.2 Functional Validation . . . 41
7.2.1 Scenario 1: Cryptographic Pipeline and Registry Operations . . . 42
7.2.2 Scenario 2: Remote Attestation and Quote Generation . . . 42
7.2.3 Scenario 3: Concurrency and Device Exclusivity Stress Test . . . 43
7.3 Summary of Evaluation Results . . . 45

8 Conclusions 46
8.1 Summary of Contributions . . . 46
8.2 Limitations . . . 46
8.3 Future Work . . . 47
8.4 Concluding Remarks . . . 47

Bibliography 48

A User's Manual 49

A.1 Build Process . . . 49

A.1.1 Building and Loading `vmars_driver` . . . 49

A.1.2 Building `mars-api` . . . 50

A.1.3 Building `libmars` . . . 51

A.1.4 Building `mars-emulator` . . . 53

A.2 Setup and Execution . . . 53

A.2.1 Loading the Kernel Driver . . . 54

A.2.2 Device Configuration (`mars-setup`) . . . 54

A.2.3 Starting the emulator . . . 54

A.3 Interacting with the Device . . . 55

A.3.1 Example Applications . . . 55

A.3.2 Writing Your First Application . . . 55

B Developer Manual 58

B.1 Introduction . . . 58

B.2 Extending the Emulator: Implementing a Pending Command . . . 58

B.2.1 Step 1: Review the Frontend API Contract . . . 58

B.2.2 Step 2: Implement the Backend Logic (`libmars`) . . . 59

B.2.3 Step 3: Route the Command in the Dispatcher . . . 59

B.3 Extending the Emulator: Integrating a New Cryptographic Suite . . . 60

B.3.1 Step 1: Define Algorithm Identifiers and State . . . 60

B.3.2 Step 2: Update the Abstraction Routing . . . 61

B.3.3 Step 3: Link the External Library . . . 61

B.3.4 Step 4: Expose the Algorithm in `mars-setup` . . . 62

B.4 Testing and Debugging . . . 62

B.4.1 Running the Test Suites . . . 62

B.4.2 Debugging the Kernel Driver (`vmars_driver`) . . . 62

B.4.3 Debugging the Emulator Daemon (`mars-emulator`) . . . 63

B.4.4 Common Development Pitfalls . . . 63

Chapter 1

Introduction

In the contemporary computing landscape, the inherent vulnerabilities of software-defined security have necessitated a fundamental shift toward hardware-anchored trust. Historically, securing a system solely at the application or operating system level has proven insufficient, as these layers remain inherently susceptible to high-level exploits and persistent threats. To mitigate these risks, the Trusted Computing Group (TCG) established standardized protocols for hardware-based security, most notably through the Trusted Platform Module (TPM). While the TPM has been instrumental in securing personal computers and enterprise servers, its complexity, cost, and resource requirements have limited its efficacy in the rapidly expanding domain of the Internet of Things (IoT).

The evolution of the IoT has introduced a massive ecosystem of autonomous, interconnected devices that operate with minimal human intervention. These devices often function under strict power, footprint, and computational constraints, making the integration of a full-scale TPM impractical. Although alternative technologies such as Arm TrustZone and Caliptra have attempted to address these niche constraints, there remains a need for a universal, lightweight, and vendor-neutral specification. In response, the TCG introduced the Measurement and Attestation Root of Trust (MARS). MARS is designed as a minimalist hardware primitive, providing the essential services of measurement and attestation without the cryptographic overhead of traditional security chips.

Despite its architectural promise, the widespread adoption of MARS is currently hindered by a critical bootstrapping challenge: commercially available physical hardware does not yet exist, and existing reference software implementations lack the system-level fidelity required for robust development and testing. This creates a significant barrier for developers looking to integrate MARS into modern software stacks ahead of hardware availability.

To address this deficiency, this thesis—developed within the TORSEC research group—presents the design and implementation of an architecturally faithful MARS emulator tailored for the Linux operating system. The core of this research involves the creation of a sophisticated, multi-layered abstraction stack positioned between the client application and the emulated security primitive. By bridging user and kernel space to virtualize the MARS interface and enforce hardware-like access controls, this emulator provides a transparent environment where applications can interact with security services exactly as they would with physical hardware. This work not only facilitates the immediate prototyping of secure IoT applications, but also serves as a critical stepping stone toward the widespread industrial adoption of the MARS specification.

Chapter 2

Trusted Computing

2.1 Trusted Computing Concepts

The concept of having secure systems at the hardware level began in the 1960s. Nowadays, this has evolved into Trusted Computing (TC), following the definitions and open standards provided by the Trusted Computing Group (TCG) in the 1990s [1]. To understand how these platforms operate, it is necessary to define three core concepts:

- Trusted: A component is considered trusted if the system relies on it to behave in a specific way. This means that if a trusted component fails or is compromised, the security of the entire system is at risk.
- Trustworthy: A component is trustworthy only when it provides clear, verifiable evidence that it behaves exactly as expected. The goal of Trusted Computing is to prove that the parts we trust are actually trustworthy.
- Trusted Platform: A trusted platform is a computing system that has a verifiable identity and can prove its current state. It provides a predictable and safe foundation for all other operations.

TCG's foundational approach centers on the Trusted Computing Base (TCB), which is a minimized and protected combination of hardware, firmware, and software. By securing this critical set of components, a platform can offer strong, verifiable guarantees of confidentiality, integrity, and availability. This assurance allows the platform to act as a "source of truth" that does not rely on potentially vulnerable operating systems or applications.

By embedding these trust mechanisms directly into the hardware, Trusted Computing is able to address modern security needs. It enables verifiable trust relationships across devices, networks, and services, ensuring that the system's integrity is maintained from the moment it is powered on.

2.2 Root of Trust (RoT)

The concept of a Root of Trust (RoT) is foundational to Trusted Computing, as defined by the Trusted Computing Group (TCG). The TCG explicitly defines a RoT as a *"system element that must be trusted because misbehavior is not detectable"* [2].

This definition highlights that RoTs are critical hardware or firmware components whose inherent integrity is assumed, as their proper behaviour cannot be verified by other means. They serve as the secure foundation upon which a secure environment, known as a Trusted Platform, is built.

To establish such a Trusted Platform, the TCG mandates the presence of three specific and interdependent types of Root of Trust:

- Root of Trust for Storage (RTS)
- Root of Trust for Measurement (RTM)
- Root of Trust for Reporting (RTR)

Root of Trust for Storage

The RTS defines a memory location that is shielded from access by an unauthorized entity. Because of this characteristic, it provides a secure storage for any information that is needed to be stored. The information stored in the RTS can be distinguished as disclosable information and confidential information. Information such as measurements log or configuration digests which will be used for reporting by the RTR, fall into the disclosable category. While content like private part of an asymmetric key and a secret key for a symmetric algorithm fall into the second category, with the RTS providing confidentiality of such information.

Root of Trust for Measurement

The RTM sends integrity-relevant informations, called measurements, to the RTS. To create those measurements, the RTM computes cryptographic hash of the software components in the boot sequence, such as the bootloader, operating system components, and configuration data. On computer systems, the RTM is typically the CPU that's being controlled by the Core Root of Trust for Measurements (CRTM). The CRTM is the first set of instructions that is being executed to establish a new chain of trust. During the reset of the system, the CRTM sends information about its identity to the RTS.

Root of Trust for Reporting

The RTR provides verifiable evidence about the state of the RTS. It typically generates a signed digest of the RTS content using a private key stored securely within it. Only information that can be safely disclosed is reported. E.g. in a TPM the keys stored within it are not included in the reports, while informations such as Platform Configuration Register (PCR), audit logs, and key properties are.

2.2.1 Trusted Platform Module (TPM)

The concept of the Trusted Platform Module (TPM) emerged in the late 1990s as the industry recognized that software-based security was insufficient against low-level system subversion. To address this, the Trusted Computing Group (TCG) was formed, releasing the first major specification, TPM 1.1b, in 2001, followed by the widely adopted TPM 1.2. As noted in [3], the TPM was designed to provide a "Hardware Root of Trust," ensuring that security identifiers and cryptographic keys remain isolated from the host operating system.

A TPM is a dedicated system component designed to operate independently from the host system, maintaining its own internal state that cannot be directly modified by external software. Interaction with the TPM is restricted by the use of the standardized interfaces defined in the specification [2], ensuring consistency and security across implementations. During the construction of a TPM, its resources may be permanently assigned to it or temporarily assigned, which could then be located within a single physical boundary or distributed across multiple ones [4]. The implementation may be done in the form of a discrete single-chip attached to the system (dTPM) or as firmware (fTPM) executed on the host processor while in a special execution mode. In both cases, the only way to interact with the TPM is via the specified interface [4].

Key Features and Version Evolution

The original TPM 1.2 specification introduced several critical security advantages to a device, establishing a hardware root of trust that moves beyond simple software controls [2, 3]:

- **Identification of devices:** Prior to the release of the TPM specification, devices were mostly identified by MAC addresses or IP addresses-not security identifiers.
- **Secure generation of keys:** Having a hardware random-number generator is a big advantage when creating keys. A number of security solutions have been broken due to poor key generation.
- **Secure storage of keys:** Keeping good keys secure, particularly from software attacks, is a big advantage that the TPM design brings to a device.
- **NVRAM storage:** Having NVRAM allows a TPM to maintain a certificate store or persistent data that remains even if the host hard drive is wiped.
- **Device health attestation:** Unlike software-only attestation, which a compromised system can spoof, the TPM verifies system integrity independently of the host environment.

The subsequent TPM 2.0 implementations enable the same features as 1.2, plus several significant advancements [3, 2]:

- **Algorithm agility:** Algorithms can be changed without revisiting the specification, should they prove to be cryptographically weaker than expected (e.g., moving from SHA-1 to SHA-256).
- **Enhanced authorization:** This unifies how all entities are authorized, extending the ability to enable policies for multi-factor and multi-user authentication.
- **Quick key loading:** Keys can now be loaded quickly using symmetric rather than asymmetric encryption, addressing the performance bottlenecks of earlier versions.
- **Non-brittle PCRs:** In the past, locking keys to device states caused management problems during authorized updates. TPM 2.0 resolves these complexities.
- **Flexible management:** Different kinds of authorization can be separated, allowing for more granular management of TPM resources.
- **Identifying resources by name:** Fixes architectural weaknesses by using cryptographically secure names for resource identification instead of indirect references.

Integrity Measurement via PCRs

Platform Configuration Registers (PCRs) are fundamental to both versions, acting as unique, tamper-evident memory locations used to cryptographically attest to the integrity of the host platform's configuration state [4]. A PCR stores a cryptographic hash that is incrementally extended by system components:

$$PCR_{new} = \text{HASH}(PCR_{old} \,\|\, \text{Measurement}) \tag{2.1}$$

This integrity record is fundamental for features like Measured Boot and is used to seal cryptographic keys, ensuring that sensitive data is only accessible when the platform is verified to be in an authorized, trusted state [4].

2.2.2 Trusted Execution Environment (TEE)

The conceptual foundation of a Trusted Execution Environment (TEE) is rooted in the principle of Privilege Separation and the minimization of the Trusted Computing Base (TCB). Unlike traditional security models that rely on the integrity of the Operating System (OS) kernel, a TEE establishes a hardware-enforced perimeter that excludes the Rich Execution Environment (REE) from the TCB. By reducing the volume of code that must be inherently "trusted," the TEE minimizes the system's attack surface. This isolation ensures that even in the event of a total compromise of the privileged REE software stack, the confidentiality and integrity of the assets within the TEE remain preserved through hardware-level access control and memory protection.

Beyond mere isolation, a TEE is defined by its ability to provide measurable security guarantees, most notably through Remote Attestation and Sealed Storage. Remote attestation allows the TEE to generate a cryptographically signed "quote" or report that proves the identity and integrity of the software running within it to an external challenger. This process relies on a hardware-anchored Root of Trust (RoT), typically a unique cryptographic key fused into the SoC during manufacturing. Furthermore, sealed storage ensures that sensitive data is bound to the TEE's identity, preventing unauthorized access even if the data is extracted from the physical storage medium. These properties transform the TEE from a passive isolation chamber into a verifiable platform for confidential computing.

In practice, TEE architectures have diverged into two primary implementation paradigms: the Split-World approach and the Enclave-based approach. The Split-World paradigm, exemplified by ARM TrustZone, logically partitions the processor into a "Normal World" and a "Secure World" via a hardware-level execution mode. This allows the TEE to manage secure peripherals directly but requires a dedicated (and often complex) Trusted OS. Conversely, the Enclave paradigm, pioneered by Intel SGX [5], allows for the creation of isolated execution contexts within a standard application's address space. In this model, the hardware encrypts specific memory regions (Enclaves), protecting them from even the most privileged system software. While the former offers broader control over hardware resources, the latter provides a more granular approach to protecting specific computational workloads in untrusted environments, such as public clouds.

2.3 Hardware RoT for IoT

In the context of Internet of Things (IoT) systems, where devices often operate under strict constraints such as limited computational power, restricted memory, and low energy availability, the design of Roots of Trust (RoT) must account for these limitations. Unlike general-purpose computing platforms, IoT devices cannot always accommodate resource-intensive security modules.

To bridge these constraints with security requirements, modern IoT RoTs focus on two primary boot strategies: **Secure Boot** and **Measured (Trusted) Boot**. Secure Boot is a restrictive mechanism that ensures only cryptographically signed, authorized code can execute, usually by halting the boot process if a signature check fails. In contrast, Measured Boot (often referred to as Trusted Boot) does not stop the boot process; instead, it records (measures) each stage of the software stack into hardware registers. These measurements form an evidence chain that can be used for remote attestation, allowing a third party to verify the device's state post-boot. The following solutions represent the industry's shift toward lightweight, verifiable, and often open-source implementations of these concepts.

2.3.1 Device Identifier Composition Engine (DICE)

The Device Identifier Composition Engine (DICE) is a lightweight, standards-based security architecture designed by the TCG specifically for scenarios where a full TPM is physically or economically unfeasible. Its primary design goal is to provide a unique, non-spoofable identity and a verifiable attestation certificate for even the simplest microcontrollers.

This mechanism fundamentally relies on a permanent, high-entropy Unique Device Secret (UDS) inherent to the silicon. DICE utilizes this UDS to recursively derive a Compound Device

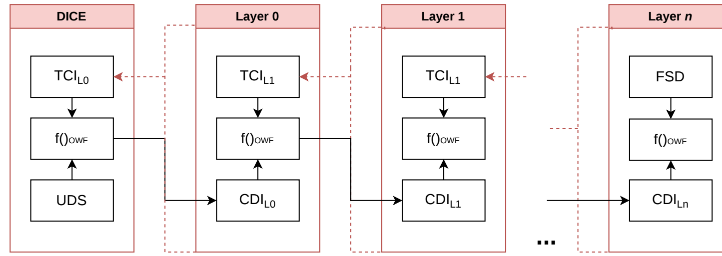

Figure 2.1. DICE: TCB Layering architecture [6]

Identifier (CDI). As shown in Figure 2.1, each loaded component is hashed to produce a TCB Component Identifier (TCI). This measurement is then cryptographically bound to the previous layer's secret.

Attestation Capabilities: Beyond identity, the DICE specification defines robust attestation capabilities. The resulting CDI is used to derive an Alias Key Pair and a corresponding X.509 certificate. Because the certificate includes the measurements of the software stack, it serves as a cryptographic proof of the device's "health." If the firmware is tampered with, the derived CDI changes, the previous keys become inaccessible, and the attestation fails, providing a seamless link between hardware identity and software integrity.

2.3.2 ARM TrustZone

While DICE focuses on identity and boot-time measurements, ARM TrustZone addresses **run-time isolation**. It is a hardware-enforced architecture that logically divides a system-on-chip (SoC) into two execution environments: the Secure World and the Non-Secure World.

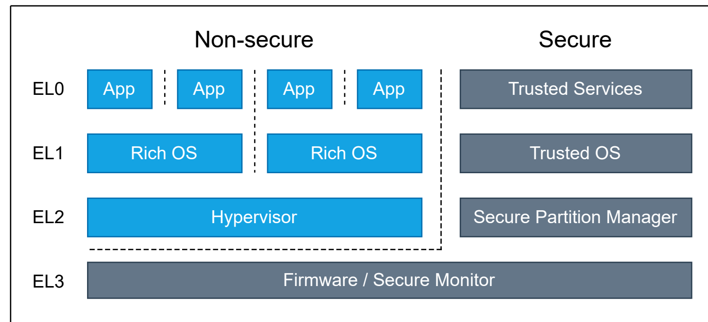

Figure 2.2. ARM Trustzone Architecture [7]

In the AArch64 architecture, this division is managed via Exception Levels (ELs). As illustrated in Figure 2.2, EL3 hosts the Secure Monitor, which handles context switching. This isolation allows developers to run a "Trusted Execution Environment" (TEE) alongside a standard OS. TrustZone acts as an RoT for isolation, ensuring that even if the main OS is compromised, the secrets held within the Secure World (such as private keys or biometric data) remain inaccessible to the attacker.

2.3.3 Caliptra

Caliptra represents a shift toward **standardized, transparent silicon RoT**. Unlike proprietary security blocks, Caliptra was designed by a consortium (including Google and Microsoft) to provide a "Macro-IP" block that is identical across different chips (CPUs, GPUs, and SoCs). The

design goal is to eliminate the "black box" nature of hardware security, providing a consistent Root of Trust for Measurement (RTM) and Identity (RTI).

Architecturally, Caliptra integrates a dedicated RISC-V core and cryptographic accelerators. Crucially, Caliptra **implements the DICE standard** (Section 2.3.1) as its foundational security model. It starts from an immutable ROM and measures the First Mutable Code (FMC), extending the chain of trust through hardware-managed Platform Configuration Registers (PCRs). By using an open-source, standardized block, cloud and IoT providers can verify the security logic of their silicon regardless of the vendor.

2.3.4 OpenTitan

Similar to Caliptra, OpenTitan [8] is an open-source RoT project, but it focuses on providing a **completely transparent silicon design**-down to the logical gates. Its primary purpose is to solve the "supply chain trust" problem by allowing anyone to inspect the hardware RTL (Register Transfer Level).

OpenTitan is highly modular, offering different configurations based on the integration needs:

- **Earl Grey:** A standalone, discrete RoT (similar to a dedicated TPM chip) optimized for low-power IoT applications.
- **Darjeeling:** An integrated RoT designed to be a "chiplet" or a block within a larger SoC, common in data centers.

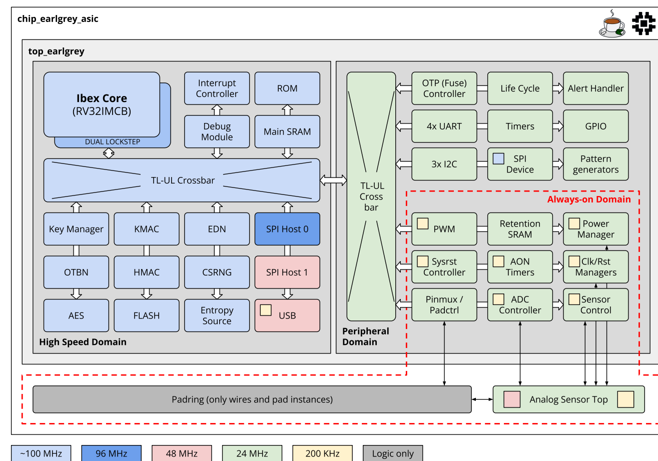

Figure 2.3. Top-Earlgrey Datasheet

A key design innovation in OpenTitan is the **OpenTitan Big Number Accelerator (OTBN)**. While many IoT RoTs are limited to standard RSA/ECC, the OTBN provides the flexibility to support **Post-Quantum Cryptography (PQC)**. This ensures that IoT devices deployed today, which may stay in the field for decades, can be updated to remain secure against future quantum computing threats.

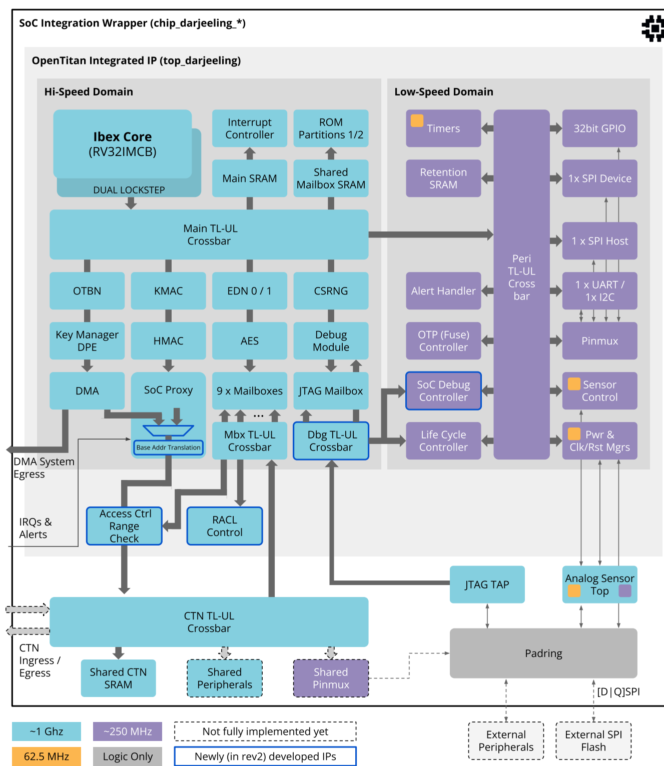

Figure 2.4. Top-Darjeeling Datasheet

Chapter 3

Secure Device Software Implementation

An alternative approach to implementing secure devices, such as the Trusted Platform Module (TPM), is through software emulation. While software-based implementations may lack the physical tamper-resistance of hardware-based solutions, they offer significant advantages in flexibility. This is particularly beneficial for testing, debugging, and understanding the internal mechanics of the device without the constraints of physical hardware.

3.1 TCG TPM Software Implementations

The Trusted Computing Group (TCG) TPM is one of the most prominent secure devices available. While hardware implementations are standard in commercial devices, robust software implementations have been developed to facilitate virtualization and development. The most notable software implementations are provided by IBM and Microsoft.

3.1.1 IBM Software TPM

The primary implementation discussed in this chapter is the suite provided by IBM, principally developed by Stefan Berger. This implementation is designed with a modular architecture, ensuring seamless interaction between applications and the virtual TPM device. The architecture relies on two main components: the `libtpms` [9] library and the `swtpm` [10] application. These tools can be integrated with a Virtual Machine Manager (VMM), such as QEMU, to provision a TPM for a virtual machine or to expose a TPM interface for remote access.

libtpms

The `libtpms` library serves as the core of this implementation. It is designed as a standalone library that can be linked by any application requiring TPM functionality. It implements all specifications mandated by the TCG [4], supporting both TPM 1.2 and TPM 2.0 standards. Essentially, it acts as the functional backend for the `swtpm` software implementation.

swtpm

The `swtpm` executable integrates the `libtpms` library to expose a TPM to the system. Primarily designed for GNU/Linux environments, this utility offers several key capabilities:

- **Device Initialization and Setup**

- **Socket-based Emulation (Network)**
- **Internal Integration (QEMU)**
- **Host Device Emulation**

Device Initialization and Setup: Before a software TPM can be used, it must be "manufactured" just like a physical chip. `swtpm` includes a utility called `swtpm_setup` [11], which generates the necessary cryptographic seeds, the Endorsement Key (EK), and the Storage Root Key (SRK). It also simulates a Certificate Authority (CA) to sign the EK certificate, ensuring the virtual device appears valid to the operating system.

Socket-based Emulation (Network): In this mode, `swtpm` listens on a TCP socket, allowing applications or even remote machines to send TPM commands over the network. This is particularly useful for containerized environments or distributed testing, where the application requiring the TPM is not running on the same machine as the emulator.

Internal Integration (QEMU): For virtual machine integration, `swtpm` uses a specialized communication protocol over Unix domain sockets. Unlike the standard network mode, this interface includes a "Control Channel." This side channel allows the hypervisor (QEMU) to perform administrative tasks that a standard user cannot do, such as freezing the TPM state during a VM snapshot or migrating the TPM state to a different physical host without losing data.

Host Device Emulation: To provide a TPM device directly to the host system, the author developed a kernel driver named `vtpm_proxy` [12]. This driver allows `swtpm` to request the kernel to instantiate a new TPM character device (e.g., `/dev/tpm1`). Once created, the driver bridges the communication between the user-space `swtpm` process and the newly created kernel device, allowing standard host tools to use the software TPM transparently.

QEMU TPM Integration

QEMU is a widely adopted open-source emulator and virtualizer, supported by all major operating systems. Historically, QEMU lacked native support for TPM devices, preventing virtual machines from utilizing security features that rely on hardware roots of trust.

This gap was bridged by Berger's integration [13], which allows QEMU to interface with external TPM emulators like `swtpm`. By establishing a communication channel (typically via a Unix domain socket) between the QEMU process and the TPM emulator, QEMU can present a fully functional TPM device to the guest operating system. From the perspective of the virtual machine, this emulated device is indistinguishable from a physical TPM chip, enabling the testing of complex security stacks and operating system features in a virtualized environment.

3.1.2 Microsoft TPM 2.0 Reference Implementation

Microsoft provides the official reference implementation for the TPM 2.0 specification, often referred to as `ms-tpm-20-ref` [14]. Unlike the IBM implementation which focuses on virtualization tools, the Microsoft implementation focuses on specification compliance and is widely used as the basis for both testing tools and firmware-based TPMs (fTPM).

Simulator

The Microsoft TPM 2.0 Simulator is a user-space application built directly from the reference code. It is designed primarily for development and testing purposes, allowing developers to write TPM-aware applications without requiring physical hardware.

The simulator listens on a TCP socket (typically ports 2321 for commands and 2322 for platform signals) and processes TPM commands exactly as a hardware device would. However, it explicitly lacks security features:

- **State Storage:** The NVRAM state is stored in a local file (e.g., `NVChip`), which is not encrypted or protected against the host OS.
- **Reset Capability:** The simulator can be easily reset or re-manufactured by deleting the state file, a feature impossible on physical hardware.

TEE Trusted Applications

While the simulator is insecure by design, the Microsoft reference code is also used to create secure "Firmware TPMs" (fTPMs). In this configuration, the software implementation runs as a **Trusted Application (TA)** inside a Trusted Execution Environment (TEE), such as ARM TrustZone.

By running inside the TEE, the software TPM achieves a level of security comparable to discrete hardware:

- **Isolation:** The TEE ensures that the main operating system (the "Rich Execution Environment") cannot access the TPM's memory or keys, even if the OS is compromised.
- **Secure Storage:** The TA uses the TEE's secure storage capabilities to protect the non-volatile state of the TPM.

This approach allows devices like smartphones and IoT endpoints to possess full TPM 2.0 functionality without the cost or board space required for a dedicated discrete chip.

3.2 TEE Software Implementations

Just as software TPMs allow for the development of secure applications without physical security chips, software-based Trusted Execution Environments (TEEs) enable the development of Trusted Applications (TAs) without specialized processor extensions like ARM TrustZone or Intel SGX. These environments are critical for prototyping and education.

3.2.1 Open-TEE

One of the most significant contributions to this field is **Open-TEE** [15], an open-source virtual TEE originally developed by the Intel Collaborative Research Institute for Secure Computing. It is designed to be fully compliant with the GlobalPlatform TEE Client API and TEE Internal Core API specifications.

Unlike hardware-backed TEEs that rely on processor isolation, Open-TEE runs entirely in user space on a standard operating system. Its architecture is composed of two primary environments:

- **The Rich Execution Environment (REE):** This side hosts the "Client Applications" (CAs). It includes a client library (`libtee`) that implements the GlobalPlatform Client API, allowing standard applications to initiate sessions with trusted applications.

- **The Trusted Execution Environment (TEE):** This side is emulated by a manager process. It includes the `tee_launcher`, which is responsible for loading Trusted Applications, and a scheduler that manages the execution of these TAs.

The communication between the REE and the emulated TEE is handled via shared memory and Unix domain sockets, simulating the secure monitor calls (SMC) that would occur in a physical implementation. This architecture allows developers to compile and debug Trusted Applications using standard tools (like GDB) before deploying them to actual hardware environments such as OP-TEE on ARM.

Chapter 4

Measurement and Attestation RootS (MARS)

Measurement and Attestation RootS (MARS) is a set of specifications developed by the Trusted Computing Group (TCG) designed to provide a lightweight, flexible Root of Trust (RoT) for embedded and resource-constrained devices. Unlike the Trusted Platform Module (TPM), which is often a discrete chip with a complex feature set, MARS is architected to be integrated directly into a host microcontroller or System-on-Chip (SoC) as a hardware block or a protected firmware environment.

Its primary design goal is to offer the minimal essential capabilities required to support *Measured Boot* and *Remote Attestation*. By standardizing the interface and the cryptographic operations (hashing, signing, and key derivation), MARS enables a diverse range of IoT devices to participate in standardized trust infrastructures without the overhead of a full TPM implementation.

4.1 Architecture

The MARS architecture is defined by its simplicity and modularity. It does not strictly mandate a specific hardware implementation but rather defines a functional model that a compliant device must meet.

At its core, a MARS device requires only a **Symmetric Cryptographic Engine**. While more advanced implementations may include a dedicated Asymmetric Cryptographic Engine (e.g., for ECDSA), a minimal MARS device can perform all required Attestation and Key Derivation operations using only the Symmetric Cryptographic Engine, e.g. MAC or AEAD tag.

The specification defines two distinct architectural models for integrating MARS into a host platform: the *Memory Mapped Architecture* and the *Serialized Architecture*.

4.1.1 Serialized Architecture

The **Serialized Architecture** is used when the MARS device is externally attached (e.g., a discrete chip connected via SPI or I2C) or isolated in a way that prevents direct memory access (e.g., running in a separate Trusted Execution Environment). This is the architecture depicted in Figure 4.1.

In this model, the Host and MARS do not share a memory space. Instead, they communicate via a strictly defined command protocol:

1. **Command Marshaling:** The Host API layer serializes the command ID and all parameters into a contiguous byte stream (a Command Block).

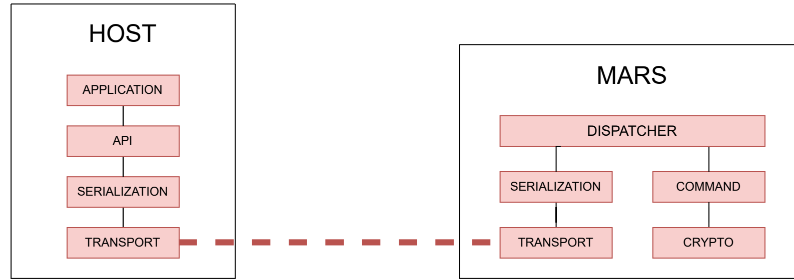

Figure 4.1. MARS Serialized Architecture [16]

2. **Transport:** This block is transmitted over the physical bus (SPI, I2C, etc.) to the MARS device.

3. **Dispatcher:** A component inside the MARS device, called the *Dispatcher*, receives the stream, parses (un-marshals) the command, and routes it to the appropriate internal function (e.g., PCR extension or Signing).

4. **Response:** The result is serialized back into a Response Block and sent to the Host.

This architecture allows MARS to be platform-agnostic, as the underlying transport layer can be changed without altering the core MARS logic.

4.1.2 Memory Mapped Architecture

In the **Memory Mapped Architecture**, the MARS device is typically attached internally (e.g., within the same Host microcontroller die or SoC). The Host interacts with the MARS functionality directly through Memory-Mapped Input/Output (MMIO) registers.

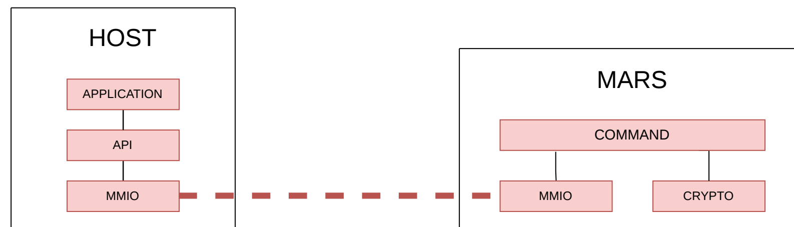

Figure 4.2. MARS Memory Mapped Architecture [17]

- **Direct Access:** The Host writes commands and data directly to the MARS hardware registers mapped into the Host's address space.
- **Efficiency:** Large parameters (such as data to be measured) can be streamed directly to the MARS registers without the need for packetization.
- **Simplicity:** This model eliminates the need for a complex command dispatcher or transport protocol, as the "API" is effectively the register map itself.

4.1.3 Cryptographic Support Functions

To operate independently of the host's processing environment, the MARS architecture defines a specific set of internal cryptographic support functions. These low-level primitives abstract the underlying cryptographic engine and provide the essential operations required for measurement, key derivation, and attestation.

A compliant MARS implementation relies on the following core functions:

- `CryptSelfTest(fullTest)`: Executes all or a subset of the internal cryptographic self-tests depending on whether `fullTest` is true or false. It returns false if any test fails, ensuring the cryptographic engine is operating correctly.
- `CryptHashInit(shc)`: Initializes the provided hash context (`shc`) to begin a new hashing operation.
- `CryptHashUpdate(shc, data, len)`: Updates the hash in the `shc` context with the provided `data` of a specific length (`len`).
- `CryptHashFinal(shc, out)`: Finalizes the hashing operation and returns the resulting digest into the `out` buffer.
- `CryptSign(key, digest)`: Digitally signs the provided `digest` using the specified `key`.
- `CryptVerify(key, digest, signature)`: Verifies the `signature` of a given `digest` using the corresponding `key`.
- `CryptSkdf(child, parent, label, ctx, ctxlen)`: Derives a symmetric `child` key from a `parent` secret, a specific `label`, and an application-provided context (`ctx`). This is utilized for Derivation Parent (DP) establishment and extension, as well as deriving bytes for external use. It serves as the fallback for `CryptXkdf` if asymmetric derivation is not supported.
- `CryptAkdf(child, parent, label, ctx, ctxlen)`: Derives an asymmetric `child` key pair from a `parent` secret, `label`, and application-provided context.
- `CryptXkdf(...)`: Maps to `CryptAkdf` (if asymmetric cryptography is supported by the hardware) or `CryptSkdf`. It is the primary function used to generate the keys required for quoting, signing, and signature verification.
- `CryptDpInit()`: Initializes and sets the value of the Derivation Parent (DP) using the primary hardware-bound secret, the Primary Seed (PS).
- `CryptSnapshot(regSelect, ctx, ctxlen)`: Computes a cumulative digest of the contents of all registers specified by the `regSelect` mask, binding it with the application context `ctx`. This operation is critical for securely sampling Trusted Sensor Registers (TSRs) and PCRs during attestation.

4.2 Root of Trust for Storage

MARS implementation of Root of Trust for Storage are quite different compared to how devices like TPM may implement them. Where as the TPM implement both a Shielded location for storage of important information and PCRs to store disclosable informations like measurements [4], MARS only implements PCRs and TSR providing only storage for disclosable informations. To make this possible, the device implements a way to recompute each key on demand, explained in more details in section 4.3.1

4.2.1 Platform Configuration Registers (PCRs)

Platform Configurations Registers are the main components used by MARS to securely store events that occurs on a system. Their value can only be updated with the use of a command called MARS_PcrExtend, which will be used to compute the cumulative digest of the content stored on the Register and the `new value` that needs to be stored, equation 4.1. Per specification, a MARS device is required to have at least one PCR, known as PCR 0, but may provide additional PCRs to record some subset of events [18]. After an event occurs and its log is recorded on a PCR, it is now possible to use that PCR for the validation of correctness for that event.

$$PCR_{new} = HASH(PCR_{old}||Measure) \tag{4.1}$$

4.2.2 Trusted Sensor Registers (TSRs)

One of the characteristics that make MARS different from many of the other alternatives, is the addition of having Trusted Sensor Registers (TSR). Thanks to these registers, MARS is able to provide a way to bind live sensor data-such as from an onboard clock, temperature probe, or pressure sensor-into the cryptographic trust framework. The main characteristic of TSRs, is that they are not extendable, instead their values is written by sampling a sensor that is linked to it. This read is executed by a command that require a `regSelect` value and then stored inside the register. The logic to perform a sample, is performed when the support function `CryptSnapshot()` is called, which creates a cumulative digest based on the values of `regSelect` and the registers that it asks. The sampled value can then be read by using the command `MARS_RegRead`, which is the only that uses `regSelect` to return the value read. Because of how they are used, the main purpose of TSRs is to sign sensor values via MARS' quoting ability [18].

4.3 Root of Trust for Reporting

The Root of Trust for Reporting (RTR) is responsible for creating a cryptographically verifiable report of the device's state. To convey the history of the transitive trust chain, the RTR is used to digitally sign the contents of the PCRs (and TSRs). This signature is used by the host device to form an attestation that is conveyed to a remote challenger.

The challenger can use this attestation to verify several critical properties:

1. The identity of the device and its Endorser.
2. The freshness of the attestation (preventing replay attacks).
3. The authenticity and values of the PCRs.
4. The integrity of the event log, if one is present.

The RTR conveys identity using either asymmetric or symmetric cryptography. In an asymmetric scheme, the RTR uses an Attestation Key (AK) certified by an Endorser (Attestation Certificate Authority). The public portion of this AK verifies signatures produced by the device. In a symmetric scheme, the AK is a shared secret between the RTR and the Endorser.

MARS implements the RTR functionality primarily through the `MARS_Quote` command. This command generates a signature over a "snapshot" of the selected registers (PCRs and TSRs) and a caller-provided nonce. This mechanism binds the current state of the device measurements to a specific point in time and a specific requestor, ensuring that the report is both current and authentic.

4.3.1 Key Hierarchy

Compared to most of the alternative RoTs, where the hierarchy is managed by generating an asymmetric key, MARS manages everything by having a single value assigned at the creation of the device, called **Primary Seed (PS)**.

The Primary Seed is at the base of the Key Hierarchy. Its main purpose is to create the **Derivation Parent (DP)**. With the DP, the MARS device is able to then create any of the keys that will be used, or even recreate the Derivation Parent in case it is needed.

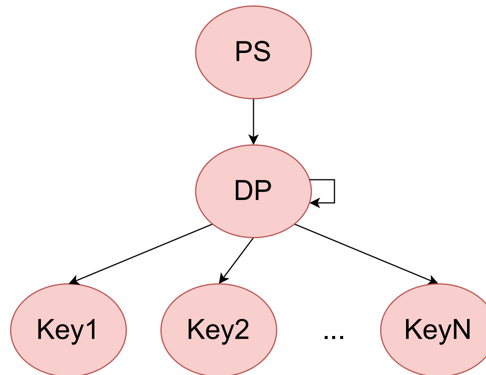

Figure 4.3. Key Hierarchy [18]

As mentioned in 4.2, where MARS implementation of RTS was explained, the keys are never stored but instead are generated every time there is a need to use them, which simplify by a lot the structure of the device.

Each key can be generated by adding a certain value, that specifies how the key will be used after its generation. These purposes are defined by specific labels used during the Key Derivation Function (KDF):

- **'X' External**, Used as a key outside the device
- **'D' Derivation Parent**, Replaces the current DP
- **'U' Unrestricted signing**, Signs external data
- **'R' Restricted attestation**, Internal use for attestation

4.4 MARS API

The MARS API provides a standard set of commands that the Host uses to interact with the RoT. These commands abstract the underlying cryptographic hardware, allowing the Host to perform measurements, manage keys, and request attestation without direct access to the Primary Seed.

Based on the MARS specification [18, 17], the API is structured into mandatory baseline commands, extended commands for a fully featured implementation, and necessary host-side state management controls.

4.4.1 Mandatory Commands

A compliant MARS implementation must support the following core commands to ensure basic measurement and reporting capabilities:

- `MARS_CapabilityGet(pt, cap, caplen)`: Retrieves the capabilities and properties of the MARS device, such as supported cryptographic algorithms or the number of available registers.
- `MARS_PcrExtend(pcrIndex, dig)`: Extends the indexed PCR with a new measurement digest. This is the foundational mechanism for Measured Boot.
- `MARS_RegRead(regIndex, dig)`: Reads the current digest value of a specific PCR or TSR. Note that for TSRs, this returns the last sampled value but does not trigger a new sensor sample.
- `MARS_Quote(regSelect, nonce, nlen, ctx, ctxlen, sig)`: Generates a digital signature over a snapshot of the selected registers (defined by the `regSelect` bitmask) and a caller-provided nonce. This serves as the primary command for Remote Attestation.
- `MARS_PublicRead(restricted, ctx, ctxlen, pub)`: Retrieves the public portion of an asymmetric key derived by the device. *Note: This command is mandatory if the MARS implementation provides asymmetric cryptographic functions.*

4.4.2 Extended Commands

To support advanced cryptographic operations, multi-part hashing, and external key management, a fully featured MARS device implements the following extended commands. While most are required for extended profiles, specific commands may remain optional depending on the implementation footprint:

- `MARS_SelfTest(fullTest)`: Requests the MARS device to perform its internal cryptographic self-tests.
- **Sequence Hashing**: A set of commands (`MARS_SequenceHash()`, `MARS_SequenceUpdate(...)`, and `MARS_SequenceComplete(...)`) used to perform a multi-part hashing operation. This is necessary when the data to be measured exceeds the standard transport buffer size.
- `MARS_Derive(regSelect, ctx, ctxlen, out)`: Derives a new key or secret from the current Derivation Parent (DP), dynamically bound to the selected registers and application context.
- `MARS_Sign(ctx, ctxlen, dig, sig)`: Signs an external digest using an "Unrestricted" derived key. This allows the MARS device to act as a general-purpose signing oracle for the host system.
- `MARS_SignatureVerify(restricted, ctx, ctxlen, dig, sig, result)`: Verifies a provided signature against a digest using a restricted or unrestricted key derived internally by the device.
- `MARS_DpDerive(regSelect, ctx, ctxlen)` **(Optional)**: Derives and updates the current Derivation Parent (DP) based on the specified registers and context. If omitted, the device relies on the default DP established at initialization.

4.4.3 Initialization and Concurrency Control

To facilitate safe and coordinated access to the MARS device, especially in multi-threaded host environments, the API provides specific functions for initialization and state management:

- `MARS_ApiInit()`: Sets the initial context required by the MARS API. This includes initializing underlying synchronization primitives (such as a mutex) and retrieving essential constants and capabilities directly from the MARS device to prepare it for operational use.

- **Concurrency Control**: Certain operational sequences require the MARS device to maintain a specific state across multiple command invocations (e.g., multi-part sequence hashing). To prevent race conditions, the API defines the following exclusive access controls:
  - `MARS_Lock()`: Grants exclusive access to the device for the calling thread. It returns `MARS_RC_SUCCESS` if exclusive access is successfully acquired, or `MARS_RC_LOCK` if the device is already locked.
  - `MARS_Unlock()`: Relinquishes the thread's exclusive access to the device. It returns `MARS_RC_SUCCESS` if the access is successfully returned, or `MARS_RC_LOCK` if the device was not currently locked.

4.5 Command Serialization

To facilitate integration into various bus architectures (such as SPI, I2C, or a software API layer), MARS utilizes a serialized command protocol. This protocol defines how commands and responses are marshaled into byte streams, ensuring interoperability between the Host driver and the MARS hardware.

In the serialized architecture, every interaction is encapsulated in a structure that typically includes a command header (containing a Command ID and payload size) followed by the command parameters. The MARS device processes this stream and returns a response structure containing a Response Code (RC), output size, and the requested data (e.g., a signature or key).

This serialization allows MARS to be implemented as a discrete hardware block or a firmware library without changing the logic of the Host application, provided the transport layer handles the movement of the serialized bytes. [16]

Chapter 5

System Design

A primary objective of this emulator is to provide a robust development platform, enabling developers to write and test software for the MARS device without requiring physical hardware. To facilitate this, the emulator requires a mechanism to seamlessly interact with host applications and vice versa. This chapter details the system design and architecture developed to achieve this bidirectional communication.

5.1 General Overview

The system design relies on a strict separation between the host application and the emulated device environment. This architecture allows an application running natively on the host machine to interact directly with the emulator.

To accomplish this, the emulator's architecture defines the following four primary components:

- vmars-driver, a kernel-space driver that manages and facilitates interactions between the application and the emulator.
- mars-api, a middleware library serving as the intermediate layer between the application and the emulated device.
- libmars, the core library responsible for implementing the emulator's instructions and cryptographic operations.
- mars-emulator, the software daemon responsible for managing the emulation lifecycle and dispatching commands.

5.2 Architecture

The MARS specification defines two potential architectures: serialized and memory-mapped. The serialized architecture is specifically designed for external MARS devices. Because it relies on a standardized communication interface, the serialized architecture is the most straightforward to emulate and is the primary focus of this thesis.

To simulate this structure, the system maps a series of software components and libraries to each part of the physical architecture (Figure 5.1). From the perspective of the host application, communication occurs directly with the `mars-api` library, which handles the necessary API calls to the device. The information within these API calls is then encoded using the CBOR format, as mandated by the MARS specification.

In a host environment running a Linux-based operating system, the transport layer is exposed to the application as a character device file (e.g., /dev/mars0). The vmars-driver manages this device file, routing the encoded information from the application to the MARS device emulator.

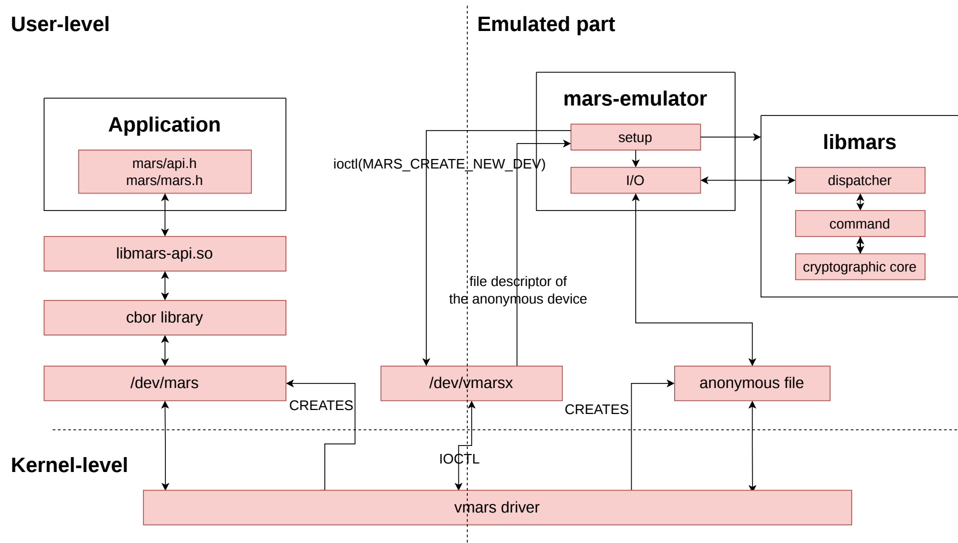

Figure 5.1. High-level block diagram of the MARS emulator software stack.

5.2.1 vmars-driver

This component is critical for enabling the emulator to run directly on the host machine. Its primary function is to handle the interaction between the application and the emulated device by providing a kernel-space interface.

During startup, the driver creates a control device node, `vmarsx`, which is responsible for establishing the interconnection layer. The emulator uses an `ioctl` system call on this device to notify the driver that a new emulator instance has started. In response, the driver provisions an `anonymous file` and a new device node named `mars`.

The anonymous file is returned to the emulator and serves as its interface for reading incoming application commands and writing responses. Concurrently, the `mars` device node acts as the entry point for the user-space application, allowing it to send commands for the emulator to execute and read the resulting output.

5.2.2 mars-api

The `mars-api` component is responsible for implementing the standard Application Programming Interface defined by the TCG MARS specification, ensuring that any host application can interract with any kind of MARS device.

To achieve this, the `mars-api` manages the transmission of the commands provided by the user application. When an application invokes a standard MARS function, the library asks the defined CBOR library to translate the request into the standardized payload required by the specification. It then transmits this payload directly to the character device node. Upon receiving a response from the device, the payload is decoded and the library returns the structured data back to the calling application.

Additionally, the library manages the lifecycle, state, and concurrency of MARS sessions. It implements the standard mechanisms for device access control, including:

- MARS_ApiInit, to initialize the library context and the required resources for establishing communication with the device node
- MARS_Lock, to request exclusive access to the MARS device, preventing race conditions or command interruptions from other processes during sensitive, multi-step cryptographic sequences

- MARS_Unlock, to release the exclusive lock, restoring shared access to the device.

5.2.3 libmars

The libmars library acts as the foundational state machine and execution engine for the MARS environment. Crucially, the architecture strictly isolates this library from the transport and I/O mechanisms handled by the mars-emulator daemon. Mirroring the design philosophy of software TPMs-specifically the separation between swtpm and libtpm-libmars is a highly portable, standalone component. By encapsulating the core MARS specification logic and command routing, it enables developers to directly integrate the library into custom emulation testbeds or alternative transport layers with minimal overhead.

To manage the complexity of the emulated device, the internal architecture of libmars is divided into functional domains:

Registers and Constants

This module functions as the virtual hardware's memory and configuration space, managing the internal state across the device's lifecycle. It is responsible for:

- Primary Seed (PS): Securely holding the fundamental hardware constant injected during the initialization phase.
- Derivation Parent (DP): Deriving and storing the runtime root key generated from the Primary Seed at startup.
- PCR and TSR Arrays: Maintaining the arrays of Platform Configuration Registers and TCI Status Registers required to securely record and report system state and Target Component Integrities.

Cryptographic Engine

The cryptographic domain is subdivided into a primitive layer and an abstraction layer, ensuring flexibility across different MARS profiles:

- Cryptographic Cores: This lower layer implements the raw mathematical primitives, providing unified symmetric core functions necessary for single-pass hashing, signing, and Key Derivation Functions (KDFs). It also houses the infrastructure for asymmetric algorithms (e.g., RSA, DSA) when required.
- Cryptographic Functions: Operating as a dynamic abstraction layer, this module exposes standardized internal interfaces (such as CryptHashInit and CryptHashUpdate). It selects the appropriate core algorithm dictated by the profile, ensuring that higher-level command logic remains completely decoupled from the underlying cryptography.

Command Execution and Internal Dispatcher

A defining feature of libmars is its self-contained command dispatcher. Rather than relying on an external daemon to parse and route individual requests, the library exposes a unified dispatch interface. This internal routing mechanism translates raw incoming payloads into the execution logic for the standard MARS command set (e.g., MARS_Quote, MARS_Sign), manipulates the internal state, and formats the appropriate response.

5.2.4 mars-emulator

The mars-emulator component is a user-space daemon that serves as the execution host and I/O manager for the emulated device. Because the core emulation logic and command routing are entirely encapsulated within the libmars library, the primary responsibility of this daemon is to manage the emulation lifecycle and facilitate bidirectional communication between the libmars execution engine and the vmars-driver.

Setup

This module governs the initialization phase of the emulator. Upon execution, the daemon interacts with the control device (/dev/vmarsx) managed by the kernel driver, requesting the creation of a new virtual MARS instance. In response, it receives an anonymous file descriptor, which it configures as its primary forward I/O interface.

Depending on the deployment state, the setup routine follows one of two paths:

- Provisioning a New Device: If initializing a fresh MARS instance, the daemon generates a new Primary Seed (PS) and provides it to libmars, which subsequently derives the runtime Derivation Parent (DP). The daemon also decodes the provided Profile list and passes it to libmars to instantiate the PCR and TSR arrays, select the appropriate cryptographic suite, and load the supported command set.
- Restoring an Existing Device: If a MARS device configuration already exists, the daemon parses the saved configuration file and initializes the libmars state utilizing the restored context.

Once the hardware state is successfully provisioned or restored, the daemon initiates the main event loop.

Input/Output Interface

To communicate with the host application, the daemon utilizes the anonymous file descriptor established during the setup phase. This module handles the low-level system calls required to read raw incoming command bytes from the kernel driver and write the resulting response bytes back to the same interface.

Dispatcher Invocation

As part of the daemon's main event loop, it continuously monitors the I/O interface for incoming activity. Rather than parsing or interpreting the received CBOR payloads itself, the daemon delegates this responsibility directly to the library.

When a command is read from the file descriptor, the daemon passes the raw buffer directly to the unified dispatch function exposed by libmars. The library internally parses the payload, executes the corresponding MARS command, updates its internal state, and returns a formatted response buffer. The mars-emulator daemon then simply takes this returned buffer and writes it back to the I/O interface, seamlessly completing the execution cycle and returning the result to the calling application via the anonymous device.

Chapter 6

Implementation

The implementation phase represents the core technical contribution of this thesis, translating the architectural design of the Measurement and Attestation Roots of Trust (MARS) into a functional emulator stack. This chapter provides an in-depth explanation of the implementation process, diving into the kernel-space driver, the user-space libraries, and the orchestration logic.

6.1 vmars-driver

The `vmars-driver` acts as the primary interface between the user-space applications utilizing the `mars-api` and the actual `mars-emulator` process. It is implemented as a Linux kernel module that creates virtual character devices to facilitate synchronized, cross-process communication.

6.1.1 Character Devices and Synchronization

To manage the lifecycle of a request and the internal state of the virtual device, the driver utilizes the `proxy_dev` structure.

```c
struct proxy_dev {
    struct cdev cdev;

    long state;
#define STATE_OPENED_FLAG        BIT(0)
#define STATE_WAIT_RESPONSE_FLAG BIT(1)
#define STATE_REGISTERED_FLAG    BIT(2)
#define STATE_DRIVER_FLAG        BIT(3)

    atomic_t device_already_open;

    struct mutex buf_mutex;

    wait_queue_head_t fops_wq;
    wait_queue_head_t device_wq;

    size_t request_length; // Amount of data given by the application to the emulator
    size_t response_length; // Amount of data given to the application by the emulator
    u8 buf[MARS_BUFSIZE];
};
```

Listing 6.1. Internal state structure of the vmars-driver

As shown in Listing 6.1, synchronization between the user-space application and the emulator is handled through a combination of mutexes and wait queues (`fops_wq` and `device_wq`).

When an application sends a command, the driver sets the `STATE_WAIT_RESPONSE_FLAG` and puts the calling process to sleep using the wait queues. The emulator, which continuously polls or blocks on the device, retrieves the request, processes it, and writes the response back to the `buf`.

Upon completion, the driver wakes the application, ensuring a synchronous request-response flow despite the decoupled, asynchronous nature of the emulator daemon.

Before this synchronized data exchange can occur, however, the internal `proxy_dev` state must be properly bootstrapped and assigned a file descriptor. This initialization is triggered by the emulator during its startup phase.

6.1.2 Virtual Device Instantiation and ioctl Handling

To allow the user-space `mars-emulator` daemon to create and attach to a virtual MARS device, the driver exposes a control node (typically `/dev/vmarsx`). This node accepts an `ioctl` command (`VMARS_PROXY_IOC_NEW_DEV`) which acts as the trigger for instantiating the virtual hardware state within the kernel.

The communication across the user-kernel boundary for this command is facilitated by the `vmars_proxy_ioc` structure.

```
struct vmars_proxy_ioc {
    uint32_t fd;    // output: File descriptor for the anonymous device
    uint32_t major; // output: Major device number
    uint32_t minor; // output: Minor device number
};
```

Listing 6.2. Structure used for ioctl communication

When the kernel receives the `ioctl` command, the standard kernel dispatcher routes the request to a dedicated handler function, `vmars_ioctl_new_dev`.

```
static long vmars_ioctl_new_dev(unsigned long arg) {
    struct vmars_proxy_ioc __user *vmars_proxy_ioc_p;
    struct vmars_proxy_ioc vmars_proxy_ioc;
    struct file *vmars_file;

    vmars_proxy_ioc_p = (void __user *) arg;

    vmars_file = vmars_proxy_create_device(&vmars_proxy_ioc);
    if(IS_ERR(vmars_file)) {
        return PTR_ERR(vmars_file);
    }

    if(copy_to_user(vmars_proxy_ioc_p, &vmars_proxy_ioc, sizeof(vmars_proxy_ioc))) {
        put_unused_fd(vmars_proxy_ioc.fd);
        fput(vmars_file);
        return -EFAULT;
    }

    pr_info(VMARS_MODULE ": successfully created device\n");

    fd_install(vmars_proxy_ioc.fd, vmars_file);
    return 0;
}
```

Listing 6.3. Kernel handler for creating a new MARS proxy device

As demonstrated in Listing 6.3, this handler performs several critical operations to safely bootstrap the anonymous device for the emulator:

1. **Dual-Device Creation**: It calls the internal `vmars_proxy_create_device` function. This function provisions **two** distinct endpoints linked by a single, shared `proxy_dev` state structure. First, it registers a public, named character device (e.g., `/dev/mars`) that client applications will use to submit commands. Second, it allocates an anonymous character device and reserves an unused file descriptor for it, which acts as the backend endpoint for the `mars-emulator` daemon.

2. **Error Handling and Cleanup**: If the subsequent data transfer to user space fails, the driver meticulously rolls back the allocation. It releases the file descriptor via `put_unused_fd` and decrements the file reference count via `fput` to prevent memory leaks in the kernel.

3. **File Descriptor Installation**: Finally, upon absolute success, `fd_install` is invoked. This binds the newly allocated file descriptor to the `file` structure of the anonymous device, effectively activating the backend communication channel for the `mars-emulator`'s execution loop.

6.1.3 Device Exclusivity and Access Control

To maintain the integrity of the Root of Trust and prevent interleaved commands from corrupting the internal state, the `vmars-driver` strictly enforces exclusive access to the frontend device node (`/dev/mars`).

This is achieved using the `device_already_open` atomic variable defined in the `proxy_dev` structure. When an application attempts to `open()` the frontend device, the driver performs an atomic test-and-set operation. If the device is already acquired by another process, the system call immediately returns a busy error (e.g., `-EBUSY`). The atomic variable is only cleared when the owning application successfully closes the file descriptor. This kernel-level mutual exclusion guarantees that no rogue application can bypass the API and disrupt an ongoing cryptographic session.

6.1.4 Read/Write Operations and Execution Flow

The core responsibility of the `vmars-driver` is to securely and synchronously shuttle serialized commands between the frontend application (`/dev/mars`) and the backend emulator daemon (the anonymous file descriptor). Because these are two distinct, asynchronous user-space processes, the driver acts as the synchronization arbiter using the shared `proxy_dev` structure.

The lifecycle of a single MARS command execution follows a strict, five-step sequence, as visualized in Figure 6.1:

1. **Command Submission (Frontend Write)**: The client application initiates the flow by calling `write()` on the frontend device. The `proxy_dev_write` handler locks the `buf_mutex`, copies the command payload from user space into the shared `proxy_dev->buf`, and records the payload size in `request_length`. It then unlocks the mutex and triggers the `wake_up_interruptible` function on the `fops_wq` wait queue to signal the emulator.

2. **Emulator Wakeup and Read (Backend Read)**: Prior to this, the emulator daemon is typically blocked inside `vmars_proxy_fops_read`, sleeping on the `fops_wq` wait queue via `wait_event_interruptible`. The wakeup signal resumes the emulator's execution. It locks the mutex, copies the command from `proxy_dev->buf` into its own memory space, and sets the `STATE_WAIT_RESPONSE_FLAG`. To prepare for the next phase, it zeroes the shared buffer and resets `request_length` to 0 before returning the data to the daemon.

3. **Application Blocking (Frontend Read)**: Expecting a result, the client application immediately calls `read()` on the frontend device. The `proxy_dev_read` handler encounters a `response_length` of 0. Consequently, it puts the application process to sleep on the `device_wq` wait queue, effectively yielding the CPU while the emulator processes the command in user space.

4. **Response Injection (Backend Write)**: Once the `libmars` execution loop finishes processing, the emulator writes the CBOR-encoded result back to the anonymous file descriptor. The `vmars_proxy_fops_write` handler verifies the `STATE_WAIT_RESPONSE_FLAG`, copies the data into the shared buffer, updates the `response_length`, and clears the waiting flag. It then wakes up the sleeping application by signaling the `device_wq` queue.

5. **Application Resumption (Frontend Read Completion)**: The wakeup signal rouses the sleeping client application. The frontend read handler re-acquires the mutex, safely copies the populated `proxy_dev->buf` back to the application's user-space memory, resets `response_length` to 0, and completes the system call.

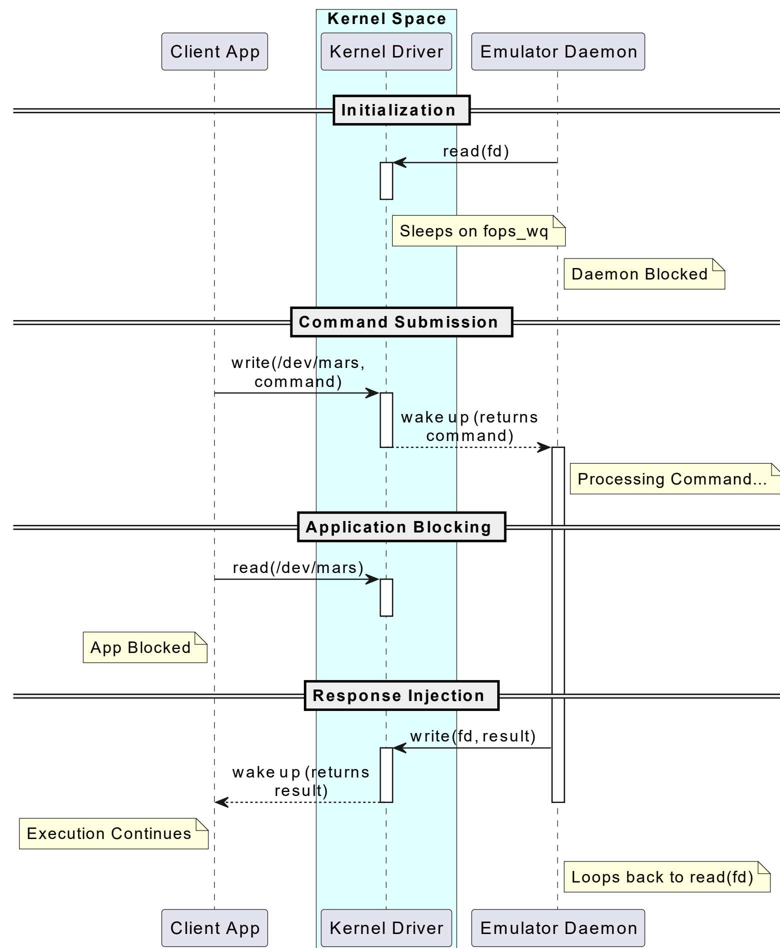

Figure 6.1. Data flow and synchronization managed by the `vmars-driver`.

This architecture guarantees that race conditions are prevented via `buf_mutex`, while CPU cycles are conserved via the wait queues, creating a perfectly synchronized, blocking API experience for the client application despite the decoupled backend.

6.2 mars-api

The `mars-api` library provides the high-level interface for developers to interact with MARS services. Its primary goal is to abstract the complexities of the underlying transport and serialization layers.

6.2.1 Initialization and Concurrency Control

The MARS specification defines a strict lifecycle and concurrency model for client applications interacting with the Root of Trust. To adhere to this standard and ensure exclusive access, the `mars-api` implements the core session management functions prescribed by the specification.

```
MARS_RC MARS_ApiInit(void *ctx);
MARS_RC MARS_Lock();
MARS_RC MARS_Unlock();
```

Listing 6.4. MARS API initialization and locking functions

As shown in Listing 6.4, these functions ensure that access to the underlying virtual hardware is safely orchestrated:

- `MARS_ApiInit`: This function bootstraps the API layer for the calling application. It evaluates the provided context (`ctx`) and initializes the internal data structures required to manage the connection to the virtual device.
- **`MARS_Lock` and `MARS_Unlock`**: To prevent interleaved command execution, the API enforces a strict mutual exclusion mechanism. Rather than relying on easily bypassed user-space locks, this exclusion relies entirely on the kernel-backed access control of the `vmars-driver`. `MARS_Lock` attempts to open the `/dev/mars` device node, securing exclusive system-level access if successful. Conversely, `MARS_Unlock` simply closes the file descriptor, safely releasing the kernel lock and freeing the resource for other applications.

6.2.2 Marshal-Unmarshal Architecture

A key design goal for `mars-api` was independence from specific serialization libraries. This was achieved by defining a generic interface that abstracts the encoding process.

```c
int32_t mu_marshal(void *dst, size_t dst_size, const char *ptypes, ...);
uint32_t mu_unmarshal(void *src, size_t src_size, const char *ptype, ...);
```

Listing 6.5. Marshal/Unmarshal interfaces

As shown in Listing 6.5, these functions allow the library to parse or encode multiple elements based on a format string. The implementation of any command within the API follows a standardized pipeline:

1. Verify the device lock for the current thread to ensure thread safety.
2. Marshal the input parameters into a CBOR-encoded buffer.
3. Transport the buffer to the `vmars-driver` and wait for the response.
4. Unmarshal the received bytes into the output variables and return the status code.

To select the serialization backend, the variable `MARS_MU_LIB` is used during the build process, currently supporting `tinycbor` and `libcbor`.

```
# Compile using the default cbor library (e.g. tinycbor)
> make

# Select specific CBOR implementations
> MARS_MU_LIB=MARS_MU_LIBCBOR make
> MARS_MU_LIB=MARS_MU_TINYCBOR make
```

Listing 6.6. mars-api compilation commands

6.3 libmars

The `libmars` library serves as the core "brain" of the emulator, containing the logic for state management, profile initialization, and the command dispatcher.

6.3.1 State and Profile Initialization

The libmars library provides a mechanism to initialize its internal state, exposing a routine that the mars-emulator implementation uses to instantiate the device. This routine consumes a profile configuration and a root seed to establish the emulator's initial conditions.

```
typedef enum {
    PROFILE_NULL = 0,
    PROFILE_ASCON,
    PROFILE_COUNT,
} Profile_Algorithm;

typedef struct {
    uint16_t count_pcr;
    uint16_t count_tsr;
    uint16_t alg_hash;
    uint16_t alg_sign;
    uint16_t alg_skdf;
    uint16_t alg_akdf;
} MARS_Profile;
```

Listing 6.7. MARS profile configuration structures

As shown in Listing 6.7, the device capabilities are defined through the MARS_Profile structure. This structure encapsulates the fundamental properties of the Root of Trust instance, allowing the emulator to specify the number of available registers (count_pcr, count_tsr) and the specific cryptographic algorithms to be used (mapped via identifiers like those in the Profile_Algorithm enumeration).

```
bool MARS_Init(MARS_Profile profile, const uint8_t *seed, uint16_t seed_size)
{
    if(!profile_init(profile)){
        return false;
    }
    if(!keys_init(seed, seed_size)) {
        return false;
    }
    fail_mode = false;

    CryptDpInit();
    return true;
}
```

Listing 6.8. MARS initialization function exposed to the emulator

The MARS_Init function (Listing 6.8) acts as the primary entry point for configuring the library. The first step of this sequence is the invocation of profile_init. This internal function evaluates the provided MARS_Profile, validates the requested parameters against the capabilities compiled into the library, and populating the library's internal state variables. Once initialized, these profile values are actively used by the MARS_CapabilityGet command to report the device's configuration to calling applications.

Key Hierarchy and Registers

A critical part of the initialization is the setup of the Root of Trust's primary seeds and volatile registers. Returning to the MARS_Init sequence:

- **Primary Seed Initialization**: If the profile initialization succeeds, keys_init is called using the seed provided by the emulator. This securely loads the Primary Seed (PS), which acts as the root of the key hierarchy.
- **Derivation Parent**: Next, CryptDpInit() is executed. This function initializes the Derivation Parent (DP), which is derived from the Primary Seed and serves as the root key for the device's derivation hierarchy, allowing for the secure generation of child keys.

- **Register Creation**: The creation of the volatile registers is dynamically governed by the variables defined in the profile. To optimize memory usage and simplify access, the emulator implements a flat, contiguous memory model for all registers.

```
uint8_t *REG = NULL;

#define reg_get(r, i) ((r)[(i) * profile_get(PROFILE_LEN_DIGEST)])

bool registers_init(uint16_t registers_count) {
    if(registers_count > PROFILE_MAX_COUNT_REG) return false;

    REG = calloc(registers_count, profile_get(PROFILE_LEN_DIGEST));
    return REG != NULL;
}

void *registers_get_pcr_address(uint16_t pcr_index) {
    if(!REG || pcr_index >= profile_get(PROFILE_COUNT_PCR)) return NULL;
    return &reg_get(REG, pcr_index);
}

void *registers_get_tcr_address(uint16_t tsr_index) {
    if(!REG || tsr_index >= profile_get(PROFILE_COUNT_TSR)) return NULL;
    return &reg_get(REG, tsr_index + profile_get(PROFILE_COUNT_PCR));
}
```

Listing 6.9. Dynamic allocation and memory layout of MARS registers

As detailed in Listing 6.9, the `registers_init` function allocates a single byte array (`REG`) large enough to hold all requested registers. The use of `calloc` is particularly important here, as it not only allocates the exact amount of memory required based on the cryptographic profile's digest length, but it also automatically zeroes out the memory block. This satisfies the requirement that volatile registers must start in a known zeroized state upon initialization.

Because all registers reside in the same array, access functions use a simple offset calculation. PCRs occupy the lower indices, while TSRs are offset by the total number of PCRs (`tsr_index + profile_get(PROFILE_COUNT_PCR)`). This contiguous design entirely avoids the fragmentation and overhead of allocating each register individually on the heap.

6.3.2 Cryptographic Abstraction

To ensure the emulator is not locked into a single cryptographic suite, an abstraction layer was implemented. The design of this abstraction is directly based on the architectural guidelines provided by the MARS specification. The specification outlines a standardized set of cryptographic interfaces to ensure interoperability and a consistent foundation across different Root of Trust implementations. This allows the emulator to call generic functions which are then mapped to specific algorithm implementations at compile time.

```
typedef union {
    ascon_shc_t ascon;
} profile_shc_t;

void CryptHashInit(profile_shc_t *hctx);
void CryptHashUpdate(profile_shc_t *hctx, const void *msg, size_t n);
void CryptHashFinal(profile_shc_t *hctx, void *dig);
```

Listing 6.10. Hash abstraction provided

As shown in Listing 6.10, which specifically illustrates the abstraction for the hash interface, the cryptographic functions map directly to the standardized operations defined by the MARS specification, while being designed to accept a generic `profile_shc_t` parameter. This encapsulates the specific state required by the underlying algorithm, ensuring that the abstraction layer's function signatures remain compliant and consistent even when supporting multiple cryptographic suites.

The decision to implement `profile_shc_t` as a `union` was driven by two key architectural constraints:

- **Stack Allocation**: A union guarantees that the size of the structure equals the size of its largest member. This allows the necessary memory for any supported cryptographic context to be allocated safely and efficiently directly on the stack, entirely avoiding the overhead and fragmentation risks associated with dynamic memory allocation.
- **Developer Ergonomics**: Using a union avoids the need for opaque pointers. Opaque pointers would require additional lifecycle management (e.g., explicit allocation and freeing functions), which introduces complexity. The union approach simplifies the developer experience, leading to safer and more straightforward code.

To illustrate how this abstraction manages the execution of the selected algorithm, Listing 6.11 shows the implementation of the `CryptHashInit` function.

```
void CryptHashInit(profile_shc_t *hctx) {
    switch(profile_get(PROFILE_ALG_HASH)) {
        case ASCON_ALG_HASH:
            AsconHashInit(&hctx->ascon);
            break;
        default:
            break;
    }
}
```

Listing 6.11. Implementation of the CryptHashInit abstraction

Within the abstraction layer, the function retrieves the currently configured hashing algorithm via the `profile_get` function. Using a `switch` statement, the operation is delegated to the specific algorithm's initialization routine-in this case, `AsconHashInit`-passing a reference to the corresponding member of the `profile_shc_t` union. This architectural pattern effectively isolates the command dispatcher from the complexities of the underlying libraries.

Currently, the implementation focuses on the **Ascon** suite for lightweight hashing and signature generation. By utilizing this abstraction, the dispatcher remains agnostic of whichever cryptographic algorithm is being used, fulfilling the specification's interoperability requirements while maintaining algorithmic flexibility.

6.3.3 Sequence Management

Certain MARS operations, such as hashing large amounts of data that exceed standard buffer sizes, require a multi-step sequence execution. The emulator must maintain an internal state to track these ongoing operations across multiple command invocations (e.g., `MARS_SequenceHash`, `MARS_SequenceUpdate`, and `MARS_SequenceComplete`).

```
typedef enum {
    SEQUENCE_NONE,
    SEQUENCE_HASH,
} Sequence_Type;

extern profile_shc_t shc;

bool sequence_start(Sequence_Type s);
Sequence_Type sequence_type();
void sequence_complete();
```

Listing 6.12. Sequence management interfaces

As shown in Listing 6.12, the emulator tracks the active sequence using the `Sequence_Type` enumeration. According to the current MARS specification, `HASH` is the only supported sequence type. However, the specification explicitly leaves the door open for future sequence types. The use of an enumeration anticipates this evolution, allowing the emulator to seamlessly identify and route new sequence types in future iterations.

The lifecycle of a sequence is managed by three core functions:

- `sequence_start()`: Validates that no other sequence is currently active and transitions the state from `SEQUENCE_NONE` to the requested sequence type.
- `sequence_type()`: Returns the currently active sequence type, allowing subsequent update or complete commands to verify they are operating on the correct sequence context.
- `sequence_complete()`: Finalizes the operation and resets the internal state back to `SEQUENCE_NONE`, freeing the engine for new requests.

To manage the cryptographic context of the active sequence, the library exposes `extern profile_shc_t shc`. This variable provides the command implementations with direct access to the sequence's internal cryptographic state. This design avoids the overhead of passing context pointers back and forth through the dispatcher, allowing commands like `MARS_SequenceUpdate` to directly feed data into the active hash engine.

6.3.4 Command Dispatching and Implementation

The `libmars` library implements the core MARS API command set. The execution of these commands is orchestrated by a central dispatcher, which serves as the boundary between the serialized transport layer and the internal emulator logic.

```c
int32_t MARS_dispatch(void *result, size_t result_size, void *command, size_t command_len);
```

Listing 6.13. MARS dispatcher interface

As shown in Listing 6.13, the `MARS_dispatch` function receives the raw byte stream forwarded by the emulator daemon. Both the incoming `command` buffer and the expected `result` buffer are encoded using the CBOR format, maintaining consistency with the `mars-api` design discussed earlier.

The dispatcher operates a strict request-response lifecycle:

1. **Unmarshalling**: It decodes the incoming CBOR `command` payload to extract the specific command code and its associated parameters, translating them into native C structures.
2. **Routing and Execution**: Based on the extracted command code, it routes the execution to the appropriate internal function, applying the cryptographic or state-altering logic against the emulator's registers and keys.
3. **Marshalling**: Upon completion, it encodes the output data, along with the execution status codes, back into CBOR format. This serialized data is written to the `result` buffer, which is subsequently transmitted back to the calling user-space application.

Through this mechanism, the dispatcher currently supports and routes the following core MARS operations:

- **MARS_CapabilityGet**: Implemented to dynamically construct its response by querying the emulator's internal static configuration state (populated during initialization), ensuring the reported capabilities strictly match the active profile.
- **MARS_Derive**: Utilizes the cryptographic abstraction layer to derive child parameters from the Derivation Parent (DP).
- **MARS_PcrExtend**: Interacts directly with the dynamically allocated register array, utilizing the abstracted hash engine to perform the cumulative measurement.
- **MARS_Quote**: Implemented to gather the requested PCR values from the contiguous memory block and sign them using the initialized profile's signature algorithm (currently Ascon).

- **MARS_RegRead**: Provides direct, bounds-checked read access to the REG memory array, returning the raw bytes of the requested PCR or TSR.
- **MARS_SequenceHash**, **MARS_SequenceUpdate**, and **MARS_SequenceComplete**: Implemented collectively using the Sequence_Type state machine. These commands manipulate the shared `extern profile_shc_t shc` context to process segmented payloads without blocking the main dispatcher loop.

6.4 mars-emulator

The `mars-emulator` is the user-space daemon responsible for orchestrating the entire virtual Root of Trust. Its lifecycle consists of loading the initial state configuration, provisioning the kernel-level virtual device, and entering a continuous loop to actively process incoming commands.

6.4.1 Virtual Device Instantiation

Before the emulator can start processing MARS commands, it must establish a dedicated communication channel with the kernel-space `vmars-driver`. This is achieved by interacting with a central proxy control node exposed by the driver.

```
1  bool vmars_create_proxy_dev(struct vmars_proxy_ioc *vmars_proxy_ioc) {
2      int fd = open("/dev/vmarsx", O_RDWR);
3      if(fd < 0) {
4          fprintf(stderr, "ERROR: could not open device\n");
5          return false;
6      }
7
8      int err = ioctl(fd, VMARS_PROXY_IOC_NEW_DEV, vmars_proxy_ioc);
9
10     if(err < 0) {
11         fprintf(stderr, "ERROR: could not do ioctl because: %s\n", strerror(errno));
12         close(fd);
13         return false;
14     }
15
16     close(fd);
17     return true;
18 }
```

Listing 6.14. Virtual device creation via ioctl

As shown in Listing 6.14, the emulator first opens the control device node, `/dev/vmarsx`. It then issues the `VMARS_PROXY_IOC_NEW_DEV` `ioctl` command, passing a pointer to a `vmars_proxy_ioc` structure.

The structure definition and memory layout of `vmars_proxy_ioc` are tightly coupled with the kernel module and are detailed in the `vmars-driver` section. From the emulator's perspective, this structure acts as an output buffer. When the kernel driver processes the `ioctl`, it dynamically allocates a new anonymous character device for this specific MARS instance. The driver populates the structure with the new device's major number, minor number, and most importantly, an open file descriptor (`fd`) directly linked to the newly created anonymous device.

Once the `ioctl` returns successfully, the emulator closes the general `/dev/vmarsx` control node. It then uses the provided file descriptor to drop into its main execution loop. Within this loop, the daemon blocks on read operations, waiting for the driver to forward marshaled commands from client applications. Upon receiving a command, it dispatches the payload to `libmars` for execution and writes the resulting CBOR-encoded output back to the file descriptor.

6.4.2 mars-setup

The `mars-setup` utility is a specialized sub-program designed to work in tandem with the `mars-emulator`. Before the emulator daemon can be launched, it requires a well-defined initial state. This utility

is responsible for generating that configuration, provisioning the device, and ensuring that the initial cryptographic parameters are correctly established.

```
> ./mars-setup ./ --PS 1234567890abcdef --pcr 4 --hash ascon
```

Listing 6.15. Example usage of the mars-setup utility

As demonstrated in Listing 6.15, the utility is invoked via the command line interface with parameters that define the device's fundamental profile. In this specific execution example:

- `./`: Specifies the target output directory where the configuration files for the new device will be generated and stored.
- `--PS 1234567890abcdef`: Injects the Primary Seed (PS) in hexadecimal format. This establishes the root of the key derivation hierarchy that `libmars` will later load during initialization. As this is a software emulator designed for testing and development, this seed is simply written to the configuration file as standard data, lacking the hardware-backed security guarantees of a physical MARS device.
- `--pcr 4`: Defines the hardware profile by specifying that this instance should be allocated exactly 4 Platform Configuration Registers.
- `--hash ascon`: Selects the cryptographic suite to be used, ensuring that the emulator's abstraction layer routes hash operations to the Ascon implementation.

A key feature of the `mars-setup` utility is its ability to automatically assign a unique identifier (ID) to the newly created device configuration. While the current implementation of the `vmars-driver` and the emulator supports running only a single MARS instance at any given time, this ID generation is crucial for configuration management. It allows the host system to persist multiple distinct state profiles on disk without file collisions, enabling developers to easily switch between different testing scenarios. Once the setup is complete, the `mars-emulator` can be pointed to a specific generated configuration directory to boot that particular virtual Root of Trust.

6.4.3 Execution Loop

After successfully obtaining the file descriptor and initializing the internal state via `libmars`, the daemon enters its main operational phase: the execution loop. This loop acts as the continuous processing engine for the virtual Root of Trust.

```c
char command_buf[BUFSIZE];
char result_buf[BUFSIZE];

// [...] (Initialization and setup code omitted)

while(1) {
    int command_len = read(vmars_proxy_ioc.fd, command_buf, sizeof(command_buf));
    if(command_len > 0) {
        int32_t result_len = MARS_dispatch(result_buf, sizeof(result_buf), command_buf,
 command_len);
        write(vmars_proxy_ioc.fd, result_buf, result_len);
    }
}
```

Listing 6.16. Main execution loop of the MARS emulator

As illustrated in Listing A.9, the core execution loop relies on standard POSIX system calls to interact with the kernel and orchestrate the workload:

- **Blocking Read**: The `read()` system call blocks the daemon until the `vmars-driver` signals that a new command has been written by a client application to the proxy device. This synchronous design ensures the emulator yields the CPU while sitting idle.

- **Command Dispatching**: Once the marshaled CBOR payload is copied into `command_buf` (verified by `command_len > 0`), it is passed to the `MARS_dispatch` function. As detailed in the `libmars` section, the dispatcher handles unmarshalling, routes the execution to the appropriate internal routine, and populates `result_buf` with the serialized result.
- **Response Write**: Finally, a `write()` system call pushes the serialized response back through the file descriptor (`vmars_proxy_ioc.fd`) to the kernel. The `vmars-driver` then buffers this response and wakes up the sleeping client application.

Daemon Termination

By design, the emulator runs indefinitely within a `while(1)` loop, mirroring the continuous uptime expected of a hardware Root of Trust. The loop does not contain internal break conditions for standard operations. Instead, termination of the emulator is managed externally by the host operating system. To halt the execution, the user or system manager must send an appropriate OS signal (such as `SIGINT` or `SIGTERM`) to the daemon process, which will interrupt the blocking `read()` system call and terminate the program.

Chapter 7

Testing and Evaluation

The primary objective of this chapter is to evaluate the functional correctness, stability, and interface transparency of the implemented MARS emulator architecture. Since the system spans both kernel-space transport mechanisms and user-space application logic, the evaluation focuses on validating the end-to-end execution pipeline and the strict concurrency controls mandated by the specification.

7.1 Experimental Setup

To ensure the emulator operates reliably in a modern, instruction-set-relevant environment, the testing and evaluation phase was conducted on an ARM64 architecture, reflecting the dominant hardware profile of constrained IoT devices.

The host machine utilized for virtualization was an Apple M1 MacBook (ARM64 architecture). The evaluation environment was provisioned as an Arch Linux virtual machine, operating with the Linux kernel version 6.18. This isolated virtualized setup provided a safe and highly controllable ecosystem for loading the custom `vmars-driver` kernel module, monitoring system call logs via `dmesg`, and executing the user-space daemon without risking host system instability. The `mars-api` and `mars-emulator` components were compiled using the standard GNU toolchain available in the Arch Linux repositories.

7.2 Functional Validation

Rather than relying on isolated unit tests, the functional validation of the emulator was conducted using a suite of purpose-built, C-based example programs. This approach was chosen to deliberately exercise the complete vertical stack of the architecture. By acting as real-world client applications, these programs verified the system's compliance with the MARS specification across three critical dimensions:

- **Session Management and Concurrency:** Verifying that the `vmars-driver` correctly enforces mutual exclusion via its atomic variables when multiple applications attempt to acquire the device.
- **End-to-End Command Routing:** Ensuring that commands are properly marshaled by the `mars-api`, safely transported through the kernel wait queues, processed by the `mars-emulator`, and correctly unmarshaled back to the client.
- **Interface Transparency:** Confirming that client applications can interact with the virtualized Root of Trust using standard MARS API calls, with zero awareness that the underlying hardware is emulated.

The following subsections detail the execution and results of these functional test scenarios.

7.2.1 Scenario 1: Cryptographic Pipeline and Registry Operations

The first phase of functional validation aimed to verify the core cryptographic and state-management capabilities of the MARS emulator. Specifically, the test evaluated the system's ability to hash a data payload, extend a Platform Configuration Register (PCR), and accurately read the resulting state from the virtualized hardware registry.

```
int main() {
    size_t outlen;
    char msg1[] = "TCG MARS demo";
    char dig[1024];

    // Initialize session and acquire kernel-level lock
    MARS_ApiInit(0);
    MARS_Lock();

    // Query device capabilities
    uint16_t diglen;
    MARS_CapabilityGet(MARS_PT_LEN_DIGEST, &diglen, outlen);
    printf("Capability digest len: %d\n", diglen);

    // Perform sequenced hashing operation
    MARS_SequenceHash();
    outlen = 0;
    MARS_SequenceUpdate(msg1, sizeof(msg1)-1, 0, &outlen);
    outlen = sizeof(dig);
    MARS_SequenceComplete(dig, &outlen);
    hexout("dig", dig, outlen);

    // Extend PCR and read back the registry value
    MARS_PcrExtend(0, dig);
    MARS_RegRead(0, dig);
    hexout("PCR0", dig, diglen);

    MARS_Unlock();
    return 0;
}
```

Listing 7.1. Functional test for MARS cryptographic and PCR operations

As shown in Listing 7.1, the client application successfully executes a sequence of complex MARS commands. The application requests the supported digest length, pushes a string payload (`"TCG MARS demo"`) through the hashing sequence, extends PCR index 0, and retrieves the updated registry value. The successful execution of this script validates that the `mars-api` correctly marshals multi-step cryptographic sequences, the `vmars-driver` maintains session state across multiple synchronous calls, and the `mars-emulator` accurately implements the TCG MARS hashing and PCR extension logic. Most importantly, the application logic remains entirely agnostic to the underlying CBOR serialization and kernel transport mechanisms.

7.2.2 Scenario 2: Remote Attestation and Quote Generation

Beyond basic cryptographic hashing and state storage, the defining feature of the MARS specification is its ability to provide hardware-backed attestations of the platform's state. To validate this capability, a test was designed to exercise the `MARS_Quote` command, which generates a cryptographic signature over a selected set of Platform Configuration Registers (PCRs).

```
int main() {
    size_t outlen;
    char msg1[] = "TCG MARS demo PCR0";
    char msg2[] = "TCG MARS demo PCR1";
    char dig[1024] = {0};
    char sig[1024] = {0};
    uint16_t cap_diglen;
    uint16_t cap_siglen;

    // Initialize session and acquire kernel-level lock
    MARS_ApiInit(0);
```

```
12      MARS_Lock();
13
14      // Dynamically retrieve supported digest and signature lengths
15      MARS_CapabilityGet(MARS_PT_LEN_DIGEST, &cap_diglen, outlen);
16      MARS_CapabilityGet(MARS_PT_LEN_SIGN, &cap_siglen, outlen);
17
18      // Hash first message and extend PCR0
19      MARS_SequenceHash();
20      outlen = 0;
21      MARS_SequenceUpdate(msg1, sizeof(msg1)-1, 0, &outlen);
22      outlen = sizeof(dig);
23      MARS_SequenceComplete(dig, &outlen);
24      MARS_PcrExtend(0, dig);
25      MARS_RegRead(0, dig);
26      hexout("PCR0", dig, cap_diglen);
27
28      // Hash second message and extend PCR1
29      MARS_SequenceHash();
30      outlen = 0;
31      MARS_SequenceUpdate(msg2, sizeof(msg2)-1, 0, &outlen);
32      outlen = sizeof(dig);
33      MARS_SequenceComplete(dig, &outlen);
34      MARS_PcrExtend(1, dig);
35      MARS_RegRead(1, dig);
36      hexout("PCR1", dig, cap_diglen);
37
38      // Create bitmask to select PCR0 and PCR1
39      uint32_t regSelect = 1 << 0 | 1 << 1;
40
41      // Generate cryptographic quote (signature) over selected PCRs
42      if (MARS_Quote(regSelect, NULL, 0, NULL, 0, sig) != MARS_RC_SUCCESS) {
43          printf("ERROR: MARS_Quote failed\n");
44      } else {
45          hexout("Quote", sig, cap_siglen);
46      }
47
48      MARS_Unlock();
49      return 0;
50  }
```

Listing 7.2. Functional test validating the MARS Quote and attestation mechanism

As demonstrated in Listing 7.2, the client application simulates a remote attestation flow. It dynamically queries the emulator for its supported cryptographic lengths, independently extends PCR0 and PCR1 with distinct digests, and constructs a `regSelect` bitmask to target both registers. The successful execution of `MARS_Quote` validates several critical components of the architecture:

1. **Asymmetric Cryptography Execution:** The `mars-emulator` successfully securely aggregates the selected PCR values and computes an asymmetric signature (the Quote) in user space.

2. **Complex Payload Marshaling:** The `mars-api` successfully encodes the bitmask and correctly allocates buffers for the inherently larger payload size of a cryptographic signature.

3. **Kernel Transport Capacity:** The `vmars-driver` seamlessly handles the routing of the larger signature response through the wait queue infrastructure without truncating or corrupting the payload.

By passing this test without requiring any emulator-specific workarounds, the system proves it is fully capable of handling the most complex, payload-heavy operations demanded by the TCG MARS specification.

7.2.3 Scenario 3: Concurrency and Device Exclusivity Stress Test

A fundamental requirement of the MARS specification is the strict isolation of the hardware device during a cryptographic session. To validate that the `vmars-driver` correctly enforces

mutual exclusion and prevents interleaved command corruption, a multi-threaded stress test was developed.

```c
#define NUM_THREAD 10

void *mars_thread(void *arg) {
    int thread_num = (unsigned long)arg;
    int i = 0;
    MARS_RC rc = MARS_ApiInit(0);
    if(rc) {
        fprintf(stdout, "MARS_ApiInit Error: got rc=%d\n", rc);
        return (uint32_t *)1;
    }
    fprintf(stdout, "[Thread %d]: MARS_ApiInit: done\n", thread_num);

    // Attempt to acquire exclusive device lock
    MARS_Lock();
    fprintf(stdout, "[Thread %d]: MARS_Lock: done\n", thread_num);

    // Hold lock and execute commands
    while(i < 3) {
        MARS_SelfTest(true);
        fprintf(stdout, "[Thread %d]: MARS_SelfTest: done\n", thread_num);
        i++;
        sleep(2);
    }

    // Release lock for the next thread
    fprintf(stdout, "[Thread %d]: MARS_Unlock\n", thread_num);
    MARS_Unlock();
    return NULL;
}

int main(int argc, char **argv) {
    signal(SIGINT, manage_sigint);
    unsigned long i = 0;
    pthread_t tids[NUM_THREAD];
    fputs("Testing the mars-apis\n", stdout);

    // Spawn 10 competing threads
    for(i = 0; i < NUM_THREAD; i++) {
        pthread_create(&tids[i], NULL, mars_thread, (void*)i);
    }

    // Await completion of all threads
    for(i = 0; i < NUM_THREAD; i++) {
        pthread_join(tids[i], NULL);
    }
    return 0;
}
```

Listing 7.3. Multi-threaded stress test validating kernel-level mutual exclusion

Listing 7.3 demonstrates a scenario where 10 concurrent threads are spawned to simultaneously access the emulated MARS device. Each thread is assigned a unique identifier, executes `MARS_Lock()`, performs a loop of `MARS_SelfTest` operations while artificially holding the CPU via `sleep(2)`, and subsequently releases the lock. The main process utilizes `pthread_join` to ensure the complete execution lifecycle of all competing threads is captured.

By monitoring the standard output logged by the unique thread identifiers, the emulator's `atomic_t` lock mechanism at the kernel driver layer was proven to perform exactly as intended. Only a single thread ID was observed progressing past the `MARS_Lock()` acquisition at any given time. The remaining 99 threads were safely blocked by the driver, queueing until the active thread printed its `MARS_Unlock` statement. This result definitively proves that the virtualized environment provides the rigorous, OS-level concurrency protections expected from a physical Hardware Root of Trust, making it impossible for competing user-space applications to hijack an active cryptographic session.

7.3 Summary of Evaluation Results

The testing and evaluation phase successfully demonstrated that the MARS emulator architecture meets the rigorous demands of the TCG specification, functioning not merely as a high-level API mock, but as a robust, system-level virtual device.

By executing a suite of targeted test applications in an ARM64 environment, the architecture was validated across all primary objectives:

- **Functional Completeness:** The successful execution of both basic cryptographic pipelines (Hashing, PCR Extension) and complex asymmetric operations (Remote Attestation Quotes) proved that the `mars-emulator` daemon correctly implements the core specification logic, and that the `mars-api` reliably handles CBOR serialization for varying payload sizes.
- **Architectural Security:** The multi-threaded stress test definitively validated the `vmars-driver`. By leveraging atomic operations within the Linux kernel, the proxy driver successfully enforced hardware-grade mutual exclusion, ensuring that active cryptographic sessions remain isolated and immune to user-space race conditions.
- **Interface Transparency:** Across all test scenarios, the client applications interacted exclusively with standard MARS API functions. No emulator-specific modifications, conditional compilation flags, or synchronization workarounds were required in the client code.

Ultimately, these results confirm that the developed architecture successfully bridges the current hardware availability gap. It provides IoT developers and security researchers with a highly accurate, thread-safe, and transparent software environment for prototyping MARS-compliant applications today, ensuring that code written against this emulator will transition seamlessly to physical silicon once available.

Chapter 8

Conclusions

8.1 Summary of Contributions

The proliferation of constrained IoT devices requires a paradigm shift in how Hardware Roots of Trust are designed and deployed. While the TCG MARS specification provides the necessary lightweight blueprint for these environments, the lack of available physical silicon has created a significant barrier to research, development, and adoption.

This thesis addressed that critical gap by designing, implementing, and validating a comprehensive, Linux-based MARS emulation architecture. By distributing the system across a custom kernel proxy driver (`vmars-driver`), a user-space abstraction library (`mars-api`), a background execution daemon (`mars-emulator`), and a core emulation library (`libmars`), this project successfully delivered a robust, architecturally faithful virtualization stack.

The core contributions of this work include:

- **API Transparency and Portability:** Developers can interact with the emulator using standard MARS API calls without requiring emulator-specific code modifications, enabling seamless future transitions to physical hardware.
- **System-Level Concurrency Control:** By pushing the device lock down to the OS kernel via atomic operations, the architecture accurately mimics the strict, single-session mutual exclusion required by physical hardware, preventing race conditions and session hijacking.
- **Decoupled Serialization Architecture:** The implementation abstracts the transport layer from the cryptographic logic, utilizing a flexible CBOR-based Marshal/Unmarshal pipeline that safely routes variable-length payloads—such as attestation quotes—across system boundaries.

Through rigorous functional and concurrent stress testing, the architecture was proven to be compliant, thread-safe, and highly stable, successfully fulfilling the objective of providing a reliable MARS prototyping environment ahead of hardware availability.

8.2 Limitations

While the emulator provides a functionally and architecturally accurate software representation of the MARS specification, it inherently possesses the limitations of any virtualized hardware. First, the emulator cannot simulate physical side-channel attacks (such as power analysis or fault injection) or physical tampering countermeasures. Second, the cryptographic operations performed by the `mars-emulator` rely on the host operating system's pseudo-random number generators (PRNG) and user-space memory protections, lacking the deep physical isolation guarantees of a true discrete silicon chip.

8.3 Future Work

The modular nature of this architecture establishes a strong foundation for several future research and development trajectories:

- **Integration with Trusted Execution Environments (TEEs):** To elevate the emulator from a pure testing tool to a deployable firmware-based Root of Trust (fRoT), the `mars-emulator` daemon could be ported to run within a TEE, such as ARM TrustZone or OP-TEE. This would isolate the cryptographic keys and PCR states from the rich operating system, providing security guarantees closer to physical hardware.
- **Virtual Machine Manager (QEMU) Integration:** To expand testing capabilities beyond host-level applications, the emulator could be integrated with full-system hypervisors such as QEMU. By exposing the `mars-emulator` to guest virtual machines as an attached hardware peripheral (e.g., via a `virtio` interface or simulated memory-mapped I/O), developers could validate the entire boot chain and security stack of lightweight IoT operating systems (such as Zephyr or FreeRTOS) in a completely virtualized environment.
- **Remote Attestation Verification Frameworks:** With the `MARS_Quote` functionality validated, future work could focus on building the corresponding remote verifier infrastructure. This would involve developing network-facing services that consume, verify, and act upon the attestation quotes generated by the emulated IoT devices.
- **Extended Cryptographic Suites:** The current implementation can be expanded to support additional cryptographic profiles defined by the TCG, including post-quantum cryptographic (PQC) algorithms, ensuring the MARS ecosystem remains resilient against emerging threats.

8.4 Concluding Remarks

The transition to a secure IoT infrastructure cannot wait for the mass manufacturing of new silicon. By delivering a robust, thread-safe, and architecturally faithful MARS emulator, this project eliminates the hardware bottleneck, empowering the cybersecurity community to build, test, and validate the next generation of lightweight secure applications today.

Bibliography

[1] J. Teo, "Features and benefits of trusted computing", 2009, DOI 10.1145/1940976.1940990
[2] TCG, "Trusted platform module 2.0 library part 0: Introduction", 2025, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf
[3] K. G. W. Arthur, D. Challener, "A practical guide to tpm 2.0", Springer Nature, 2015
[4] TCG, "Trusted platform module 2.0 library part 1: Architecture", 2025, https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf
[5] V. Costan and S. Devadas, "Intel SGX explained", Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086
[6] TCG, "Dice layering architecture", 2020, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf
[7] ARM, "Learn the architecture - trustzone for aarch64", 2024, https://developer.arm.com/documentation/102418/0102
[8] lowRISC, "Opentitan: Open source silicon root of trust", https://opentitan.org/
[9] S. Berger, "libtpm implementation", 2015, https://github.com/stefanberger/libtpm/
[10] S. Berger, "swtpm source code", 2015, https://github.com/stefanberger/swtpm/
[11] S. Berger, "swtpm-setup implementation in the swtpm source code", 2015, https://github.com/stefanberger/swtpm/tree/master/src/swtpm_setup
[12] S. Berger, "Virtual tpm proxy driver for linux kernel", 2015, https://elixir.bootlin.com/linux/v6.12.4/source/drivers/char/tpm/tpm_vtpm_proxy.c
[13] S. Berger, "Qemu with vtpm support", 2015, https://github.com/stefanberger/qemu-tpm
[14] Microsoft and Trusted Computing Group, "Tpm 2.0 reference implementation (ms-tpm-20-ref)", 2015, https://github.com/microsoft/ms-tpm-20-ref
[15] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee – an open virtual trusted execution environment", 2015 IEEE Trustcom/BigDataSE/ISPA, 2015, pp. 400–407, DOI 10.1109/Trustcom.2015.400
[16] TCG, "Mars serialization interface specification", https://trustedcomputinggroup.org/wp-content/uploads/Measurement-and-Attestation-RootS-Serialization-Interface-Specification-Version-1-Revision-0_pub-1.pdf
[17] TCG, "Mars api specification", https://trustedcomputinggroup.org/wp-content/uploads/TCG_MARS_API_Spec_Version-1_Revision-2_pub.pdf
[18] TCG, "Mars library specification", https://trustedcomputinggroup.org/wp-content/uploads/TCG_MARS_Library_Spec_v1r14_pub.pdf

Appendix A

User's Manual

A.1 Build Process

This section details the compilation and setup instructions for the MARS Emulator suite from source. To ensure a smooth installation and resolve dependencies correctly, it is recommended to build and configure the components in the following order:

- `vmars_driver`: The Linux kernel module that acts as the virtual proxy device.
- `mars-api`: The user-space library and headers required for custom applications to interact with the device.
- `libmars`: The core foundational library that implements the MARS hardware logic.
- `mars-emulator`: The host program that utilizes `libmars` to simulate the device and attaches to the kernel driver.

The following subsections outline the specific system requirements, source directory structures, and compilation commands for each of these individual components.

A.1.1 Building and Loading `vmars_driver`

The `vmars_driver` is a Linux kernel module that acts as a virtual proxy device. It is responsible for routing the communication between user-space applications (using `mars-api`) and the background `mars-emulator`.

Source Structure

The driver repository is organized to separate the public definitions from the internal module source code:

```
vmars_driver/
|-- include/
|   `-- vmars.h            # Structures and IOCTL definitions
|-- module/                # Source code for the kernel module
|-- test/                  # Test scripts and utilities
|-- Makefile
`-- README.md
```

The `include/vmars.h` header file is highly important for the broader suite. It exposes the necessary functions, structures, and IOCTL commands needed by the `mars-emulator` to instantiate and communicate with a new virtual MARS device on the host system.

Requirements

To compile the driver, ensure your system has the following dependencies installed:

- `make`
- `gcc`
- Linux kernel headers matching your currently running kernel (e.g., available via `linux-headers-$(uname -r)` on Debian/Ubuntu systems).

Compilation and Usage

Because of the provided `Makefile` structure, compiling the driver is as simple as running the `make` command in the root directory. This process generates the kernel object (`.ko`) file within the `module/` directory, which will be inserted into the Linux kernel in a later step (detailed in Section A.2.1).

```
git clone https://github.com/torsec/vmars_driver.git
cd vmars_driver

# Compile the kernel module
make
```

Listing A.1. Compilation and loading commands for vmars_driver

A.1.2 Building `mars-api`

The `mars-api` component provides the user-space libraries and headers required for custom applications to interact with the MARS device.

Requirements

To compile the `mars-api`, ensure the following dependencies are installed on your host system:

- `gcc`
- `make`
- A CBOR (Concise Binary Object Representation) library.

Source Structure

The repository is organized to separate the public API headers from the frontend command implementations, with each MARS command implemented in its own source file.

```
mars-api/
|-- include/
|   `-- mars/
|       |-- mars.h              # Core MARS definitions
|       `-- api.h               # Public API function declarations
`-- src/
    |-- mu/
    |   |-- mars_mu.h           # Marshal/Unmarshal definitions
    |   `-- mars_mu.c           # Marshal/Unmarshal implementation
    |-- common.h                # Common internal definitions
```

```
|-- MARS_CapabilityGet.c      # API command implementations...
|-- MARS_Derive.c
|-- MARS_DpDerive.c
|-- MARS_PcrExtend.c
|-- MARS_PublicRead.c
|-- MARS_Quote.c
|-- MARS_RegRead.c
|-- MARS_SelfTest.c
|-- MARS_SequenceComplete.c
|-- MARS_SequenceHash.c
|-- MARS_SequenceUpdate.c
|-- MARS_Sign.c
|-- MARS_SignatureVerify.c
`-- api.c                     # Core API routing
```

Supported CBOR Libraries

The API requires a backend to process CBOR data. You can choose between two supported libraries depending on your environment and requirements. This selection is handled during compilation using the `MARS_MU_LIB` environment variable:

- `libcbor` (`MARS_MU_LIBCBOR`): A robust, general-purpose C library for parsing and generating CBOR. It offers complete RFC conformance and flexible memory management. It is well-suited for standard host systems where resource constraints are not a primary concern.
- `tinycbor` (`MARS_MU_TINYCBOR`): A highly optimized C/C++ implementation of CBOR (originally developed by Intel). It is designed for extremely fast operation and a minimal memory footprint, making it an excellent choice for lightweight emulators or resource-constrained environments.

Compilation

The compilation process for the API uses `make`. The default CBOR library configured in the `mars-api` Makefile is `tinycbor`. You can override this using the `MARS_MU_LIB` variable.

```
# Clone the repository and navigate into it
git clone https://github.com/torsec/mars-api.git
cd mars-api

# Compile the API using tinycbor (the default library)
make

# Alternatively, compile the API using libcbor
MARS_MU_LIB=MARS_MU_LIBCBOR make
```

Listing A.2. Compilation commands for mars-api

A.1.3 Building `libmars`

Once the API is ready, you can compile `libmars`. This component serves as the foundational emulation library that actually implements the MARS hardware logic under the hood.

Requirements

Before compiling `libmars`, ensure the following dependencies are installed on your host system:

- `wget`
- `unzip`
- `gcc`
- `make`
- A CBOR library.

Note: The cryptographic library required for the MARS implementation (currently the Ascon lightweight cryptography suite) does not need to be manually installed beforehand. The `Makefile` is configured to automatically download and extract it by default during the build process.

Compilation

To build the library, navigate to the root directory of the `libmars` project and run the `make` utility. Just like the `mars-api`, you can customize which CBOR library to use by passing the `MARS_MU_LIB` environment variable. If no environment variable is provided, the build system will default to its pre-configured CBOR library.

```
git clone https://github.com/torsec/libmars.git
cd libmars

# Compile using the default CBOR library (and automatically download Ascon)
make

# Speed up the build process using all available CPU cores
make -j$(nproc)

# Compile specifically using tinycbor
MARS_MU_LIB=MARS_MU_TINYCBOR make

# Compile specifically using libcbor
MARS_MU_LIB=MARS_MU_LIBCBOR make
```

Listing A.3. Compilation commands for libmars

Directory Structure and Output

Upon a successful build, the compilation artifacts and necessary header files will be available in the following structure:

```
libmars/
|-- include/
|   |-- dispatcher.h     # Functions and data structures for the dispatcher
|   `-- profile.h        # Functions and data structures for the MARS profile
|-- src/                 # Source files
|-- build/               # Generated upon compilation
|   `-- libmars.so       # The compiled dynamic library
`-- Makefile
```

To link against `libmars`, include the `include` directory in your compiler's include path (e.g., `-I/path/to/libmars/include`) to access the dispatcher and profile definitions, and link the generated library file (e.g., `-L/path/to/libmars/build -lmars`).

A.1.4 Building mars-emulator

The `mars-emulator` repository provides the central programs that bring the entire suite together. It utilizes the `libmars` library to simulate the MARS device logic and attaches to the `vmars_driver` to expose this virtual device to the host system. The build process produces both a setup utility to configure virtual instances and the main emulator executable.

Requirements

Before compiling the emulator, ensure the following dependencies and previously built components are available on your system:

- `gcc`
- `make`
- `libmars` (for core device emulation)
- `vmars_driver` (to instantiate the proxy device)
- `mars-api` (required only if compiling the example applications)

Compilation

To compile the emulator suite, navigate to the cloned repository and use the `make` command. If you have built `libmars`, `vmars_driver`, or `mars-api` in custom local directories rather than installing them system-wide, you must export specific environment variables.

```
git clone https://github.com/torsec/mars-emulator.git
cd mars-emulator

# Compile the emulator tools
make

# Compile the setup application
make mars-setup

# Compile the included example applications
make examples

# Compiling with custom paths for libmars and vmars_driver
export LIBMARS_PATH=/path/to/libmars/build
export LIBMARS_INCLUDE_PATH=/path/to/libmars/include
export VMARS_INCLUDE_PATH=/path/to/vmars/include
make

# Compiling examples with custom paths for mars-api
export MARS_API_PATH=/path/to/mars-api/build
export MARS_API_INCLUDE_PATH=/path/to/mars-api/include
make examples
```

Listing A.4. Compilation commands for mars-emulator

A.2 Setup and Execution

Once all components are compiled, you must initialize the environment before interacting with the virtual device. This involves loading the driver, configuring the instance, and starting the daemon.

A.2.1 Loading the Kernel Driver

The main component needed to run the emulator is the driver. You need to insert the compiled kernel object (.ko) into the Linux kernel. This operation requires root privileges.

```
sudo insmod path/to/vmars_driver/module/vmars.ko
```

Listing A.5. Loading vmars_driver

A.2.2 Device Configuration (mars-setup)

Before starting the emulator, you must create a persistent device configuration using the `mars-setup` utility. This program generates a new instance configuration that defines the cryptographic algorithms, register parameters, and Primary Seed (PS) to be used. This ensures your virtual MARS device behaves consistently across reboots.

```
Usage: build/mars-setup <path> [options]
    <path> Path to device configuration folder

OPTIONS:
    --PS <hex> Specify the value of PS by passing an hex string
    --pcr <num> Specify the number of pcr to use
    --tsr <num> Specify the number of tsr to use
    --hash <alg> Specify the hash algorithm to use(DEFAULT ascon)
    --sign <alg> Specify the sign algorithm to use(DEFAULT ascon)
    --skdf <alg> Specify the skdf algorithm to use(DEFAULT %s)
    --akdf <alg> Specify the akdf algorithm to use(DEFAULT ascon)

# Example: Create a configuration in the current directory with 4 PCRs
build/mars-setup ./ --PS 1234567890abcdef --pcr 4 --hash ascon
```

Listing A.6. Usage and example for mars-setup

A.2.3 Starting the emulator

Once your configuration is generated, you can launch the emulator using the `mars-emu` executable. Running the emulator requires `sudo` (or root) privileges because it must interact directly with the `vmars_driver` kernel module.

You can use command-line flags to point the emulator to the configuration folder you created in the previous step.

```
# Basic syntax and options
Usage: build/mars-emu [options]
OPTIONS:
    --path <path> Path to device configuration folder (working dir by default)
    --id <id> Specify the id of the device to use (default: 0)
    --help Print this help message

# Example: Run the emulator using the configuration in the current directory
sudo build/mars-emu --path ./ --id 0

# If libmars was built in a custom directory, provide its location dynamically
sudo LD_LIBRARY_PATH=/path/to/libmars/build build/mars-emu --path ./
```

Listing A.7. Execution commands for mars-emu

A.3 Interacting with the Device

With the emulator actively running in the background, you can now interact with it using applications linked against the `mars-api`.

A.3.1 Example Applications

To demonstrate how to interact with the virtual MARS device using the `mars-api`, the emulator repository includes several example applications. These examples serve as excellent reference points for developers writing their own custom software.

Available Examples

The source code for these applications can be found in the `examples/` directory. The provided examples include:

- `mars_derive.c`: Demonstrates how to perform key derivation operations using the MARS device.
- `mars_getcapability.c`: Shows how to query the virtual device for its supported capabilities and configuration.
- `mars_quote.c`: Illustrates the process of generating a cryptographic quote for attestation purposes.
- `mars_read_extend.c`: Provides a practical example of reading from and extending the Platform Configuration Registers (PCRs).
- `mars_repl.c`: An interactive Read-Eval-Print Loop (REPL) utility that allows users to send commands to the MARS device dynamically from the command line.

Execution

As covered in the compilation section, these examples are built by running `make examples` in the root of the `mars-emulator` repository. The compiled binaries are output to the `examples/` directory.

Because the examples communicate with the `vmars_driver`, they must be run with `sudo` (or root) privileges. Additionally, if the `mars-api` library was built in a custom directory rather than installed system-wide, you must point the dynamic linker to its location using the `LD_LIBRARY_PATH` environment variable.

```
# Assuming you are in the root of the mars-emulator repository
# and mars-api is located at /path/to/mars-api/build

sudo LD_LIBRARY_PATH=/path/to/mars-api/build examples/mars_quote
```

Listing A.8. Running the mars_quote example

A.3.2 Writing Your First Application

Once the emulator is running and the driver is loaded, you can write custom C applications that interact with the virtual MARS device. This requires including the `mars-api` headers in your source code and linking against the compiled API library.

Application Skeleton

A typical application follows a standard lifecycle: it includes the necessary MARS headers, requests exclusive access to the device using a lock, invokes the required MARS API functions to perform cryptographic or attestation operations, and finally unlocks the device.

Locking the device before issuing commands is mandatory to prevent conflicts in environments where multiple threads or processes might attempt to access the MARS emulator concurrently.

Below is a basic boilerplate example demonstrating how to lock the device, query its capabilities, and unlock it:

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

/* Include the MARS API headers */
#include <mars/mars.h>
#include <mars/api.h>

int main() {
    printf("Starting MARS Device Interaction...\n");
    int32_t rc;

    /* 1. Request exclusive access to the MARS device.
     * Note: MARS_Lock() returns an error if the device is already locked by the app.
     * Since this is the first and only lock in this app, we omit the check. */
    MARS_Lock();

    /* 2. Execute MARS API commands */
    uint16_t ptc = 0;
    rc = MARS_CapabilityGet(MARS_PTC_TARGET, &ptc);

    if (rc == MARS_RC_SUCCESS) {
        printf("Successfully communicated with the MARS emulator.\n");
        printf("Device PTC: 0x%04X\n", ptc);
    } else {
        fprintf(stderr, "Failed to get capabilities. Error code: %d\n", rc);
    }

    /* 3. Release the lock to allow other threads to access the device */
    MARS_Unlock();

    return (rc == MARS_RC_SUCCESS) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing A.9. Example: my_mars_app.c

Compiling Your Application

To compile your custom application, you must instruct the compiler where to find the `mars-api` headers and the compiled library file.

Assuming you saved the code above as `my_mars_app.c`, you can compile it using `gcc` as follows:

```
# Set the path to your mars-api repository
export API_DIR=/path/to/mars-api

# Compile the application
gcc my_mars_app.c -o my_mars_app \
    -I${API_DIR}/include \
    -L${API_DIR}/build \
    -lmars-api
```

Listing A.10. Compiling a custom application

Running Your Application

Just like the included examples, your custom application must be run with sufficient privileges to interact with the `vmars_driver`, and the dynamic linker must know where the API library is located.

```
sudo LD_LIBRARY_PATH=${API_DIR}/build ./my_mars_app
```

Listing A.11. Running the custom application

Appendix B

Developer Manual

B.1 Introduction

While Chapter 6 details the architectural design and internal state management of the MARS Emulator suite, this Developer Manual serves as a practical, action-oriented guide. It outlines the specific workflows required to extend the emulator's capabilities, integrate new cryptographic algorithms, and debug the stack across the user-kernel boundary.

B.2 Extending the Emulator: Implementing a Pending Command

The frontend user-space library (`mars-api`) is already feature-complete with respect to the MARS specification. It contains the function definitions and CBOR marshalling logic for every command. Therefore, extending the emulator to support a pending command only requires implementing the backend logic within the `libmars` repository.

To maintain compliance with the MARS specification, the emulator architecture strictly decouples serialization from execution. Core cryptographic functions must accept native C types, leaving the `MARS_dispatch` function entirely responsible for CBOR marshalling and unmarshalling.

The following steps outline the process of implementing `MARS_Sign`, a standard specification command that is already routed by the API but lacks backend execution.

B.2.1 Step 1: Review the Frontend API Contract

Before writing any code in the emulator, review the existing implementation in the `mars-api` repository to understand the expected input parameters and CBOR formatting.

The command codes and native C signatures are defined in `mars-api/include/mars/mars.h`:

```c
#define MARS_CC_Sign 11

MARS_RC MARS_Sign(const void * ctx, uint16_t ctxlen, const void * dig, void * sig);
```

Listing B.1. Existing definitions in mars.h

Next, inspect the frontend routing logic (e.g., `mars-api/src/MARS_Sign.c`) to see exactly how the parameters are packed into the CBOR payload.

```c
/* Inside mars-api/src/MARS_Sign.c */
#include <mars/api.h>
#include <mars/mars.h>
#include "common.h"
```

```c
#include "mu/mars_mu.h"

MARS_RC MARS_Sign(const void * ctx, uint16_t ctxlen, const void * dig, void * sig) {
    char cmdbuf[MU_MAX_SIZE], rcvbuf[MU_MAX_SIZE];
    int32_t cmdbuf_len;
    size_t rcvbuf_len;
    MARS_RC rc;

    // check if the current thread locked the device
    if(!mctx_locked()) return MARS_RC_LOCK;

    /* Note the format 'hxx': half-word (CC), string (ctx), string (dig) */
    cmdbuf_len = mu_marshal(cmdbuf, MU_MAX_SIZE, "hxx",
                        MARS_CC_Sign, ctx, ctxlen, dig, mctx_diglen());
    if(cmdbuf_len < 0) return MARS_RC_IO;

    rcvbuf_len = MARS_transport(rcvbuf, MU_MAX_SIZE, cmdbuf, cmdbuf_len);

    /* Note the format 'hX': half-word (RC), fixed-string (sig) */
    if(mu_unmarshal(rcvbuf, rcvbuf_len, "hX", &rc, sig, mctx_siglen()) && !rc) {
        return MARS_RC_IO;
    }

    return rc;
}
```

Listing B.2. Reviewing the existing frontend routing

This tells you exactly what the `libmars` dispatcher must unmarshal (`"hxx"`) and what it must return (`"hX"`).

B.2.2 Step 2: Implement the Backend Logic (`libmars`)

Move to the `libmars` repository. Create the native C function that actually performs the MARS hardware logic. This function must mirror the `mars.h` specification exactly and must not contain any serialization logic.

To maintain the project's organizational structure, place this new file within the `src/commands/` directory.

```c
/* Inside libmars/src/commands/MARS_Sign.c */
#include "mars/mars.h"
#include "profile.h"

MARS_RC MARS_Sign(const void * ctx, uint16_t ctxlen, const void * dig, void * sig) {
    /* 1. Derive the Restricted Key using the DP and the provided ctx.
     * 2. Use the cryptographic abstraction (e.g., Ascon) to sign the digest.
     * 3. Write the output to the 'sig' buffer.
     */

    // Implementation of core MARS specification logic goes here...

    return MARS_RC_SUCCESS;
}
```

Listing B.3. Implementing the backend logic in libmars

Note: Remember to update the `libmars` Makefile to include `src/commands/MARS_Sign.c` in the build compilation process.

B.2.3 Step 3: Route the Command in the Dispatcher

Finally, expose this new backend function to the serialized data stream. Update the `MARS_dispatch` function (located in `libmars/src/dispatcher.c`) to unmarshal the payload, invoke the backend function, and marshal the generated signature back to the user.

```c
/* Inside MARS_dispatch() switch block */
switch(mars_cc) {
    // ... existing commands ...

    case MARS_CC_Sign: {
        char ctx[BUFFER_LEN];
        char dig[BUFFER_LEN];

        /* NOTE: tinycbor requires the length pointers to be pre-populated
         * with the maximum capacity of the destination buffers to prevent overflows. */
        size_t ctx_len = BUFFER_LEN;
        size_t dig_len = BUFFER_LEN;

        /* Unmarshal inputs: CC (h), ctx (x), digest (x) */
        if(mu_unmarshal(command, command_len, "hxx", &mars_cc,
                        ctx, &ctx_len, dig, &dig_len)) {
            return -1;
        }

        /* Execute the native C function */
        mars_rc = MARS_Sign(ctx, ctx_len, dig, buf);

        /* Marshal the return code and the generated signature */
        /* Note: buf acts as the output signature container in the dispatcher */
        result_len = mu_marshal(result, result_size, "hx", mars_rc,
                                buf, profile_get(PROFILE_LEN_SIGN));
    } break;

    default:
        // Unimplemented command
        result_len = mu_marshal(result, result_size, "h", MARS_RC_COMMAND);
}
```

Listing B.4. Updating the libmars dispatcher

B.3 Extending the Emulator: Integrating a New Cryptographic Suite

The `libmars` repository is designed to be highly extensible. It abstracts all cryptographic operations (hashing, signing, and key derivation) away from the core MARS command dispatcher.

Currently, the emulator utilizes the Ascon lightweight cryptography suite. If a developer wishes to add support for a new algorithm (e.g., a standard SHA-256 implementation or a post-quantum signature scheme), they must integrate it into the abstraction layer and expose it to the configuration utility.

The following steps outline how to integrate a hypothetical new algorithm, `MYALGO`, into the emulator stack.

B.3.1 Step 1: Define Algorithm Identifiers and State

First, the new algorithm must be assigned a unique identifier, and its internal state structure must be added to the shared cryptographic union. Because the architecture strictly separates public definitions from internal memory layouts, this requires updating two separate headers.

Part A: Update the Public Profile
Navigate to `libmars/include/profile.h` and add the new algorithm identifier to the `Profile_Algorithm` enumeration:

```c
typedef enum {
    PROFILE_NULL = 0,
    PROFILE_ASCON,
    PROFILE_MYALGO, /* Add the new algorithm ID */
    PROFILE_COUNT,
```

```
} Profile_Algorithm;
```

Listing B.5. Updating the profile enumeration in profile.h

Part B: Update the Internal Cryptographic State

Next, navigate to the internal cryptographic header located at `libmars/src/crypto/crypto.h`. Include your external library's header and add its context structure to the `profile_shc_t` union. This ensures the emulator can allocate the correct amount of memory safely on the stack.

```c
/* Include the external library header providing the context struct */
#include <myalgo_lib.h>

typedef union {
    ascon_shc_t ascon;
    myalgo_ctx_t myalgo; /* Add the new algorithm's state struct */
} profile_shc_t;
```

Listing B.6. Updating the cryptographic context union in crypto.h

B.3.2 Step 2: Update the Abstraction Routing

Next, open the main cryptographic abstraction implementation at `libmars/src/crypto/crypto.c`. This file contains the wrapper functions (`CryptHashInit`, `CryptSign`, etc.) that the MARS dispatcher calls.

Add a new `case` block to the `switch` statements inside these wrappers, mapping the emulator's generic calls to your specific external library functions.

```c
void CryptHashInit(profile_shc_t *hctx) {
    switch(profile_get(PROFILE_ALG_HASH)) {
        case PROFILE_ASCON:
            AsconHashInit(&hctx->ascon);
            break;
        case PROFILE_MYALGO:
            /* Route to the new algorithm, passing its specific union member */
            MyAlgo_Init(&hctx->myalgo);
            break;
        default:
            break;
    }
}

void CryptHashUpdate(profile_shc_t *hctx, const void *msg, size_t n) {
    switch(profile_get(PROFILE_ALG_HASH)) {
        case PROFILE_ASCON:
            AsconHashUpdate(&hctx->ascon, msg, n);
            break;
        case PROFILE_MYALGO:
            MyAlgo_Update(&hctx->myalgo, msg, n);
            break;
        default:
            break;
    }
}
```

Listing B.7. Updating the abstraction routing in crypto.c

Note: You must repeat this process for `CryptHashFinal`, `CryptSign`, `CryptVerify`, and the key derivation functions (`CryptSkdf`, `CryptAkdf`), depending on which primitives the new suite provides.

B.3.3 Step 3: Link the External Library

To ensure the compiler can resolve the new functions (e.g., `MyAlgo_Init`), the external cryptographic library must be linked during the build process. Update the `libmars/Makefile` to link against the new library (e.g., adding `-lmyalgo` to the compiler flags).

B.3.4 Step 4: Expose the Algorithm in `mars-setup`

For a user to instantiate a virtual MARS device utilizing the new algorithm, the `mars-setup` configuration utility must be updated. Because the utility parses arguments dynamically based on an array mapping, this process is incredibly straightforward.

Navigate to the `mars-emulator` repository and open the source code for the setup utility (`src/mars-setup.c`). Add the string representation of your new algorithm to the `profile_alg_names` array, matching the index of the enum you defined in Step 1:

```c
/* Inside src/mars-setup.c */
const char *profile_alg_names[] = {
    [PROFILE_NULL] = "NULL",
    [PROFILE_ASCON] = "ASCON",
    [PROFILE_MYALGO] = "MYALGO", /* Add mapping for CLI parsing */
};
```

Listing B.8. Updating the mars-setup utility mapping

Because `mars-setup.c` utilizes `strcasecmp` to iterate through this array dynamically during `parse_algorithm()`, no changes to the CLI parsing logic (`--hash`, `--sign`, etc.) are required.

Once recompiled, users can immediately provision a new device using the new suite:

```
build/mars-setup ./ --PS 1234567890abcdef --pcr 4 --hash myalgo
```

Listing B.9. Generating a profile with the new algorithm

B.4 Testing and Debugging

Because the MARS Emulator suite spans multiple architectural layers-from user-space client applications down through a kernel-space proxy and back up to a background daemon-debugging requires isolating the specific layer where an error occurs. This section outlines the testing framework and the primary strategies for tracing data across the stack.

B.4.1 Running the Test Suites

Whenever a new command or cryptographic algorithm is added, developers should run the provided test suites to ensure no regressions have been introduced into the core logic.

Each repository typically includes its own set of tests. For example, the `vmars_driver` repository contains a dedicated `test/` directory with scripts to validate the proxy device's concurrency and file operations.

Conversely, because `libmars` acts as a foundational backend library, it does not include a standalone unit test suite. Instead, the core cryptographic logic and the daemon's dispatching capabilities are validated by compiling and running the example client applications provided in the `mars-emulator` repository. By executing these examples against a running daemon (as detailed in Section A.3.1), developers can simulate real-world API usage and verify that the backend accurately processes the modified commands.

B.4.2 Debugging the Kernel Driver (`vmars_driver`)

The kernel driver acts as the synchronization arbiter. If an application hangs indefinitely, or if the emulator fails to attach to the control node (`/dev/vmarsx`), the driver's internal logs are the first place to investigate.

The driver makes extensive use of the `pr_info` and `pr_err` kernel logging macros, prefixing all messages with `vmars_driver:`. You can view these logs in real-time using the `dmesg` utility.

```
# View the last few kernel messages related to the driver
dmesg | grep vmars_driver

# Tail the kernel log in real-time (useful during active debugging)
sudo dmesg -w | grep vmars_driver
```
Listing B.10. Tailing the kernel logs for vmars_driver events

Common driver log events to look for include:

- `vmars_driver: successfully created device`: Verifies that the `mars-emulator` daemon successfully issued the IOCTL to create an anonymous file descriptor.
- `vmars_driver: vmars opened / closed`: Indicates when a client application successfully acquires or releases the hardware lock.
- `vmars_driver: proxy device not opened` or `invalid size`: Usually indicates that the emulator daemon crashed or that the client application passed a malformed buffer size.

B.4.3 Debugging the Emulator Daemon (`mars-emulator`)

When testing new commands, it is highly recommended to run the `mars-emu` executable directly in the foreground of a terminal rather than as a background daemon. This allows you to immediately see any `fprintf(stderr, ...)` output generated by the daemon or `libmars`.

If a command fails at the daemon level, it is usually due to one of two issues:

1. **Profile Mismatch:** The requested cryptographic algorithm (e.g., in a `MARS_Sign` command) does not match the algorithms configured in the active profile generated by `mars-setup`.
2. **Dispatcher Unmarshalling Failure:** The dispatcher was unable to parse the CBOR payload.

B.4.4 Common Development Pitfalls

When extending the emulator, developers frequently encounter the following specific issues:

1. CBOR Format String Mismatches

The most common cause of a `MARS_RC_IO` error is a mismatch between the marshalling string in the `mars-api` and the unmarshalling string in the `libmars` dispatcher.

- Ensure that fixed-length strings (`X`) and variable-length strings (`x`) are used symmetrically.
- **Important:** When the dispatcher unmarshals a string (`x` or `X`), the `tinycbor` library requires the length variable passed to `mu_unmarshal` to be pre-initialized with the buffer's maximum capacity (e.g., `size_t len = BUFFER_LEN;`). If this is initialized to `0`, the library will silently fail the decoding process, assuming the destination buffer is too small.

2. Missing Device Locks

If a custom user-space application immediately fails without sending data to the emulator, ensure the thread has requested exclusive access. The `mars-api` commands internally check `mctx_locked()`; if `MARS_Lock()` was not called first, the API will abort and return `MARS_RC_LOCK`.