THESIS

PREVENTING MALICIOUS MODIFICATIONS TO FIRMWARE USING HARDWARE ROOT OF TRUST (HROT)

Submitted by

Rakesh Podder

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2025

Master's Committee:

Advisor: Indrajit Ray
Co-Advisor: Sarath Sreedharan

Indrakshi Ray
Anura Jayasumana

ABSTRACT

PREVENTING MALICIOUS MODIFICATIONS TO FIRMWARE USING HARDWARE ROOT OF TRUST (HROT)

As computing devices such as servers, workstations, laptops, and embedded systems are transported from one site to another, they are susceptible to unauthorized firmware modifications. Additionally, traditional over-the-air (OTA) firmware update mechanisms often lack robust security features, exposing devices to threats such as unauthorized updates, malware injection, etc. While the industry has made efforts to secure the boot process using a hardware root of trust (HRoT), post-boot firmware tampering remains a significant risk.

In this work, we introduce a comprehensive framework that addresses firmware security across both transit and remote update phases by leveraging HRoT and cryptographic techniques. To prevent unauthorized firmware modifications during device shipment, we propose the PIT-Cerberus (Protection In Transit) framework, which enhances the HRoTs attestation capabilities to securely lock and unlock BIOS/UEFI. In addition, we introduce the Secure Remote Firmware Update Protocol (S-RFUP) to fortify OTA firmware updates by incorporating industry standards such as Platform Level Data Model (PLDM) and Management Component Transport Protocol (MCTP). These standards enable interoperability across diverse platforms while reducing management complexity. The protocol enhances security and operational integrity during updates, ensuring that only authenticated and verified firmware modifications occur.

Both frameworks are implemented within a trusted microcontroller as part of Project Cerberus, an open-source security platform for server hardware. We present a security analysis, implementation details, and validation results, demonstrating the effectiveness of our approach in securing firmware both in transit and during remote updates.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Dr. Indrajit Ray, for his invaluable guidance, unwavering support, and insightful mentorship throughout this journey. His expertise and encouragement have been instrumental in shaping this work. I am also profoundly grateful to my co-advisor, Dr. Sarath Sreedharan, whose guidance and insightful feedback have helped refine my research and expand my understanding. Additionally, I extend my heartfelt thanks to Dr. Indrakshi Ray for her continuous support, encouragement, and invaluable contributions, which have played a significant role in my academic progress.

I owe immense gratitude to my parents, Goutam Kumar Podder and Lovely Podder, for their unwavering love, sacrifices, and steadfast support. Their belief in me has been a source of strength and motivation throughout this journey. I also want to honor the memory of my late grandfather, Gopal Chandra Podder, whose encouragement and love instilled in me the courage to dream big and strive for excellence.

I would also like to extend my appreciation to my colleagues and co-authors who have contributed their insights, collaboration, and support throughout this research journey. Their intellectual contributions, discussions, and camaraderie have greatly enriched my experience and helped shape this work.

I am sincerely grateful for the generous support provided by the National Institute of Standards and Technology (NIST) under Award Number 60NANB23D152 and the National Science Foundation (NSF) under Award Numbers CNS 2335687, DMS 2123761, and CNS 1822118. Additionally, I would like to acknowledge NIST, the Army Research Laboratory (ARL), Statnett, AMI, NewPush, and Cyber Risk Research for their invaluable contributions in supporting this research. Their funding and resources have played a crucial role in enabling this work.

Furthermore, I extend my heartfelt thanks to my friends and family, whose encouragement, patience, and belief in me have been a source of strength. Their presence has made this journey more fulfilling and meaningful.

DEDICATION

Dedicated to all who strive relentlessly to achieve their dreams.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Firmware (BIOS/UEFI/BMC)

Firmware is the main software that allows a device's hardware to communicate with the operating system [1]. When a computer is turned on, its BIOS (Basic Input/Output System) firmware takes on the task of initializing the hardware by loading the System Management Interrupts, starting the Advanced Configuration and Power Interface and initiating the loading of the operating system. This mode of operation of the device is a high-privilege CPU mode, which stands apart from standard operating system execution modes like protected mode or long mode. The Unified Extensible Firmware Interface (UEFI) boot process, increasingly replacing BIOS, mirrors the conventional BIOS boot procedure's flow. Most platforms [1] based on UEFI starts their boot process with a minimal core block of code. This stage is known as the Security (SEC) phase [2]. It lays the groundwork for a secure boot, acting as a gatekeeper to verify and authenticate the integrity of all subsequent firmware and software that will be loaded onto the system.

The conventional BIOS boot process is a set of procedures that take place when a computer is powered on, resulting in the loading of the operating system. The basic input/output system (BIOS) is a part of firmware that boots the computer by initializing hardware components and providing a basic set of instructions. Figure 1.1 shows a conventional BIOS boot process for x86-compatible systems [3].

- The BIOS boot block is a small code segment that is stored in a protected area of a motherboard's BIOS chip. The boot block's function is to provide a backup method for the BIOS in the event that the main BIOS code becomes corrupted or fails. The BIOS boot block is

[1] We use the term "platform" to mean any computer or hardware device and/or associated operating system, or a virtual environment on which software can be installed and run. Source NISTIR 7698, https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir7698.pdf

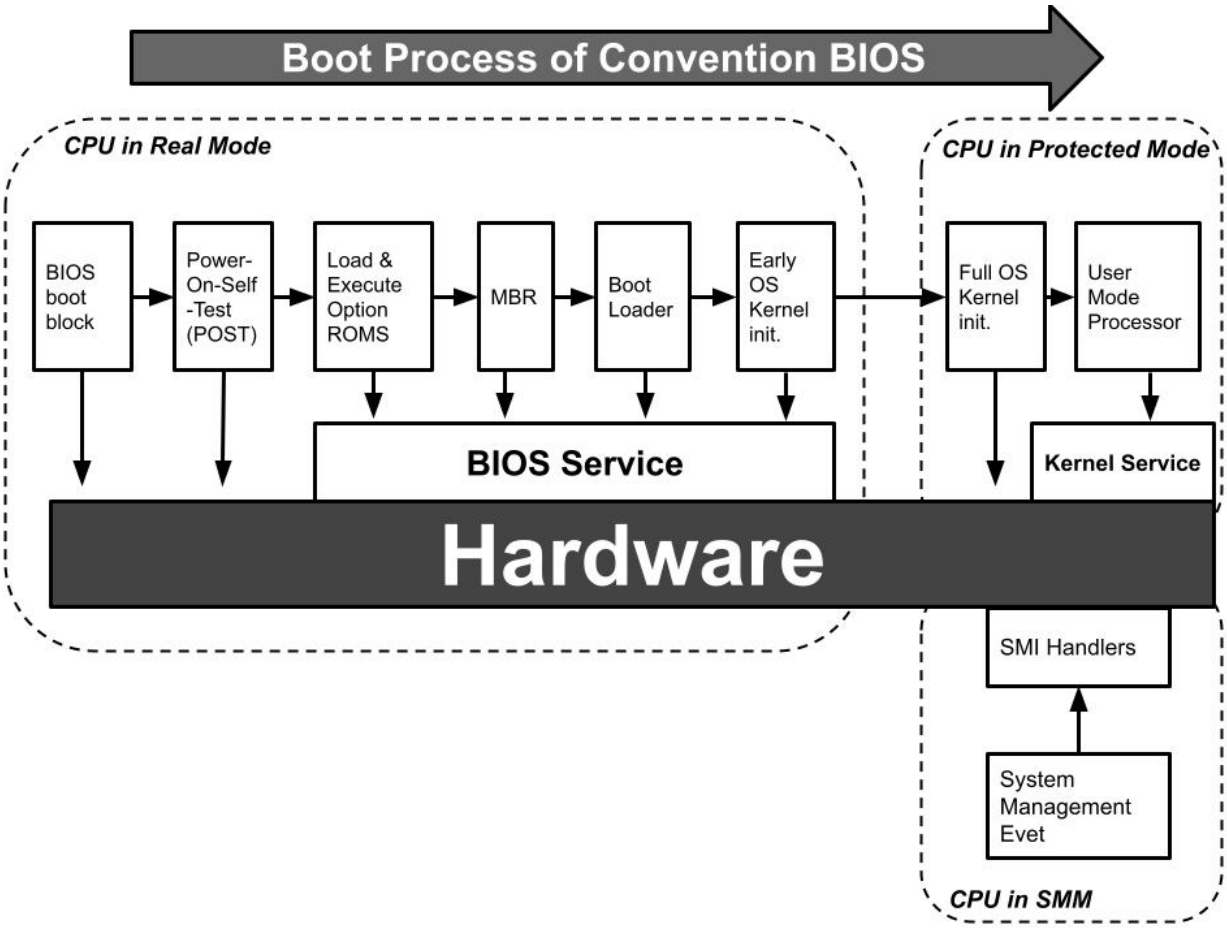

Figure 1.1: Conventional BIOS Boot Process.

the first code to be executed when a computer is turned on. It runs a rudimentary self-test to confirm that the main BIOS code is working properly.

- When the computer is turned on, the BIOS does a self-check to ensure that all important hardware components, such as the processor, memory, and hard drive, are operational. Whenever an error occurs, the BIOS usually emits an audible beep signal or display an error message on the screen.

- The system BIOS then searches for further peripherals and microcontrollers and executes any Option ROMs required to initialize them. Option ROMs execute extremely early in the boot process and can enhance the boot process with a range of features. The Option ROM

on a network adapter, for example, could load the Preboot Execution Environment (PXE), which enables a computer to boot over the network.

- The master boot record (MBR) is a small program that provides information on the partition layout of the hard drive and the location of the operating system boot loader. It is contained in the boot loader code in the first sector of the bootable device.

- The boot loader transfers the operating system kernel into the memory, which is the operating system's core. Following that, the kernel initializes device drivers and other system components.

- After initializing the essential system components, the kernel loads the rest of the operating system into memory and begins executing the first user-space program.

Modern computing systems are also often equipped with baseboard management controllers (BMC) which are specialized processors that monitor the physical state of the machine using sensors and communicate that to system administrators via independent communication channels. BMC firmware is highly privileged and allows for remote management and control, even when the system is shut down. A boot to BIOS, UEFI, a hypervisor or OS is not necessary as the BMC functions even if the server is shutdown.

1.2 Security Concerns

The BIOS/UEFI or the BMC controller are the most critical component of a modern day computing device. Any compromise of the BIOS/UEFI or the BMC firmware code can give an attacker complete control over a system, allowing them to circumvent nearly all higher-level security safeguards. Moreover, once a device has booted up, an attacker can compromise other device functionalities that are present. Unfortunately, firmware attacks can be very difficult to identify and repair [4]. Firmware attacks can be difficult to detect and mitigate, making prevention a paramount concern.

Given these challenges, firmware security must be ensured in two critical aspects: first, while the device is in transit, and second, during the remote update procedure. In the following sections, we discuss the security concerns in each of these aspects.

1.2.1 During Transit

When a device is physically shipped from a manufacturer to a user or transferred between legitimate users, there is a risk of firmware tampering before secure boot. We define this concern as "*Protection in Transit – PIT*." Unauthorized firmware modifications typically require either writing to the firmwares storage location or physically replacing it. Our concern is specifically with unauthorized writes, which require booting the device to a minimal state where firmware memory I/O is enabled.

1.2.2 During Remote Update

Like any other software, firmware needs periodic updates to address vulnerabilities, enhance performance, and introduce new features. Traditionally, firmware updates had been conducted locally. However, as devices grow in complexity (think of cloud servers) and scale (think Internet of Things (IoT)) significant challenges arise for local firmware updates, such as physical access requirements, operational downtime, and logistical complexities.

1.3 Problems with Existing Security Solutions

Professionals have proposed a range of mitigation strategies to address the numerous security concerns surrounding firmware during transit or remote updates. Hardware-based trusted computing utilizes Trusted Execution Environments (TEE) and secure elements like the Trusted Platform Module (TPM) [5] to enhance firmware security. TPMs, which have been included in computers for over a decade, establish a root of trust in a secure cryptographic core, protecting keys and credentials even if the OS is compromised. The latest standard, TPM 2.0, is present in most modern computers and supports various elliptic curve signature schemes, essential for certain core

security services [6]. However, vulnerabilities like CVE-2020-10713 in the GRUB2 bootloader and others in bootloaders from Eurosoft, New Horizon Datasys, and CryptWare have shown that attackers can bypass Secure Boot to execute malicious code. These vulnerabilities can be somewhat mitigated by blacklisting the affected bootloaders in the UEFI Secure Boot Forbidden Signature Database (DBX), which can be updated via UEFI firmware updates or Windows Update. Beyond bootloaders, attackers can target deeper UEFI components, exploiting specific vulnerabilities for targeted attacks. Recent research has uncovered numerous high-impact vulnerabilities (CVE-2022-28858, CVE-2022-36372, CVE-2022-32579, CVE-2022-27493 and CVE-2022-33209, CVE-2022-23930, CVE-2022-31644, CVE-2022-31645, CVE-2022-31646, CVE-2022-31640 and CVE-2022-31641) in UEFI firmware components, indicating an ongoing risk despite advancements in UEFI security technologies [7].

The latest TPM chips support Windows Secure Boot, ensuring the OS only boots if its hashes match those in the chip, preventing rootkit interference. However, TPM doesn't ensure complete security against keyloggers or phishing attacks [8]. Microsoft's installation criteria for Windows 11 include a mandatory TPM 2.0 module to enhance security features. Despite this, new Registry modifications [9] have been identified, enabling users to circumvent not only the TPM 2.0 stipulation but also the minimum memory and secure boot prerequisites for the operating system.

Whereas, Remote Firmware Update (RFU) is emerging as a viable solution for firmware update, enabling updates to be deployed over-the-air (OTA) [10], minimizing disruptions and eliminating the need for physical proximity. However, over-the-air (OTA) update mechanisms, while convenient, are often targeted by attackers such as replay attacks [11], denial of services [12], legacy firmware updates [13], tampering [14], fake and malicious updates [15], and eavesdropping [16]; these attacks allow adversaries to tamper with the firmware, execute arbitrary code or roll back the firmware version to expose prior vulnerabilities [17, 18]. Recent vulnerabilities in the update mechanisms of Jeep Cherokee [19], Samsung SmartThings Hub[2], and Asus Router[3] high-

[2]CVE-2018-3926: https://nvd.nist.gov/vuln/ detail/CVE-2018-3926

[3]CVE-2021-3166: https://nvd.nist.gov/vuln/ detail/CVE-2021-3166

light these concerns. Ensuring the security of RFU protects the integrity of the firmware, maintains device functionality, and safeguards sensitive information contained within these devices. This is particularly crucial in industries where compromised firmware could lead to severe operational disruptions.

While each firmware security solution offers a unique defense mechanism, none are entirely foolproof, and a combination of methods may be required to achieve comprehensive firmware security. As the attack surface for firmware continues to expand, particularly with the rise of remote updates and connected systems, it is crucial to adopt a defense-in-depth approach to protect these critical software components from unauthorized modifications, tampering, and exploitation

1.4 Research Questions

To address these security concerns and problems with remote firmware updates and in transit, we explore the following research questions:

- RQ1: What specific threats exist in transit and remote firmware update mechanisms, particularly in over-the-air (OTA) updates, that could be exploited to compromise device integrity?
- RQ2: How can we ensure the integrity and security of a device's firmware during transit from the vendor to the end user, preventing unauthorized modifications and malware installation?
- RQ3: How can we secure and standardize remote firmware updates across diverse device platforms while minimizing security vulnerabilities?
- RQ4: To what extent do the proposed Protection in Transit (PIT) and Secure Remote Firmware Update Protocol (S-RFUP) enhance security and standardization during transit and remote firmware updates, and how can their effectiveness be empirically validated?

1.5 Contributions

This thesis makes several key contributions to the field of firmware security, particularly in the context of protection in transit and remote firmware updates.

We realize the "Protection in Transit" objective by having the device manufacturer implement a BIOS or BMC lock post-production and introduce a mechanism for user authentication by the device before self-unlock to ensure its use is only by the rightful authorized user [20, 21]. Manufacturers of modern computing devices frequently use a tamper-proof micro-controller [22] as a hardware root of trust for a verified and reliable machine boot-up following software attestation principles. We employ such a hardware root of trust (HRoT) in the BIOS/UEFI or BMC boot process to perform user authentication, locking, and unlocking of the device. The locking of the device is executed at the BIOS or BMC level and the HRoT triggers the unlock after successful authentication. Building upon the foundations laid by Project Cerberus [23], an open-source initiative aimed at establishing a hardware root of trust (HRoT) for server platforms, we enhance Cerberus capabilities to carve out a more resilient security protocol. While Cerberus effectively orchestrates secure attestations for firmware on devices, it does not inherently possess lock/unlock or user authentication functionalities. Our implementation of the PIT-Cerberus framework incorporates this lock/unlock mechanism tied to an authentication protocol, allowing for more stringent control over the BIOS or BMC even before the boot-up phase. This is pivotal in preventing unauthorized access during a device's transit and ensures that the integrity and security of the firmware remain uncompromised from the point of departure to the point of receipt. AMI International, an industry leader in BIOS/BMC firmware, is currently beta-testing the framework and plans to incorporate it in their production lines. We are in the process of open-sourcing our extended PIT-Cerberus library, confident in its ability to bolster the security landscape.

Once the device is successfully unlocked, we enforce stringent security measures and standardized firmware update protocols to ensure integrity and consistency throughout the firmwares lifecycle. The proposed Secure Remote Firmware Update Protocol (S-RFUP) systematizes the update process and provides needed security features to protect against tampering, unauthorized firmware rollback, and intellectual property (IP) theft [24]. S-RFUP ensures compatibility across a diversity of devices and reduces complexity, which, in turn, also enhances the overall security posture by minimizing the inconsistencies that can be exploited in an attack. S-RFUP leverages

Project Cerberus' paradigm of hardware root of trust [25] to enable secure attestation, ensuring that all firmware and software boot processes are verified and secure and making it an ideal foundation for secure RFU. We integrate industry standards such as Platform Level Data Model (PLDM), and Management Component Transport Protocol (MCTP) with Project Cerberus for standardizing remote management and monitoring of firmware updates. The proposed S-RFUP framework leverages strong cryptographic techniques, such as AES-256 and ECDH Key Exchange, to encrypt and decrypt messages during communication between Update Agent (UA) and Firmware Device (FD).

The main contributions of this work can be summarized as:

- We developed PIT-Cerberus framework (PIT = Protection in Transit) protocol to prevent device tampering during transit. It leverages strong cryptographic techniques and has been implemented within a trusted microcontroller framework (Project Cerberus).

- We present a novel framework (S-RFUP) that integrates industry-standard protocols (PLDM, MCTP) with a hardware root of trust (HRoT) to ensure interoperable and secure RFU across multiple device platforms.

- We extend Project Cerberus by introducing new libraries ('PIT & S-RFUP) to perform lock-/unlock mechanisms on firmware and to handle PLDM message construction, PLDM over MCTP binding, and encryption for an end-to-end secure pipeline and standardization of RFU accordingly.

- We perform empirical evaluations and security analyses to validate the functionality and protocol's exception and failure handling capabilities in various attack scenarios to demonstrate robustness against known vulnerabilities.

- The PIT-Cerberus and S-RFUP frameworks libraries, APIs, and documentation have been open-sourced to Project Cerberus.

1.6 Document Organization

The remainder of this document is structured as follows:

- Chapter 2 provides background information on firmware specifications and industry standard solutions, like HRoT, PLDM, MCTP, etc.
- Chapter 3 discusses existing firmware security threats like unauthorized firmware modifications, unauthorized updates, malware injection, etc., and the drawbacks of existing solutions.
- Chapter 4 presents the threat model, approach, implementation, evaluation, and security analysis of the PIT-Cerberus framework.
- Chapter 5 details the Secure Remote Firmware Update Protocol (S-RFUP) methodology, threat model, implementation, evaluation, and security analysis.
- Chapter 6 concludes with key findings from the research and future research directions.

Chapter 2

Background

2.1 Hardware Root of Trust (HRoT)

Hardware Root of Trust (HRoT) refers to a secure, tamper-resistant hardware component that serves as the foundation for verifying the integrity and authenticity of firmware and system components. It is implemented as a secure microcontroller or cryptographic module embedded within a devices hardware. Figure 2.1 illustrates how the HRoT is incorporated in a convention boot process.

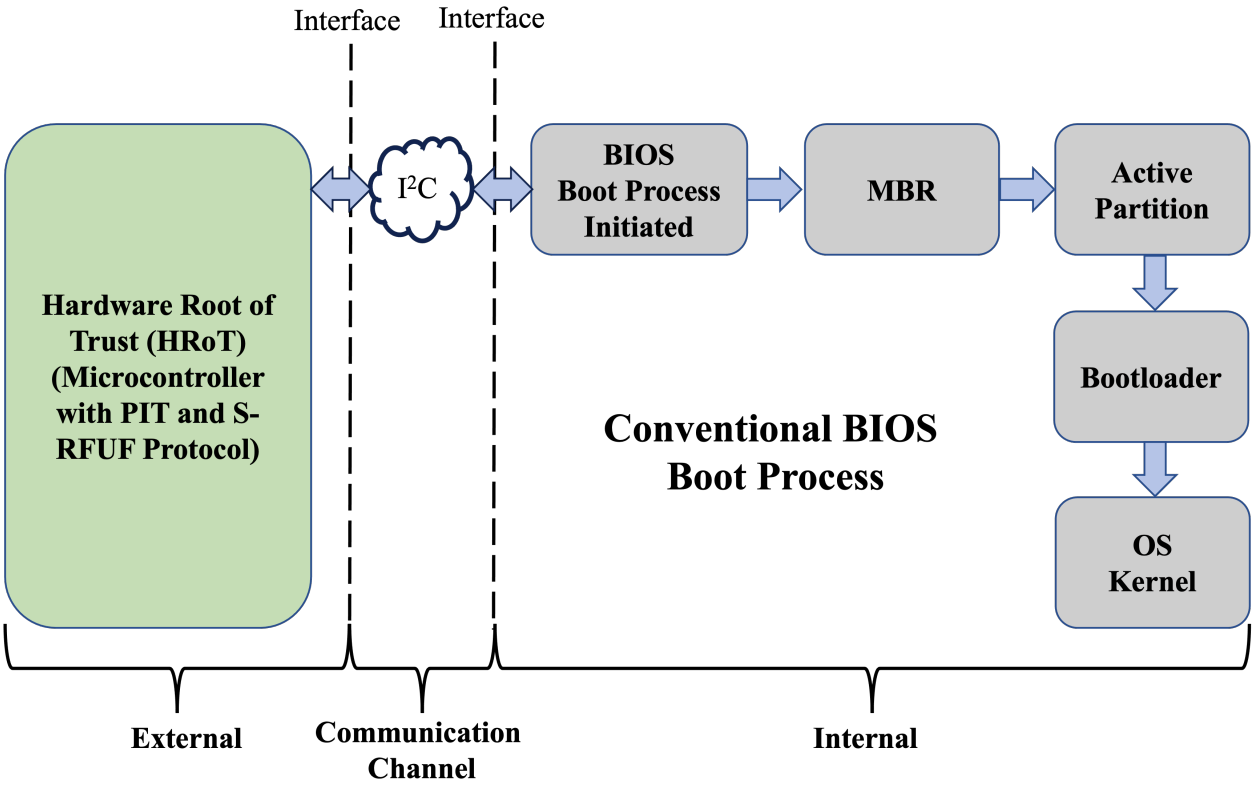

Figure 2.1: HRoT-based Boot Process.

HRoT in PIT-Cerberus and S-RFUP

- In **PIT-Cerberus**: The HRoT is used to lock and unlock the BIOS/UEFI or BMC firmware. The device remains locked until an authorized user successfully authenticates. This prevents unauthorized modifications during device transit.

- In **S-RFUP**: The HRoT ensures secure firmware updates by providing cryptographic attestation. It verifies the authenticity of firmware updates, ensuring they have not been tampered with before installation.

By leveraging HRoT, both frameworks provide strong security guarantees against unauthorized firmware modifications, ensuring trustworthiness from manufacturing to deployment.

2.2 Microsoft Project Cerberus

Project Cerberus [25] is developed by Microsoft as a hardware root of trust (HRoT) specifically for server platforms. It enables secure boot functionality for device firmware, whether or not the devices inherently support secure boot. Additionally, it offers a secure method to verify and attest to the firmware state of the devices. We are using Project-Cerberus as a server-platform for Update Agent (UA) and as a HRoT for Firmware Device (FD).

2.3 Firmware Base Specifications

2.3.1 Platform Level Data Model (PLDM) Base Specification

The Platform Level Data Model (PLDM) Base Specification is a standardized protocol designed to facilitate efficient communication and management within platform management subsystems. The specification outlines the fundamental elements and message formats necessary to support a wide range of platform-level management tasks. The primary purpose of the PLDM Base Specification is to establish a common protocol for monitoring and controlling various platform components, such as sensors, firmware, and hardware subsystems. This includes inventory management, event notifications, control functions, and data transfer operations. By providing a stan-

dardized framework, PLDM ensures interoperability between different management controllers and devices, thereby simplifying the integration and management of heterogeneous systems.

Figure 2.2 illustrates the fields that constitute a generic PLDM message. These fields are transferred starting from the lowest offset. The PLDM Completion Code appears exclusively in PLDM response messages.

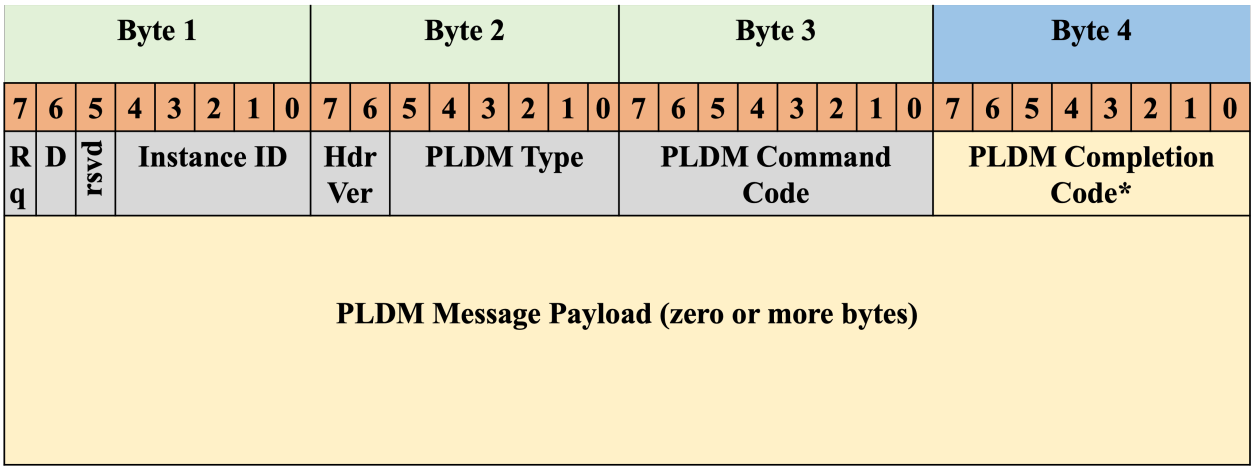

Figure 2.2: Generic PLDM Message Fields.

2.3.2 Management Component Transport Protocol (MCTP) Base Specification

The Management Component Transport Protocol (MCTP) Base Specification is a comprehensive communication model designed to facilitate interactions between management controllers and between management controllers and management devices within a platform. MCTP establishes a standardized protocol that can be implemented across various physical transport mediums, enabling flexible and robust platform management solutions.

MCTP provides a framework that includes a message format, transport descriptions, message exchange patterns, and initialization and configuration messages. This protocol is designed to be independent of the underlying physical bus properties and the data-link layer messaging used on

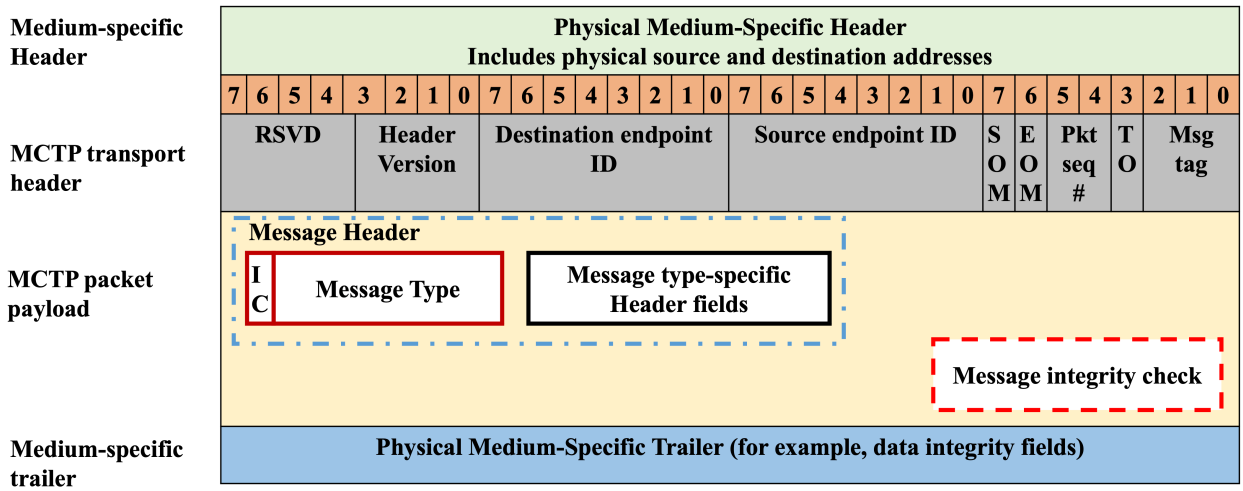

Figure 2.3: Generic MCTP Message Fields.

the bus. This abstraction allows MCTP to be implemented over different transport bindings such as PCIe, SMBus/I2C, and potentially other mediums like USB and RMII in the future.

Figure 2.3 illustrates a generic MCTP packet fields. The MCTP base protocol specifies the common fields for MCTP packets and messages and their usage. While there are medium-specific packet header fields and trailer fields, the fields for the base protocol are consistent across all media. These common fields facilitate the routing and transport of messages between MCTP endpoints and the assembly and disassembly of large messages into and out of multiple MCTP packets. The base protocol's common fields include a message type field, which identifies the particular higher layer class of message being carried by the MCTP base protocol.

2.3.3 PLDM over MCTP Binding Specification

PLDM over MCTP Binding Specification [26] outlines how PLDM messages are transported over MCTP, establishing a common format and ensuring interoperability between different hardware and software components within a platform management subsystem. The main objective of the *PLDM over MCTP Binding Specification*is to establish the message format and protocol requirements for transmitting PLDM messages via the MCTP transport protocol. Utilizing MCTP as the transport mechanism allows PLDM to efficiently manage and monitor platform components through a standardized communication model.

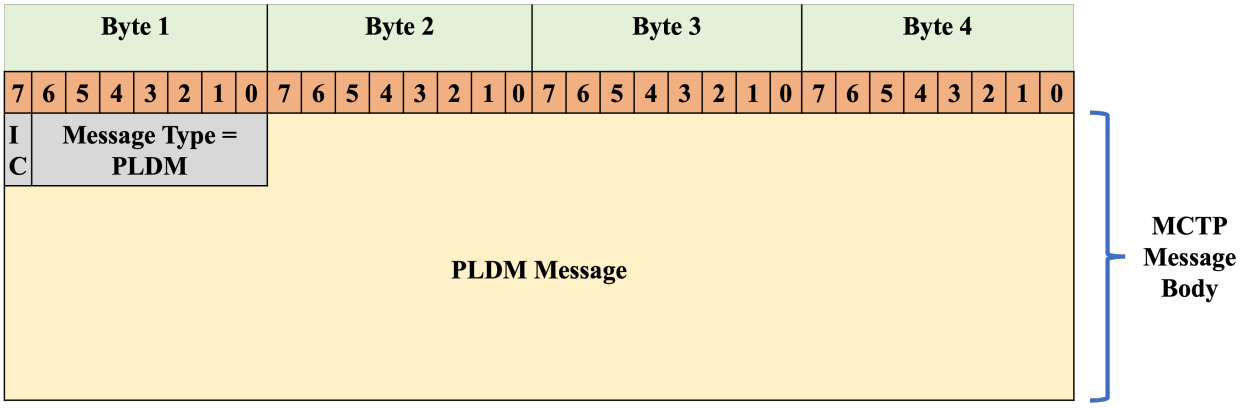

Figure 2.4: PLDM over MCTP Message Fields.

This specification defines how the platform-level data models and platform functions are implemented using MCTP communications. PLDM is supported as a message type over MCTP. PLDM over MCTP binding defines the format of PLDM over MCTP messages. Figure 2.4 shows the fields of an MCTP message body carrying a PLDM message. This approach to firmware updates utilizes a unified, standardized method that doesn't depend on operating system-specific tools from individual manufacturers. Instead, it sends $pldm$ messages to directly transfer firmware component images to the FD. The updates can be delivered as a single combined image, individual component images, or multiple images for various components of a device, managed through an offset-based method by the FD. The update process involves the UA recognizing the FD using unique identifiers and descriptors, such as PCI Vendor ID and Device ID. The component images are organized by function type and identified with a unique vendor-specific identifier, possibly including a comparison stamp, to ascertain if an update is needed based on the version.

This introduces a unified approach to firmware updates that does not rely on traditional, operating system-based tools provided by individual device manufacturers. Instead, it facilitates the transfer of firmware component images directly to a FD using PLDM messaging, thus standardizing and simplifying the firmware update process across different devices. The mechanism allows for the transfer of either a single combined image, individual images for specific firmware components, or multiple images for various components of the same device. This transfer is conducted through an offset-based method managed by the FD. The firmware update process itself involves

the UA communicating with the FD, identifying it via unique identifiers and descriptors, and using this information to correctly apply the firmware update. The component images in the update package are classified by function type and identified by a unique vendor-specific identifier, which may include a comparison stamp to help in determining if the firmware needs an update based on versioning. This standardized approach not only streamlines firmware updates across different devices but also ensures a more coordinated and error-free update process.

2.4 OpenBMC/libpldm

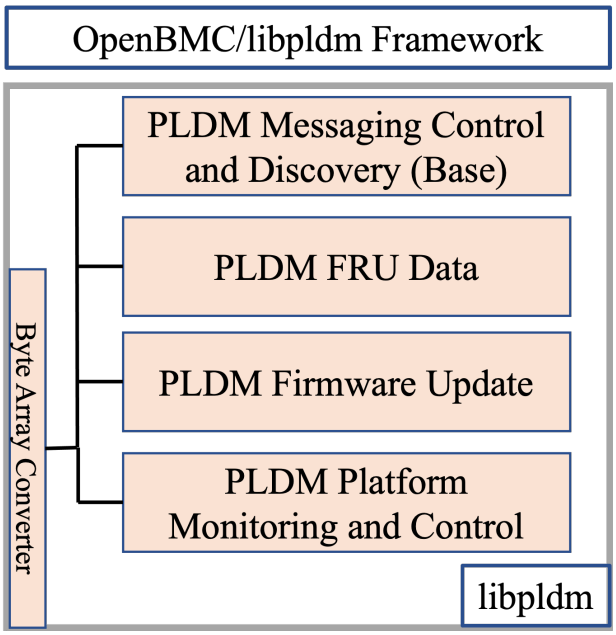

Figure 2.5: OpenBMC/libpldm library Framework.

This library is part of the OpenBMC project, aimed at providing an open-source firmware solution for baseboard management controllers (BMCs). The '*libpldm*' deals with the encoding and decoding of PLDM messages. Fig. 2.5 shows various core module to facilitate tasks such as firmware updates, monitoring, and control of hardware devices across different hardware platforms.

Chapter 3

Literature Review

Several potential attacks against conventional BIOS and EFI/UEFI firmware have been demonstrated by security researchers under laboratory conditions. Sacco and Ortega discuss the topic of BIOS infections and describe a proof-of-concept demonstration of a persistent BIOS malware infection [4]. According to the authors, detecting and eradicating BIOS infections is challenging because they reside in a section of the computer's memory that is inaccessible to standard antivirus or anti-malware software. Wojtczuk and Rutkowska have presented research on the security of Intel BIOS and described several vulnerabilities and attack vectors [27] that can be utilized to attack the BIOS such as exploiting firmware vulnerabilities, accessing the BIOS via hardware debugging tools, and employing software-based assaults such as buffer overflow attacks. A new type of malware known as System Management Mode (SMM) rootkits [28] has been introduced by Duflot and Pornin. In this paper, the authors explain the nature of SMM, its role in computer hardware, and the vulnerabilities that can be used to get access to SMM.

Firmware level malware has been somewhat more common in embedded systems. Using the HP-RFU vulnerability in LaserJet printers as a case study, A. Cui, M. Costello, and S. Stolfo demonstrate the development of a proof-of-concept printer malware capable of network reconnaissance, data exfiltration, and propagation to other devices [17]. They highlight the widespread nature of vulnerable embedded devices and the challenges in patching them, with only a small percentage of the vulnerable population being patched. The paper also emphasizes the limitations of firmware update signing and identifies vulnerabilities in third-party libraries found in firmware images. Overall, the findings underscore the need for effective host-based defense mechanisms to protect vulnerable embedded systems.

PsycoB0t [29], a notable router botnet, infiltrated the firmware of about 85,000 DD-WRT home routers, turning them into instruments for conducting severe network-disrupting Distributed Denial of Service (DDoS) attacks. Barnaby Jack demonstrated the unauthorized extraction of cash

from ATMs through the modification of their firmware, a technique infamously known as “jackpotting” [30]. Charlie Miller uncovered serious risks within the firmware of certain Apple laptop batteries that could potentially lead to malfunctions or overheating [31]. Employing PostScript [32], a ubiquitous language in electronic and desktop publishing, Costin showcases the susceptibility of certain Lexmark printers to memory inspection and arbitrary modifications, which can lead to exposure of sensitive data or disruption of device functionality. Kevin Fu’s groundbreaking work in medical device security highlights the perilous reality of exploiting embedded devices [33], with his real-world attacks on an implantable cardioverter defibrillator and an automated external defibrillator illuminating these life-threatening vulnerabilities. These instances underscore the crucial need for robust firmware security across diverse devices, as firmware forms the base code controlling hardware, making successful infiltration by an attacker profoundly dangerous.

The integrity of the system BIOS is critical for ensuring the security and functionality of computer systems, particularly during their transit through the supply chain. Unauthorized modifications, such as malicious firmware updates and firmware modifications, pose significant threats [34]. Malicious actors exploit vulnerabilities or weak security measures to alter the BIOS, introducing malware or Trojans that can lead to data breaches, device malfunctions, or system takeovers [35]. These threats are exacerbated by man-in-the-middle attacks [36] and supply chain attacks [37], which intercept or tamper with firmware updates or implant malware, respectively. Such attacks exploit the low-level operation of firmware, making detection and mitigation difficult, and represent a persistent security challenge [3, 38–42].

Ian Haken shows how attacks bypass BitLocker by exploiting systems without pre-boot authentication and using a mock domain controller [43]. It requires the target machine to be domain-joined and have had a domain user log in previously. The attacker sets up a fake domain, tricks the system into accepting a new password due to an "expired" password scenario, and corrupts the local credentials cache. This allows offline login with the new password, giving the attacker access to all user data and the ability to install malware. The method is simple, requires physical access, and quickly circumvents BitLocker without complex tools, representing a serious security risk.

Security researchers have demonstrated several concerns with conventional remote BIOS and EFI/UEFI firmware updates [11–16, 44–46]. In the following, we discuss some of these works and the limitations of proposed solutions.

The integrity of the system BIOS is essential for the security and operational reliability of computer systems, especially for critical systems such as cloud servers, healthcare devices, etc. Unauthorized firmware modifications, including malicious firmware updates and alterations, present serious risks [34]. These unauthorized modifications can occur when attackers exploit vulnerabilities or insufficient security practices during a remote firmware update process to manipulate the BIOS, potentially introducing malware or Trojans that compromise data security, disrupt device functionality, or enable system hijacking [35]. The threat landscape is further complicated by man-in-the-middle attacks (MITM) [36] and supply chain attacks [37], which may intercept or tamper with firmware updates or embed malware. Such attacks leverage the foundational-level operations of firmware, challenging both detection and mitigation efforts, and pose ongoing security concerns [3, 38]. Cui et al. [17] present a proof-of-concept for printer malware that can perform network reconnaissance, extract data, and spread to additional devices. The research points out the inadequacies in the security of firmware update processes and the existence of vulnerabilities within third-party libraries used in firmware.

The widespread adoption of non-secure protocols like HTTP further exposes firmware update processes to potential MITM and backdoor exploits [45, 47]. PsycoB0t [29] is a well-known example of a worm infection during firmware update of a router. Additionally, there have been cases where vulnerabilities in update procedures are exploited to carry out firmware modification attacks [46]. Jack using "jackpotting" [30] demonstrated that unauthorized firmware modification can be done in ATM machines. Costin showcases the susceptibility of certain Lexmark printers to memory inspection and arbitrary firmware modifications by employing PostScript [32].

Both the academic community and the Internet Engineering Task Force (IETF) are working on creating software update mechanisms for Class 1 and Class 2 devices (Upkit [48], [49]) and developing hot-patching techniques such as RapidPatch [50], and Hera [51] that can be potentially

used. However, the emphasis in these protocols is on patching low-capability devices and not necessarily on the security of the same. There are some works by researchers on designing secure firmware updates [16, 52–56]. Falas et al. proposed a Public PUF model [57] for secure firmware updates, but its dependency on high computational resources and complex key management limits its feasibility for low-power IoT devices and presents scalability and standardization challenges in large-scale deployments. The OTA firmware update mechanism proposed by Frisch et al. [58], requires manual intervention to update the framework version and rebuild the firmware, potentially leading to higher latency in updates when API changes occur. Also, the method is unable to verify the cryptographic signature before writing the firmware to flash, potentially allowing corrupted or malicious updates to be partially written before being detected. Similarly, the proposed Narrowband IoT (NB-IoT) as the wireless communication standard for firmware updates by Mahfoudhi et al. [59], does not implement any support for firmware authenticity and confidentiality and thus can be susceptible to legacy firmware updates and MITM attacks. The current state-of-art protocols fail to provide a high-level security with standardization in firmware updates across diverse device ecosystems. The research question (RQ1) is addressed here by identifying and analyzing the attack surface for both firmware transit and remote OTA updates, highlighting vulnerabilities such as unauthorized firmware modifications, man-in-the-middle attacks, replay attacks, rollback attacks, and malicious firmware injections, which exploit weak authentication, lack of encryption, and non-standardized update protocols, ultimately compromising device integrity. Our proposed S-RFUP can provide firmware integrity, and a streamlined and standardized operational functionality to secure sensitive information during remote firmware update.

Chapter 4

Protection in Transit

4.1 Threat Model

In this section, we describe the threat model. We begin by identifying the key entities within the PIT-Cerberus framework. The Hardware Root of Trust (HRoT) is a critical component that leverages the Project Cerberus embedded framework. Devices such as laptops, workstations, and commercial servers, which integrate BIOS/BMC functionalities, constitute the DEVICE. The HRoT together with the DEVICE comprise the PRODUCT that is shipped to the USER. The USER is defined as an authorized individual with access rights to the DEVICE. Additionally, the COMPANY Server (or simply the SERVER), a part of the COMPANY's PIT-Cerberus framework infrastructure, is responsible for overseeing the DEVICE's locking and unlocking mechanisms.

4.1.1 Assumptions

We assume that the HRoT processor is tamper-proof and trusted. The COMPANY is trusted and there is no malicious insider threat at the COMPANY. The COMPANY programs the HRoT processor with the PIT-Cerberus and related libraries and data in a secure manner (Note that any modern computing device that employs a root of trust for software attestation, also makes a similar assumption about the root of trust). We also assume that the SERVER is honest and not curious. Any confidential information that is stored on the SERVER is protected against confidentiality and integrity breaches. We also assume that the HRoT can operate independent of the DEVICE and can setup and sustain communication with SERVER. The USER will need to use a second computing device (for example, a smartphone or a laptop) to send and receive information to/from SERVER and send information to the PRODUCT during initial boot up. We assume that the USER trusts this device.

4.1.2 Attacker Model

An adversary is able to sniff on the communication channel between the HRoT and the SERVER; it can replay messages on this communication channel, tamper with the messages and can also insert messages on this communication channel. The threat model is particularly concerned with the attacker's capabilities, especially during the period when the PRODUCT is in transit from the COMPANY to the boot-up of the DEVICE by the USER. This transit phase is the un-trusted zone. Within this scenario, attackers may attempt to intercept and analyze packets exchanged during the DEVICE unlocking process, potentially gaining access to sensitive information. Replay attacks represent another threat, where attackers could delay or resend packets to mislead the HRoT, USER, or SERVER. Additionally, there is a risk of attackers injecting false information, aiming to disrupt ongoing services. PIT-Cerberus is not designed to protect against such denial-of-service attacks. A significant threat is posed by man-in-the-middle attacks, where an attacker impersonating the SERVER or a legitimate USER could try to compromise the firmware integrity or gain unauthorized access by bypassing security measures.

The PIT-Cerberus framework is not designed to protect the PRODUCT from physical tampering during transit. Such an attack can for example, replace the HRoT with a micro-contoller of the attacker's choosing or embed a hardware Trojan. Such protection is difficult to implement. If needed, we can assume that when a device is shipped from the COMPANY to the USER, it is protected in a package using tamper-proof seals [60] [61]. This type of attacks would not only cause damage to the PRODUCT but also alert the USER, which is contrary to the attacker's objectives.

The Federal Information Processing Standard (FIPS 140-2) [62] outlines four security levels for cryptographic modules, ranging from minimal physical protection at Level 1 to comprehensive environmental safeguards and tamper detection at Level 4. An example of the highest security standards is IBMs 4758 PCI cryptographic adapter [63], which adheres to FIPS 140-1 Level 4, equipped with internal tamper detection and sensors for environmental attacks.

Additionally, we consider the COMPANY Server a trusted zone because the COMPANY employs its own Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), alongside

security policies and personnel, to guard against any form of privilege escalation, information disclosure, or DoS-type attacks [64]. Furthermore, the protocol design and the implementation of various protocols, such as the key size of Advanced Encryption Standard (AES), prime modulus of Elliptic-curve cryptography (ECC), curve selection for ECC, and the Digital Signature Algorithm (DSA), are meticulously chosen and programmed within a Sensitive Compartmented Information Facility (SCIF) at the COMPANY. So, we assume that any form of physical tampering and the analysis of side-channel information, including power, timing, and electromagnetic radiation, are impractical for attackers.

4.1.3 Desired Security Properties

The essential security properties we need to preserve while the firmware is in transit include firmware integrity, confidentiality, and availability.

- **Confidentiality**: Maintains the secrecy of firmware data by protecting sensitive proprietary information from unauthorized access.
- **Integrity**: Ensures that the firmware device has not been tampered with.
- **Availability**: Guarantees that the firmware device will be unlocked only by the authorized user after mutual authenticity.

By addressing these security properties, the protocol (PIT-Cerberus) aims to provide a robust, secure, and reliable process during transit.

4.2 Proposed Approach

Our approach to solving the problem of protection in transit is to implement a BIOS-level lock/unlock facility before the secure boot implemented by a hardware root of trust – HRoT – microcontroller. The same protocol can be implemented at the BMC level and we use the term BIOS lock to also include locking of the BMC firmware. An overall workflow of PIT-Cerberus Protocol is shown in Figure 4.1.

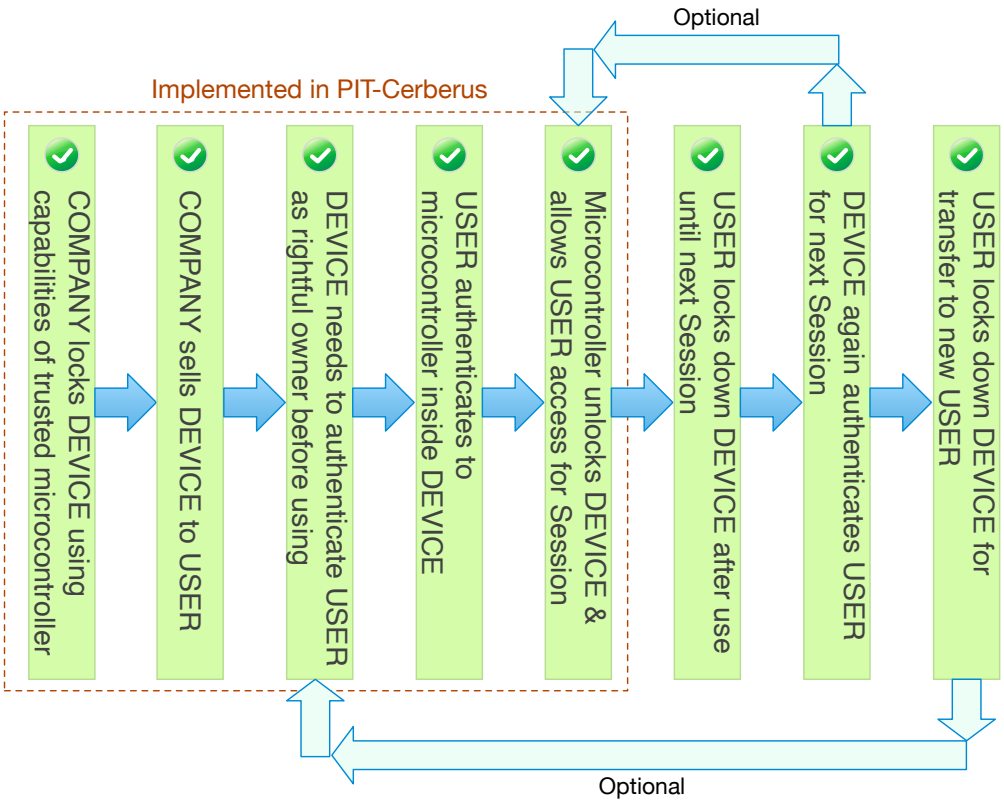

Figure 4.1: High-Level Workflow of PIT-Cerberus Protocol.

This protocol is implemented in the HRoT that is embedded with the DEVICE. Prior to starting the boot-up of the BIOS, the HRoT establishes a secure state, which verifies the user and product information. This process occurs before the full BIOS boot-up and OS loading. We assume that the HRoT processor is able to establish a rudimentary communication with the outside world without relying on the networking capabilities offered by the host machine. The PIT-Cerberus protocol terminates with success if HRoT has the confidence that the correct USER has initiated an unlock request. The HRoT relies on the SERVER to validate if a correct USER has initiated the unlocking process. The PIT-Cerberus protocol implements a novel challenge-response authentication scheme involving the HRoT, USER and SERVER for this validation. One important feature of our protocol is that this authentication does not require the HRoT to be programmed with the legitimate user's identity or shared secret. Thus, if the device is resold, the HRoT can perform the same interaction with the next user (shown as the lower "Optional" flow in Figure 4.1). If the next user is established as the legitimate owner of the device, the user is granted access to boot-up the BIOS. Once veri-

fication is complete, the device turns on, and the BIOS continues with the hardware initialization process. We assume that secure attestation of firmware is performed as usual after the unlock.

In the following, we discuss the PIT-Cerberus protocol in more details.

4.2.1 The PIT-Cerberus Protocol

The protocol in shown in Figure 4.2. There are 4 major entities in the protocol: (i) HRoT: It is the tamper-proof micro-controller that acts as the hardware root of trust. It is the main component that will lock or unlock the BIOS of the DEVICE. PIT-Cerberus protocol is implemented in the HRoT processor, (ii) DEVICE: It is the entity (with BIOS/BMC) being protected in transit via PIT-Cerberus, (iii) USER: USER is the individual authorized to operate the DEVICE. USER has established a trust relationship with the COMPANY in Figure 4.1, and (iv) SERVER (COMPANY Server): It works on behalf of the COMPANY to mediate the HRoT - USER authentication. PRODUCT is the asset that COMPANY shipping to an USER which is a DEVICE with HRoT. The HROT and USER communicate via the internet with the SERVER to lock or unlock the DEVICE. We assume that this channel ensures the authenticity of endpoints and integrity of messages.

The PIT-Cerberus protocol is divided into two main parts: the Locked State and the Unlocked State. Under the Locked State, the HRoT is operational but BIOS has not been loaded and the remaining part of the DEVICE is non-functional in the PRODUCT. The Locked State is further divided into various sub-states.

1. Locked State: The "Locked State" is the initial state of the PRODUCT when it leaves the manufacturer. To put the DEVICE of the PRODUCT to the Locked State, the SERVER creates a unique PRODUCT registration ID, REG.ID, programs it into the HRoT, and also stores it locally. The SERVER also allocates this unique REG.ID to a particular authorised USER. The DEVICE is locked by BIOS or BMC and the DEVICE becomes unresponsive to typical USER inputs. This Locked State has several sub-states that outline the conditions and steps needed to transition the DEVICE into the Unlocked State. Each sub-state serves as a checkpoint in verifying the authenticity of the USER trying to access the DEVICE.

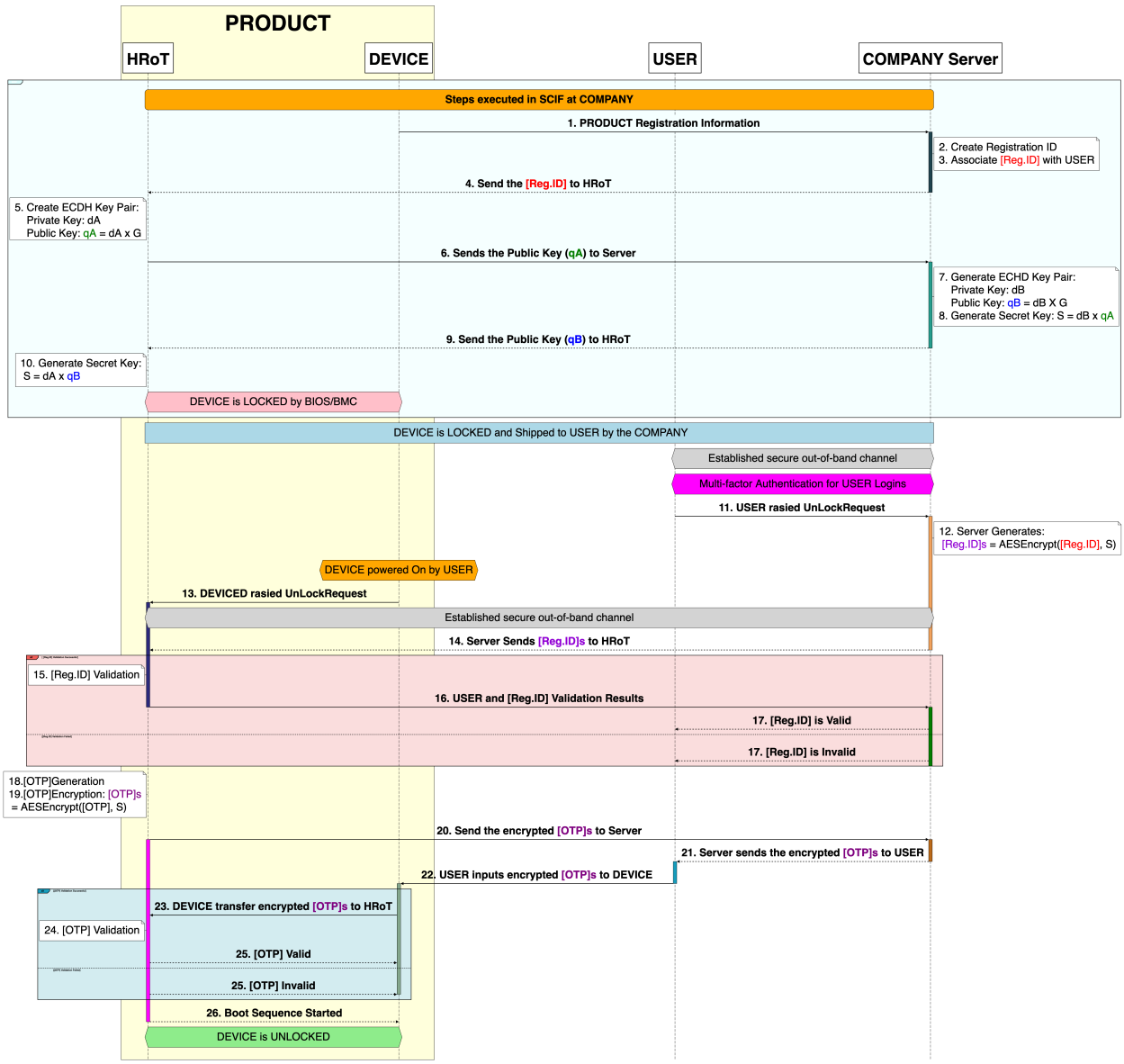

Figure 4.2: PIT-Cerberus Protocol for Locking and Unlocking DEVICE.

When a DEVICE is powered on, by default it is in the Locked State but the HRoT inside the PRODUCT transitions from one sub-state to another.

(a) In the Locked State, the HRoT first enters the Key Generation sub-state in which it first generates an ECC (Elliptic Curve Cryptography) private key (d_A) - public key (q_A) pair, where, $q_A = d_A \times G$. G is the base point of the chosen elliptic curve. The Key Generation sub-state together with the Key Exchange sub-state (described next) forms a Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol [65].

(b) Once the key pair is successfully generated, the HRoT enters the Key Exchange sub-state. In this state, the HRoT sends the public key (Q_A) to the SERVER. The SERVER generates a private key d_B, computes a public key $q_B = d_B \times G$ and an AES secret key $S = d_B \times q_A$. The SERVER transmits its public key to the HRoT, which then computes the same AES secret key S as $S = d_A \times q_B = d_A \times \{d_B \times G\} = d_B \times \{d_A \times G\} = d_B \times q_A$.

(c) At the end of the ECDH protocol, the HRoT enters the Idle sub-state waiting for a USER to initiate an unlock request.

As illustrated in Figure 4.2, all steps of the locking procedure are performed within a Sensitive Compartmented Information Facility (SCIF) at the COMPANY.

2. Unlocked State: The "Unlocked State" represents the phase where the DEVICE, initially in a locked position, is unlocked and allowed to boot fully, enabling the USER to use it as intended. The PRODUCT is shipped to a legit USER and, a secure connection has been established between USER–SERVER and SERVER–HRoT. To raise an unlock request, USER has to log into SERVER with an multi-factored authentication schema.

(a) Once a USER makes an unlock request to SERVER and the DEVICE is powered on, the SERVER first sends REG.ID$_S$ to HRoT. The HRoT then initiated Reg.ID$_S$ Validation state where it decrypts the REG.ID$_S$ and validated against the stored REG.ID. This state validates that the USER is correctly identified by the HRoT. Then the HRoT

enters the OTP Generation sub-state. The HRoT creates a one-time password (OTP) for the USER, and encrypts OTP using AES-GCM [66] with the secret key S to generate OTP_S. The HRoT sends these encrypted value to the SERVER and enters the OTP Exchange sub-state.

(b) The HRoT continues to wait in the OTP Exchange sub-state until it receives another response from the USER.

(c) The SERVER, upon receipt of OTP_S, looks up the correct USER to send the the encrypted OTP_S via an out of band channel (secure email or mobile phone). We assume only the authorized USER has access to this communication facility between the SERVER and the USER.

(d) On receiving OTP_S, the USER enters it into the HRoT, which then enters the OTP Validation sub-state. Upon successful validation of the OTP by the HRoT, it triggers the DEVICE Unlocked State which is nothing but initializing the booting process in the DEVICE.

To manage the boot process of a BIOS/BMC device using a microcontroller such as Microchip CEC1702 as a Hardware Root of Trust (HRoT), the process involves initializing the HRoT upon device power-up to conduct self-tests and initiate secure boot procedures. The HRoT verifies the integrity and authenticity of the USER's legitimacy. If the firmware passes these checks, the HRoT allows the boot process to proceed, handing over control to the BIOS/BMC for hardware initialization and operating system launch. These actions leverage the HRoT's secure boot capabilities to either permit or prevent the device from reaching an operational state, ensuring that only authenticated and integrity-checked firmware can execute, thus providing a robust security measure against unauthorized access.

4.3 Implementation

The PIT-Cerberus protocol has been implemented through an open-source embedded framework, known as Project Cerberus [23]. Originally conceived by Microsoft, Project Cerberus is

a framework for hardware root-of-trust that is compliant with the NIST 800-193 standards, with an unclonable identity. This platform enforces a secure boot process for DEVICE firmware and provides a secure mechanism to attest to the state of the DEVICE firmware.

PIT-Cerberus is distributed as set of APIs. These APIs encapsulate the various functionalities extended by the HRoT processor to lock and unlock the BIOS/BMC of the DEVICE. The design of these APIs is guided by the principle of enabling Cerberus to lock and unlock the BIOS in a secure and efficient manner. The API consists of the following functions:

1. `lock()`: This function uses the internal `pit_keygenstate()`, `keyexchangestate()`, `pit_secretkey()` to perform all the operations needed to lock the BIOS and loads the secret key in a 32-byte empty array (*secret*).

2. `pit_keygenstate()`: This function is responsible for generating the ECC key pair for HRoT. It loads the length of the key in *key_length*, initializes private key in *privkey*, public key in *pubkey*, and the state of the HRoT in *state* parameter.

3. `keyexchangestate()`: Exchange the public key of HRoT and SERVER. On success, `keyexchangestate` will initialize *pubkey_cli* with the HRoT's public key and load the *pubkey_serv* variable with a public key received from the SERVER.

4. `pit_secretkey()`: It takes ECC private key from HRoT and SERVER, computes the secret key and loads in *secret* parameter.

5. `unlock()`: Do all the unlocking operations. These operations generate OTP, encrypt it, send to SERVER and validate the OTP.

6. `receive_product_info()`: This function receives the product information from the SERVER and assigns it to some parameters, such as encrypted registration ID (*EncryptedProductID*), tag (*EncryptedProductIDTag*), registration ID size (*ProductIDSize*), an initialization vector (IV) used for encryption (*aes_iv*), and size (in bytes) of the vector in (*aes_iv_size*).

7. `pit_connect()`: It initiates a connection to designated SERVER. It takes the port address of the SERVER as a input and returns an integer pointing to the file descriptor (socket) which can be used to send/receive from the SERVER.

8. `pit_OTPgen()`: This function generates a random string representing *OTP*. Additionally, encrypts that *OTP* using the secret key (*secret*) as key for the AES-GCM and loads in *encOTP* parameter.

9. `pit_encryption()`: It encrypts a message using a secret key. This function takes secret key, message and use AES-GCM method to encrypt the message and loads into *ciphertext* parameter.

10. `send_unlock_info()`: This function sends the encrypted OTP (*encOTP*) to the SERVER which will be later sent to USER.

11. `pit_OTPValidation()`: This function validates the encrypted OTP (*encOTP*). It does this by taking encrypted *OTP* (OTP_S) as input, decrypting it, and comparing it against the original OTP (*OTP*). If valid, the function returns 1 and the parameter *result* will hold *true* otherwise *false*.

12. `pit_decryption()`: This function takes encrypted message (*ciphertext*), secret key (*secret*) as input, decrypts it and loads the message to the provided *plaintext* buffer.

Figures 4.3 and 4.4 describe the sequence of the function calls to implement the PIT-Cerberus protocol.

In the SERVER implementation, we leverage the cryptography library for Python, employing three key modules Elliptic Curve Cryptography (ECC), serialization, and Advanced Encryption Standard - Galois/Counter Mode (AES-GCM). The ECC module facilitates the generation of a public-private key pair for the SERVER and furnishes a method to perform the Elliptic-curve DiffieHellman (ECDH) protocol, enabling the generation of a shared secret using the public key of another party. This module is indispensable to our implementation, although any library that

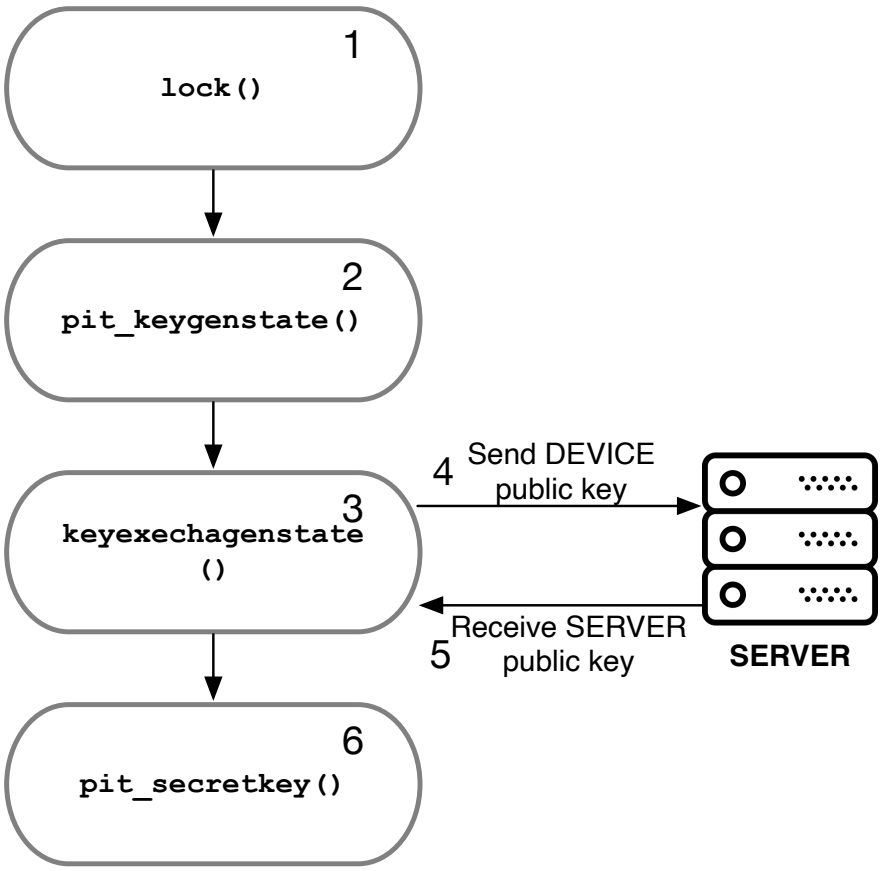

Figure 4.3: Function call sequence in Locked State of "*pit*".

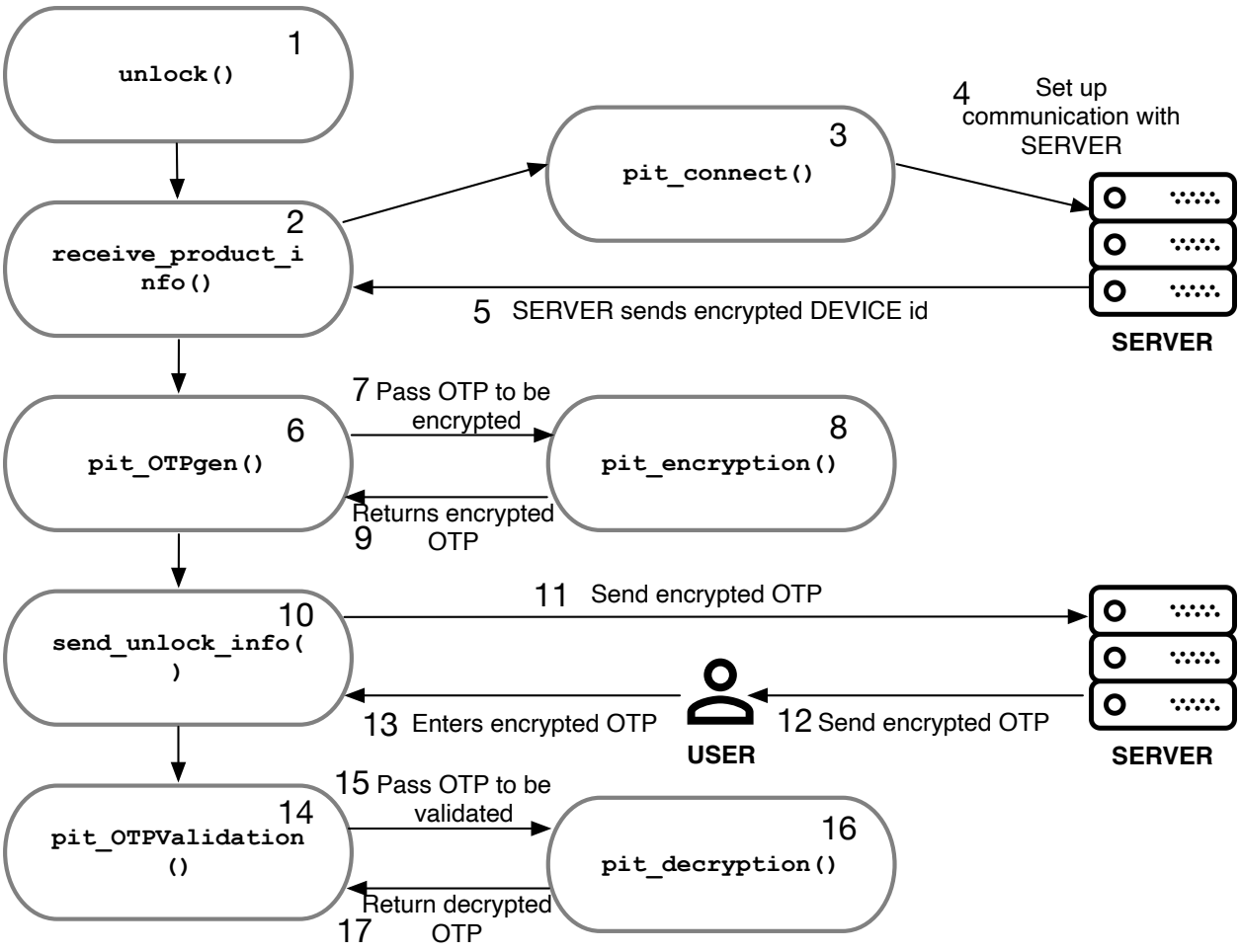

Figure 4.4: Function call sequence in Unlocked State "*pit*".

supports the generation of an elliptic curve key pair and the ECDH protocol compliant with NIST standards [68] would suffice.

The serialization module is tasked with key distribution and reception. Prior to the SERVER's public key transmission to HRoT, it is encoded into DER format via the serialization module. Similarly, HRoT's public key undergoes DER serialization before being sent to the SERVER. Upon receipt, the SERVER employs the serialization module to convert the DER-encoded public key from HRoT into a format compatible with the cryptography library, specifically an EC key. Libraries providing key serialization that adhere to NIST standards are suitable for this process.

The AES-GCM module is responsible for data encryption and decryption. AES-GCM is the chosen form of AES owing to its native support in Cerberus. Coupled with the shared secret derived from ECDH, the AES-GCM module serves as the key for encryption and decryption, ensuring secure and coherent communication between the SERVER and HRoT. For DEVICES requiring Cerberus functionality, the core set of source code delivers a suite of foundational features that can be integrated and ported. The provided code, largely device-agnostic, defines the required abstraction layers. Therefore, we tailored Cerberus by introducing a new library and APIs encompassing the lock and unlock mechanisms for BIOS/BMC. These modifications were instituted within the core module, designated as *PIT-Cerberus*.

A detailed documentation about the APIs and "*pit*" library is already publicly available in GitHub [69].

4.4 Evaluation & Discussion

Our primary focus with this evaluation was to evaluate the PIT-Cerberus framework's performance with the library. We developed various experimental test scenarios to evaluate our framework with the library's performance. We ran our experiments on 2 virtual Linux servers. The client side (assumed as HRoT) has a 5000 MHz 12th Gen Intel(R) Core(TM) i7-12700K processor, x86_64 architecture, 20 cpus and Server in Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz, x86_64 architecture, 12 CPU(s).

In order to evaluate that the PIT-Cerberus framework is working as it is supposed to, we used server level Project Cerberus with "*pit*" library with a C socket as a client and a SERVER implemented in Python. As designed and expected, our protocol delivered the anticipated results, with successful operations observed across both HRoT that uses PIT-Cerberus framework, and the SERVER. Figures 4.5 and 4.6 provide visual confirmation of these outputs.

As depicted in Figure 4.5, the DEVICE (BIOS) initiates in a locked state as the SERVER, equipped with its own distinct set of ECC key pair as shown in Figure 4.6, establishes a connection with HRoT. Following the successful exchange of ECC key pairs between HRoT and the SERVER, computation of the secret key is initiated.

We designed a simple two-step verification protocol for login. Once the USER logs in with their credentials, they can raise an unlock request on the SERVER. Upon receiving an unlock request from the USER on SERVER, SERVER commences communication with the HRoT. The unlocking process initiates when the SERVER transmits the encrypted PRODUCT registration ID (REG.ID_S) to HRoT. Subsequently, HRoT decrypts and validates this PRODUCT registration ID, the output of which is displayed in the HRoT terminal. Upon successful Product ID validation, HRoT generates an one-time password OTP. This OTP is encrypted using AES-GCM with the secret key (S), and dispatched to the SERVER, which then forwards this encrypted OTP (OTP_S) to the USER's email address (as illustrated in Figure 4.7). The USER, upon receiving the OTP_S, inputs it into the HRoT terminal. HRoT then decrypts this OTP_S utilizing the secret key (S), and performs a validation check. If the validation is successful, HRoT proceeds to unlock the DEVICE, as demonstrated in Figure 4.5.

```
pit_crypto: test_OTPgen
pit_crypto: test_OTPvalidation
pit_crypto: test_decryption
pit: test_pit_lock
Device is Locked.
pit: test_pit_unlock
User initiated Unlock Request......
PRODUCT ID Validation Successful. pid_status is: 1
OTP Generation and Encryption Successful.
Please Enter your OTP:
b'\x03hs\x7f ?\x86\xfc\xa9\xfa\xb3\x12\xec\x1cGa1\x07^\xf3\x04\xca!\xa9]\x11\xea0wpb?1\x13\x16?\xf8\xc1\xac\x11\xe6\xda\x8c7\x07\xce\xedU
\x1d\xc9\tOaH\xa3\xdf\x98i\xb5AD\x1c?=V{\xef_\x0bx\x80\xcex\xb3\xed\x0e]9\x9f\xda\xe1W\x8e\xa2\r\xdb\xd54\xce\xdd\xe3\xe8q\x15\x00/b[0\xf
0\xc8\xc7\xee,\xce\x9f\x1b\xe6\xbd,\xcdY&\x1aj^c3\xbd\xef\x80\xc1\xd7~g\xc0\xcf\x1e'
Encrypted OTP sent to Server.
OTP Validation Successful. pid_status is: 1
```

Figure 4.5: Output from HRoT Terminal.

In addition to the evaluation of the PIT-Cerberus library with the core of Project Cerberus, we also carry out extensive testing ("Compatibility Testing") to validate their compatibility with different SERVER libraries that use for the generation of ECC key pairs and shared secret key. Different SERVER environments employ varied libraries for the ECC key pair generation and shared secret key computation, each having its own intricacies and idiosyncrasies. It was crucial to confirm that the PIT-Cerberus library interacts seamlessly with these different SERVER libraries without any interoperability issues. To achieve this, we conduct a series of interoperability tests in multiple SERVER environments, each employing different libraries for key pair generation and shared secret key computation. We have tested with commonly used libraries, such as OpenSSL [70], libsodium [71], and Cryptlib [72], which are widely accepted for their robustness and compliance with industry standards. In each case, we confirm that the PIT-Cerberus library that has been used in HRoT, is able to correctly and efficiently compute and exchange ECC key pairs and shared secret keys without any compatibility issues. We also ensure that the computed keys adhered to the relevant NIST standards, and the elliptic curve cryptography functionality is up to the mark.

Furthermore, we have validated that the PIT library could correctly interpret the serialized keys received from the SERVER and could successfully execute the Elliptic-curve DiffieHellman (ECDH) operation to generate a shared secret key. We have also checked the successful encryption and decryption of data using the derived shared secret key. Through this extensive testing, we have validated that the PIT-Cerberus library is compatible with a wide range of libraries for generating ECC key pairs and shared secret key, thus enhancing the universal applicability and adaptability of our protocol.

We show a few results of our validation process in a proof of concept incorporating an OpenSSL-based SERVER. The objective is to demonstrate the universal capabilities of our designed library in handling operations regardless of the library types. In the execution of our model, HRoT transmits the public key to the SERVER, encoded in Distinguished Encoding Rules (DER) format. As is depicted in Figure 4.8 (top-right), the OpenSSL SERVER effectively interprets the public key delivered by HRoT. The DER format public key is subsequently outputted in the SERVER ter-

```
tallahassee:~/Rakesh$ python3 pid-server.py
Generated: Server ECC Key Pair.

Server X value : 768465705340819100253629315231074600515073783807449255146945688342708940824
Server y value : 109252678345389310469317484089853095411478063296788366629630286751158294466709
listening
accepted
Connected by ('127.0.0.1', 40564)
Recived: Cerberus ECC Key Pair.

Client X value : 864774840645440664077226990236261786998423991788212449554964793995626422512 39
Client y value : 457020251591109096967518400286968162193238137950203812681101274813747646578 07

Generating shared secret..... Successful.
listening
listening
accepted
Connected by ('127.0.0.1', 34996)
[DEMO]: Encrypted OTP : b'\x03hs\x7f ?\x86\xfc\xa9\xfa\xb3\x12\xec\x1cGa1\x07^\xf3\x04\xca!\xa9]\x11\xea0wpb?1\x13\x1
6?\xf8\xc1\xac\x11\xe6\xda\x8c7\x07\xce\xedU\x1d\xc9\tOaH\xa3\xdf\x98i\xb5AD\x1c?=V{\xef_\x0bx\x80\xcex\xb3\xed\x0e]9
\x9f\xda\xe1W\x8e\xa2\r\xdb\xd54\xce\xdd\xe3\xe8q\x15\x00/b[0\xf0\xc8\xc7\xee,\xce\x9f\x1b\xe6\xbd,\xcdY&\x1aj^c3\xbd
\xef\x80\xc1\xd7~g\xc0\xcf\x1e'

Enter your email: rakeshpodder3@gmail.com
```

Figure 4.6: Output from SERVER Terminal.

rakrock121212@gmail.com 6:43 PM (3 minutes ago)

to

b'\x03hs\x7f ?\x86\xfc\xa9\xfa\xb3\x12\xec\x1cGa1\x07^\xf3\x04\xca!\xa9]\x11\xea0wpb?1\x13\x16?\xf8\xc1\xac\x11\xe6\xda\x8c7\x07\xce\xedU\x1d\xc9\tOaH\xa3\xdf\x98i\xb5AD\x1c?=V{\xef_\x0bx\x80\xcex\xb3\xed\x0e]9\x9f\xda\xe1W\x8e\xa2\r\xdb\xd54\xce\xdd\xe3\xe8q\x15\x00/b[0\xf0\xc8\xc7\xee,\xce\x9f\x1b\xe6\xbd,\xcdY&\x1aj^c3\xbd\xef\x80\xc1\xd7~g\xc0\xcf\x1e'

Figure 4.7: Encrypted OTP Sent to Email.

```
Openssl attempting to read the server's public key

Public-Key: (256 bit)
pub:
    04:77:46:1f:52:9c:73:ca:9e:f1:2f:c7:cd:55:32:
    fd:94:76:ff:f1:18:ab:18:57:78:df:e3:d2:57:a4:
    9e:40:b0:34:95:ba:ec:ef:43:5c:fe:5c:4c:2b:6c:
    ee:df:cd:34:8f:da:c8:68:ff:41:c4:ed:10:71:a5:
    24:ec:26:97:d5
ASN1 OID: prime256v1
NIST CURVE: P-256

Using openssl to derive shared secret :
zi.RSIb    {n

Using the server to derive shared secret :
zi.RSIb    {n
```

```
Openssl attempting to read the client's public key :

Public-Key: (256 bit)
pub:
    04:19:4e:f7:a9:c5:92:d5:05:b9:68:a9:9c:7c:09:
    f8:c6:cc:50:88:5c:23:f4:e1:a0:7f:8d:28:46:d2:
    5b:c1:34:bd:08:e1:92:e6:34:7f:4f:d7:24:c8:55:
    46:4b:b0:8e:a2:52:e9:39:80:83:e0:4d:f5:50:24:
    68:cb:2c:be:a1
ASN1 OID: prime256v1
NIST CURVE: P-256
```

```
Is this shared secret the same as the one generated by client AND the same as the one generated by server? True
```

Figure 4.8: SERVER (top-left) & HRoT's (top-right) public key in DER format, (bottom) secret key of OpenSSL and Python SERVER.

minal, further validating the successful exchange of cryptographic keys between HRoT and the SERVER.

As further demonstrated in Figure 4.8 (top-left), it can be observed that OpenSSL is proficient in interpreting both public and private keys generated by the SERVER. This empirical evidence substantiates that cryptographic library, as implemented in our framework, is compatible with OpenSSL and by extension, compliant with the National Institute of Standards and Technology (NIST) standards for cryptographic algorithms. Therefore, our methodology exhibits a broad spectrum of interoperability, ensuring compatibility with any cryptographic systems that adhere to NIST standards. The evidence provided in Figure 4.8 (bottom) offers definitive proof that, by employing OpenSSL in conjunction with our Python SERVER, identical secret keys can be produced. This not only demonstrates the robustness and reliability of our key generation process, but also unequivocally confirms the compatibility of our implementation with systems conforming to NIST standards. As such, we can assure the broader applicability of our system, facilitating secure and effective communication between diverse cryptographic frameworks that adhere to these industry-accepted norms.

Apart from validations and compatibility tests, we tested the protocol's exceptions and failures handling capability thoroughly. For this we designed various test scenarios. Table 1 shows some scenarios of our testing. Its all during the unlocking phases. We also tested out when the DEVICE is locked by USER and shipped to another USER, the process will be started from starting phase of key_gen_state to the unlocking_state.

We have successfully modified the Project Cerberus embedded framework, giving birth to PIT-Cerberus for HRoT, a system equipped to secure PRODUCT during transit by locking the DEVICE (with BIOS/BMC) to address the research question (RQ2) mentioned in Section 1.4. PIT-Cerberus boasts the necessary cryptographic capabilities that a microcontroller or microprocessor requires for BIOS locking. Our work, including the "*pit*" library, is made publicly available in our GitHub repository [69]. Additionally, a comprehensive set of instructions for operating PIT-Cerberus has been provided, further promoting the accessibility and usability of our solution.

Test Scenarios	Description	Handling	State
Server unavailability	The Server loses connections during locking or unlocking	The program will wait until the timer specified time. If the connection is not back, it will revert back to the previous state and throws a timeout exception.	Lock state
REG.ID validation fails	The encrypted REG.ID_S fails to validate due to wrong REG.ID or empty.	If the encrypted REG.ID_S cannot be validated, the program throws a REG.ID validation failure error and reverts back to the lock state	Lock state
OTP validation fails	The encrypted OTP_S fails to validate due to wrong OTP or empty generated by HRoT.	If the encrypted OTP_S cannot be validated, the program throws a OTP validation failure error and reverts back to OTP_gen state.	OTP_gen state
Unable to send OTP_S to USER email	The encrypted OTP_S fails to deliver to the USER due to wrong connectivity issues.	If the encrypted OTP_S is not available to the USER, the HRoT will wait until the specified timer expires, then throw a timeout exception and revert back to the OTP_gen state.	OTP_gen state

Table 4.1: Protocols exceptions and failures handling capability in various test scenarios.

4.5 Security Analysis

In the event of attacks that could lead to SERVER or service unavailability, we have implemented several exceptions and error handling mechanisms within the program to prevent unwanted interruptions or crashes during the unlocking procedure. We will not consider such attacks in the security analysis.

During the unlocking process, attackers can monitor and sniff on the communication channel established between the HRoT–SERVER and SERVER–USER. By doing so, they can capture encrypted PRODUCT registration ID (REG.ID_S) sent from the SERVER to HRoT upon an unlock request initiated by the USER. Additionally, attackers can intercept and steal login credentials when the USER attempts to log into the SERVER, potentially gaining unauthorized privileges. Furthermore, when HRoT sends an encrypted OTP to the SERVER, and the SERVER forwards this encrypted OTP (OTP_S) to the USER, attackers monitoring the communication channel could obtain the encrypted OTP (OTP_S) and misuse it for improper authorization access to the PRODUCT.

Another potential threat is replay attacks, where the attacker does not need to decrypt the message but merely reuses it. For instance, an attacker could capture a packet during USER logging into a system and replay it to gain unauthorized access without needing the USER's password. This allows the attacker to impersonate a valid USER, tricking the SERVER into sending the OTP_S to a malicious USER. This constitutes a man-in-the-middle (MitM) attack, where the attacker impersonates USER to perform privilege escalation through sniffing and replaying data. An attacker could also capture the REG.ID_S and replay them back to the HRoT to impersonate the SERVER and launch an MitM attack.

The protocol we've designed and implemented is robust against such attacks. For instance, when the SERVER transmits the REG.ID_S, they are encrypted using the Advanced Encryption Standard - Galois/Counter Mode with a 256-bit key (AES-GCM 256 Key) encryption scheme. AES-GCM is resistant to several types of attacks including *Known Plaintext Attack (KPA)*, *Chosen Plaintext Attack (CPA)*, *Chosen Ciphertext Attack (CCA)*, and *Ciphertext-Only Attack (COA)*. The

design of this algorithm ensures that knowing a *plaintext-ciphertext* pair does not significantly aid in deducing the encryption key, as the relationship between *plaintext* and *ciphertext* yields minimal useful information. Additionally, with a 256-bit key size, AES-GCM-256 is currently infeasible to break through brute-force attacks, considering the 2^{250} possible key combinations.

However, to address advanced threats like the *Adaptive Chosen-Plaintext Attack (ACPA)* and *Adaptive Chosen-Ciphertext Attack (ACCA)*, where the attacker adapts their choices based on feedback from previous encryptions or decryptions, we employ Elliptic Curve Diffie-Hellman (ECDH) with a 256-bit or higher prime modulus for the elliptic curve. This is used to generate a secret key for AES-GCM encryption. In every session or iteration, this secret key is randomly generated, ensuring a new secret key for subsequent encryptions. Utilizing ECDH-256 for key exchange allows for the generation of a new AES-GCM-256 key for each session or even each message, greatly bolstering security. This method prevents key reuse and significantly complicates any attempt by attackers to compromise the communication. The synergy of AES-GCM-256 and ECDH-256 offers extensive security measures, where AES-GCM-256 secures the confidentiality and integrity of communications, and ECDH-256 ensures secure key negotiation, safeguarding against eavesdropping and MitM attacks.

To mitigate replay attacks, where an attacker could reuse captured REG.ID_S to launch MitM attacks or disrupt workflow, we've incorporated Lamport timestamps to ensure the freshness and integrity of messages transmitted over the communication channel. Thus, when the SERVER sends the REG.ID to the HRoT, it's encrypted using our protocol's encryption scheme, which maintains encryption randomness and is coupled with a Lamport timestamp to preserve message freshness. Furthermore, as the HRoT establishes a connection with the SERVER, each connection utilizes a unique session key. Every message sent to or from the SERVER or HRoT undergoes validation checks for session integrity, timestamps, and encryption, effectively preventing replay attacks on the HRoT or SERVER.

Additionally, to thwart adversaries attempting to impersonate the SERVER and launch a MitM attack, we employ the Digital Signature Algorithm (DSA), which leverages keys derived from El-

liptic Curve Cryptography (ECC). This approach effectively safeguards against MitM attacks, ensuring the authenticity of communications and protecting against impersonation and data integrity threats.

We implement multi-factor authentication (MFA) when the USER tries to log into the SERVER using an otp (generated and sent by the SERVER). This means that even if an attacker manages to sniff the login credentials, they would not be able to log into the SERVER, as the login otp is sent to the USER's registered email ID. Additionally, a session is established between the USER and the SERVER at the start of communications to ensure the freshness of messages and prevent replay attacks, significantly reducing the possibility of successfully impersonating a valid USER in a MitM attack.

While there is a possibility that an attacker might sniff the OTP_S as the SERVER sends them to the USER, obtaining these OTP_S would first require the attacker to log into the SERVER and initiate the unlock request. This process validates the legitimacy of the USER by checking the REG.ID with the HRoT, after which the HRoT generates the OTP and sends it to the SERVER. However, as previously mentioned, attacks on this process are considered infeasible. This indicates that the PRODUCT (via HRoT) is connected and in possession of a legitimate user. Therefore, intercepting the OTP_S would not be beneficial for the attacker, as the OTP needs to be manually inputted into the PRODUCT to initiate DEVICE booting process. This layered security approach ensures that the authentication mechanism is robust against unauthorized access attempts.

Chapter 5

Secure Remote Firmware Update Protocol (S-RFUP)

5.1 Threat Model

In this section, we delineate the threat model pertinent to the secure remote firmware update framework. The key entities within this framework include the Hardware Root of Trust (HRoT), firmware devices (FD), the Update Agent (UA), the sender, and the recipient. The HRoT is a critical component leveraging the Project Cerberus embedded framework with PIT (Protection in Transit) capabilities. Firmware devices include laptops, workstations, and commercial servers integrating BIOS/BMC functionalities. The UA is a function within a firmware update subsystem designed to identify firmware devices capable of executing a PLDM firmware update and to facilitate the transfer of component images to these devices. Notably, each firmware update subsystem supports only a single UA function. The sender is the manufacturer or company responsible for maintaining the firmware and initiating firmware updates, while the recipient is the end user utilizing the firmware device with HRoT capabilities.

5.1.1 Assumptions

For this work, we made a number of assumptions. First, we assume the HRoT processor is considered tamper-proof and trusted. The company is deemed trustworthy, with no insider threats, and securely programs the HRoT processor with the PIT-Cerberus and related libraries and data. The Update Agent (UA) is honest and not curious, ensuring the protection of any stored confidential information against breaches of confidentiality and integrity. The company server used by UA is considered a trusted zone, protected by Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), alongside strict security policies and personnel and not a target for intrusion and denial-of-service attacks. Protocols and cryptographic methods, the key size of Advanced Encryption Standard (AES), prime modulus of Elliptic-curve cryptography (ECC), curve selection

for ECC, and the Digital Signature Algorithm (DSA), are carefully selected and implemented in a secure environment (SCIF[4]) to resist physical tampering and side-channel attacks. The initial key establishment is completed before deployment and is not regarded as an issue. Encryption keys are generated with strong randomness and installed in the firmware devices during the manufacturing process.

5.1.2 Attacker Model

The primary target of the attacker is the communication channel between UA, and FD during firmware updates. We adopt the Dolev-Yao attacker model [73], where the attacker can eavesdrop, intercept, modify, or inject messages into the communication channel. Replay attacks involve attackers delaying or re-sending packets to mislead the FD or UA. Attackers might also inject false information to disrupt ongoing services. An attacker impersonating (MITM attacks) the UA or a legitimate FD could compromise firmware integrity or gain unauthorized access. The FD is vulnerable to physical tampering during transit, such as replacing the HRoT with a malicious microcontroller or embedding a hardware Trojan. To mitigate this, tamper-proof seals are assumed to protect the device during shipment, alerting the Recipient if breached. This paper does not consider scenarios involving physical tampering with the Firmware Device.

5.1.3 Desired Security Properties

The essential security properties required for network traffic within a secure remote firmware update protocol include data integrity, data authentication, data confidentiality, and data freshness.

- **Data Integrity**: Ensures that the firmware updates received by the Firmware Device (FD) have not been tampered with.
- **Data Authentication**: Verifies the source and integrity of the firmware update to ensure that the updates come from legitimate sources and prevent unauthorized modifications.

[4]Sensitive Compartmented Information Facility.

- **Data Confidentiality**: Maintains the secrecy of firmware data by protecting sensitive proprietary information from unauthorized access.
- **Data Freshness**: Guarantees that the firmware updates are recent.

By addressing these security properties, the protocol (S-RFUP) aims to provide a robust, secure, and reliable process for remote firmware updates.

5.2 Description of S-RFUP

The S-RFUP operates as a client-server model, as depicted in Fig. 5.1. The key entities within this protocol include the Hardware Root of Trust (HRoT), Firmware Devices (FD), Update Agent (UA), Sender, and Recipient. UA is a function within S-RFUP framework, designed to identify firmware devices, capable of executing a PLDM firmware update and to facilitate the transfer of component images to these devices. HRoT is the tamper-proof micro-controller that acts as the hardware root of trust. It is a critical component leveraging the Project Cerberus embedded framework for Firmware Device. FD is a PLDM endpoint (terminus) that comprises one or more processor elements that execute firmware. The Sender is the manufacturer or company responsible for maintaining the firmware and initiating firmware updates, while the Recipient is the end user utilizing the firmware device with HRoT capabilities.

5.2.1 Proposed Approach

The proposed S-RFUP framework is divided into two main segments; 1) the establishment of a secure channel, 2) the initiation of remote firmware update process. Initially, UA and FD establish a connection and compute public and private key pairs using Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol [20, 21]. UA generates an ECC (Elliptic Curve Cryptography) private key (d_U) - public key (q_U) pair, where, $q_U = d_U \times G$. G is the base point of the chosen elliptic curve. Then UA sends the public key (q_U) to the FD. FD generates a private key d_F, computes a public key $q_F = d_F \times G$ and an AES secret key $S = d_F \times q_U$. FD transmits its public key to the UA, which then computes the same AES secret key S as $S = d_U \times q_F = d_U \times \{d_F \times G\} = d_F \times \{d_U \times G\} = d_F \times q_U$.

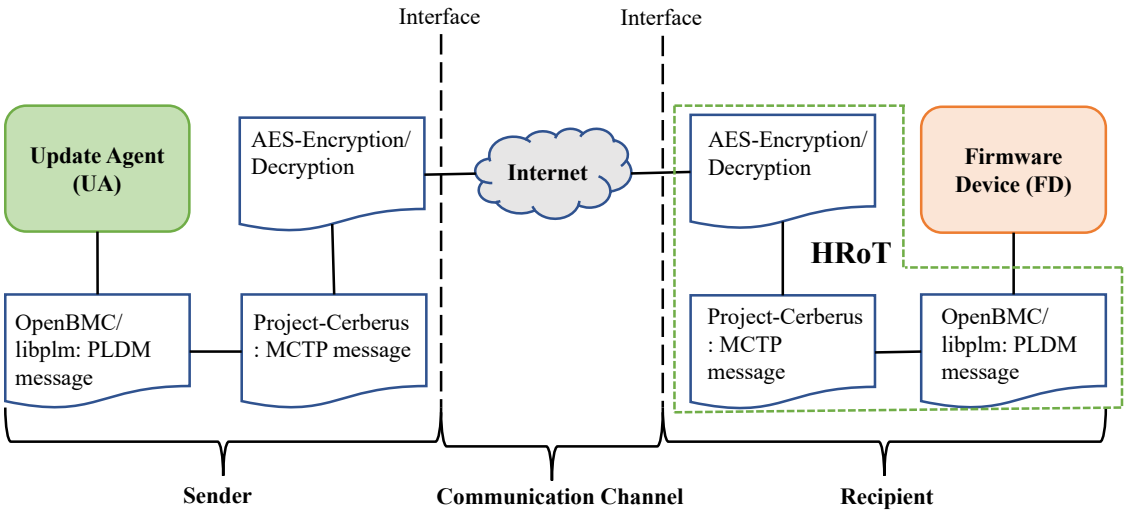

Figure 5.1: High-level representation of S-RFUP framework.

In the second segment, as illustrated in Fig. 5.2, UA initiates the firmware updates, converting the firmware image into a Platform Level Data Model payload ($pldm$ message) using '*OpenBMC/libplm*' library. The $pldm$ message is then transformed to a $mctp$ message by the Project-Cerberus using Management Component Transport Protocol (MCTP) & S-RFUP. After that, the $mctp$ message is encrypted with AES encryption [67] schema ($encData = AESEncryption(mctp, S)$) using a shared key ($S$) generated previously by UA, before sending it to HRoT. HRoT, containing the Project Cerberus framework (MCTP Protocol) with S-RFUP functionalities, decrypts it using AES ($mctp = AESEncryption(encData, S)$) and converts the $mctp$ to $pldm$ message, before sending it to FD. Based on the request data ($pldm$ message) FD generates an response data using '*libpldm*' and sends it back to UA.

Similarly, if FD has request data, it follows the same encoding, encrypting, transferring, decrypting, and decoding steps, ultimately returning the response data to UA. This process will continue till all the firmware components (including firmware package header and payload) transfer to FD and FD activates the firmware updates.

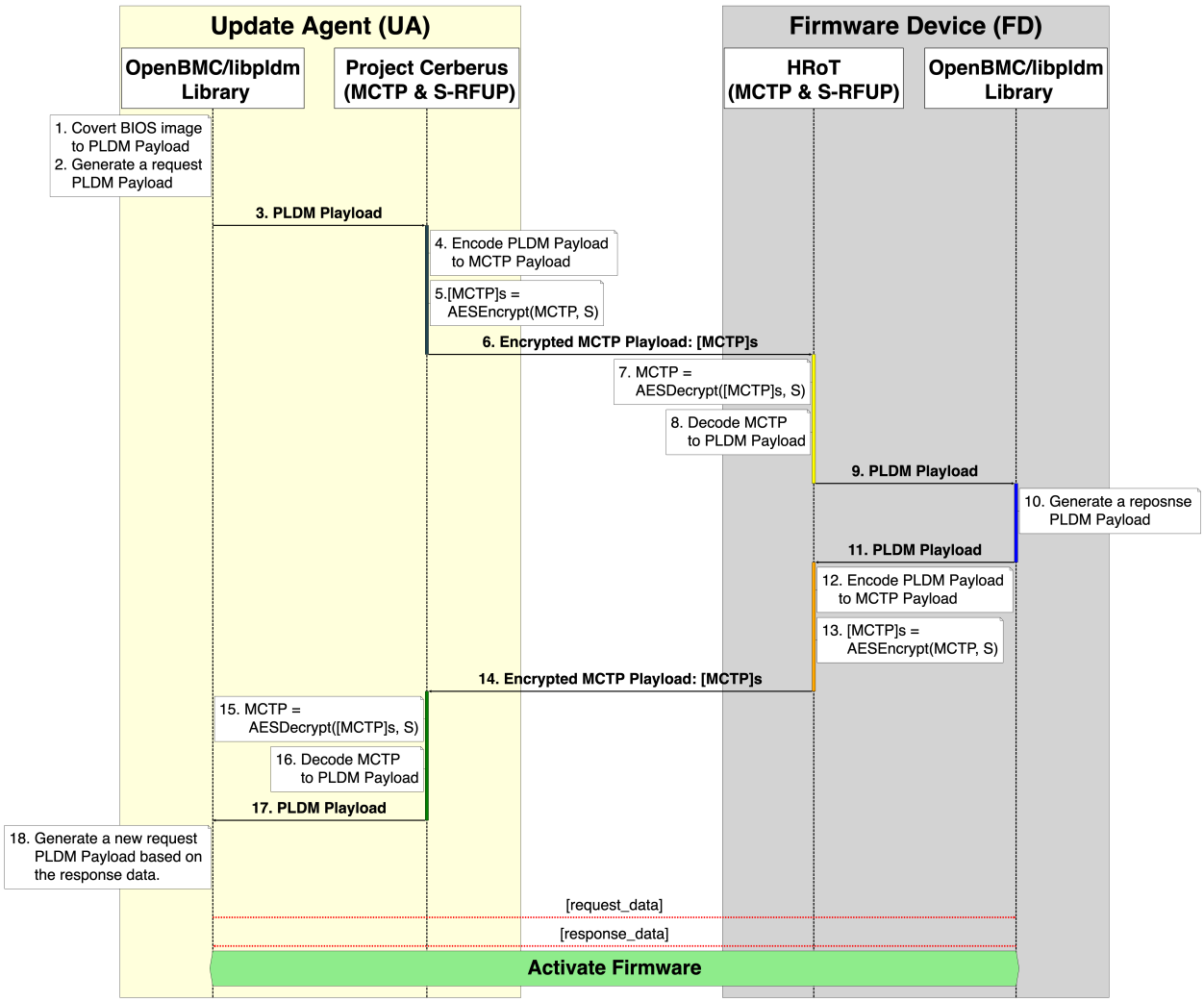

Figure 5.2: Sequence diagram of S-RFUP firmware update process.

5.2.2 PLDM firmware update package

The firmware update package is designed to work in conjunction with PLDM Firmware Update commands and contains several essential elements. These elements include a firmware package header that outlines the update package's contents as illustrated in Fig. 5.3. Specifically, the header provides a description of the overall packaging version and the date it was created. It also includes device identifier records, which specify the firmware devices (FDs) targeted for the update. Further, the header details the package contents, listing each component image's classification, offset, size, and version. Additionally, the package incorporates a checksum to ensure the integrity of the data.

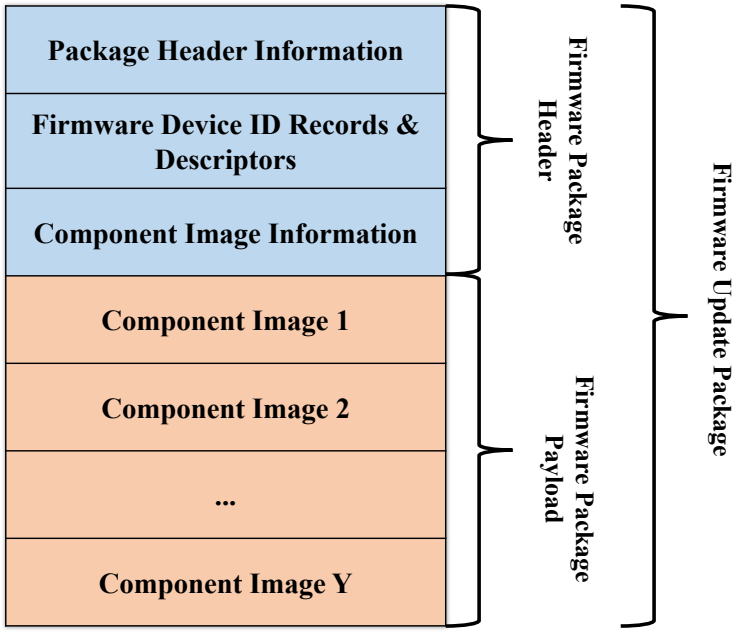

Figure 5.3: PLDM firmware update package.

Within the S-RFUP framework, three primary types of PLDM commands facilitate the transfer of firmware package headers and component images during updates. These include: {Inventory: QueryDeviceIdentifiers, GetFirmwareParameters},{Update: RequestUpdate, PassComponentTable, UpdateComponent, TransferComplete, VerifyComplete, ApplyComplete, ActivateFirmware, GetStatus, CancelUpdateComponent, CancelUpdate}, and {Transfer: RequestFirmwareData, GetPackageData, GetDeviceMetaData, GetMetaData}. These commands utilize the '*OpenBMC/libpldm*' library for efficient serialization and de-serialization of the PLDM messages.

5.3 Implementation of S-RFUP

In this section we discussed the implementation and execution flow of S-RFUP within the Project Cerberus framework. It outlines the libraries, procedural steps, and interactions between various components involved in the firmware update process. Fig. 5.4 illustrates a generic $mctp$ message that has encapsulated a $pldm$ message. This $pldm$ message is generated by UA using '*libpldm*' library. For each PLDM command described in section 5.2, the fields of $pldm$ message

will be populated with different values. Once the $pldm$ message is generated it will be encoded to $mctp$ message shown in Fig. 5.4 before encrypting or decrypting. For this purpose, we developed the following core libraries that could handle the firmware update process.

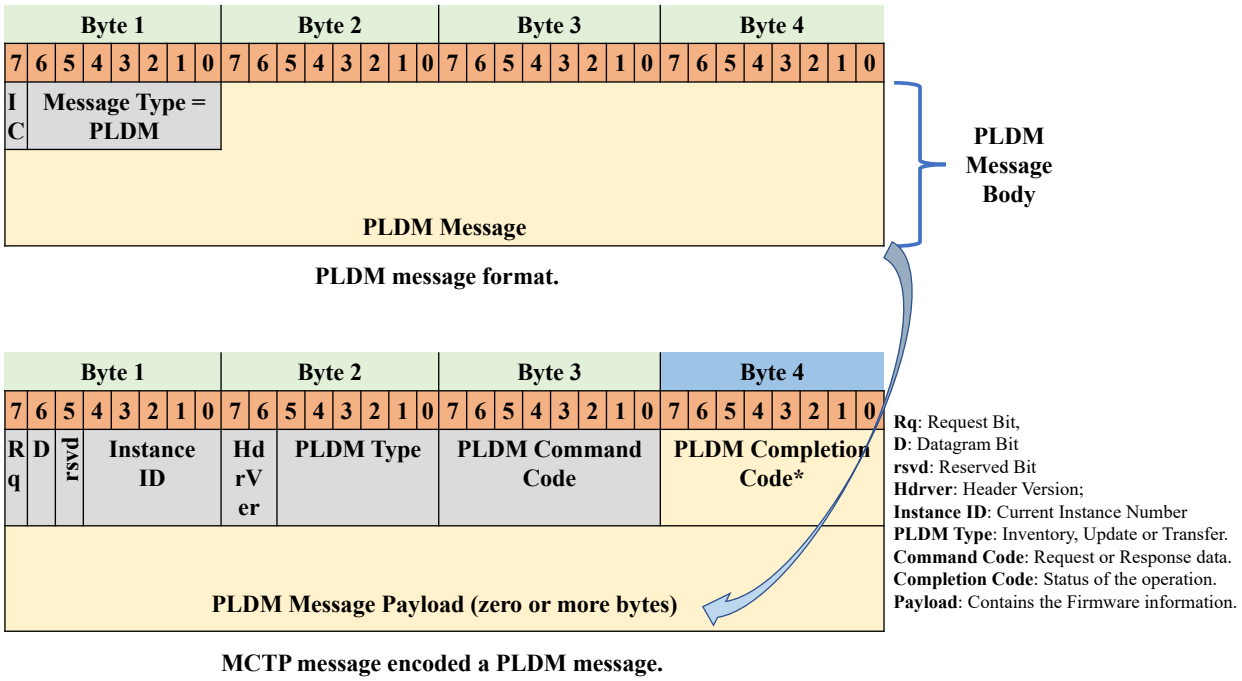

Figure 5.4: Generic MCTP message encoded a generic PLDM message.

5.3.1 S-RFUP Core Libraries

The S-RFUP architecture employs a modular approach to firmware updates. The core modules of S-RFUP and their interaction with various external framework such as Project Cerberus and '*OpenBMC/libplm*', are illustrated in Fig. 5.5.

The core functionality for performing a PLDM firmware update is contained within the '*core/-pldm*' folder in Cerberus as shown in figure 5.6. The source code layout is as follows:

PLDM FWUP Crypto

The '*pldm_fwup_crypto*' supports the encryption and decryption of messages, securing communications across the network. It utilised the `aes.h & ecc.h` to generate ECDH key pair

Source	Function/API
New S-RFUP Libraries	
pldm_fwup_crypto	`keyGeneration(), keyExachange(), secretKey()` `AESEncryption(), AESDecryption(), generateDSA()`
cmd_interface_pldm	`cmd_interface_pldm_process_request()` `,cmd_interface_pldm_process_response()`
pldm_fwup_handler	`pldm_fwup_handler_run_update_ua(),` `pldm_fwup_handler_start_update_fd()`
pldm_fwup_manager	`pldm_fwup_manager_init(), pldm_fwup_manager_deinit().`

pldm_fwup_protocol_commands	`pldm_fwup_process_query_device_identifiers_request(),`
	`pldm_fwup_prcocess_get_firmware_parameters_request(),`
	`pldm_fwup_process_request_update_request(),`
	`pldm_fwup_process_request_update_response(), ...,`
	`pldm_fwup_generate_activate_firmware_request(),`
	`pldm_fwup_generate_activate_firmware_response()`

pldm_fwup_protocol	`struct pldm_fwup_protocol_version_string, struct`
	`pldm_fwup_fup_component_image_entry, struct`
	`pldm_fwup_protocol_component_parameter_entry`

Modified Project Cerberus Libraries	
core/mctp	`mctp_interface_process_packet()`
core/projects/linux	`platform_config()`
core/tools/testing	`setup_fwup_flash_virtual_disk()`

Table 5.1: Description of S-RFUP core libraries.

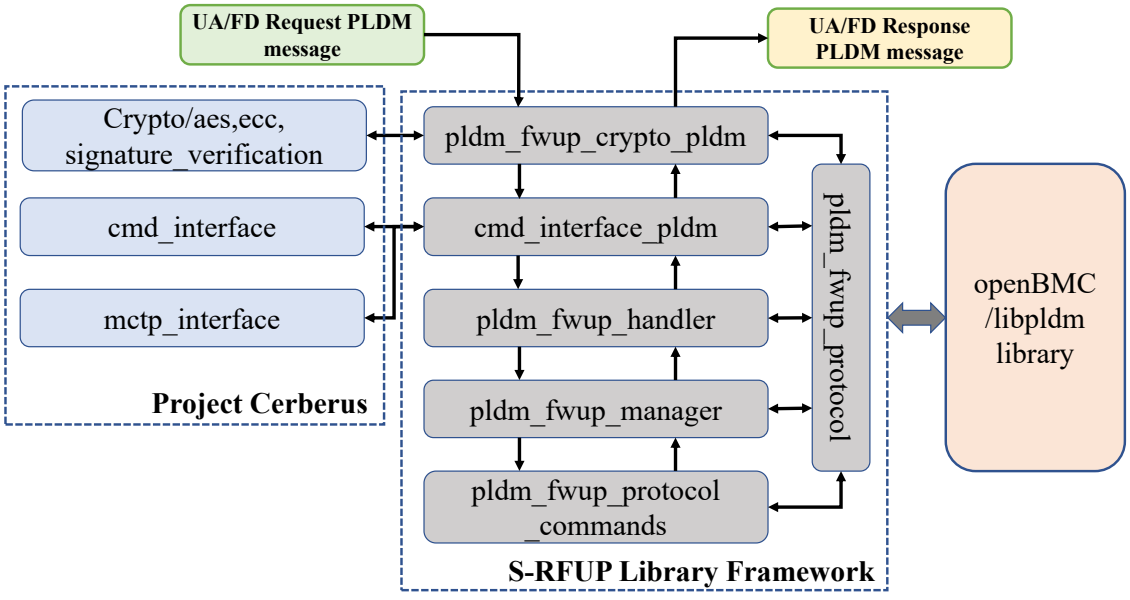

Figure 5.5: S-RFUP library framework and interaction with Project Cerberus & *libpldm*.

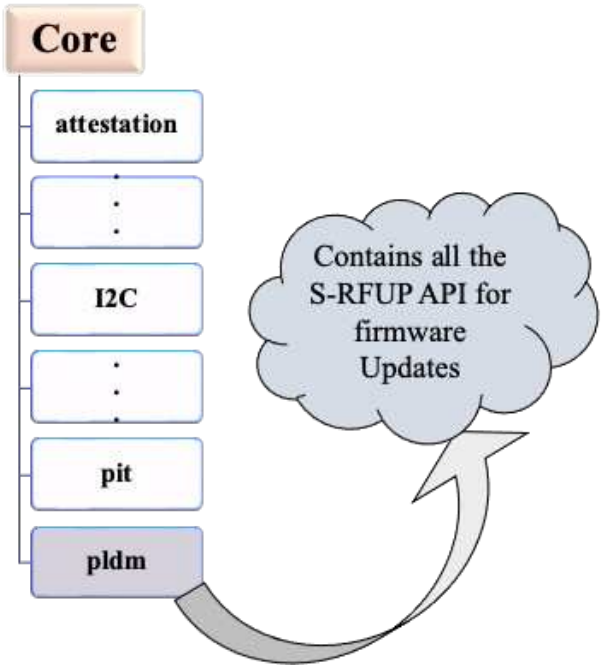

Figure 5.6: S-RFUP PLDM library.

and AES encryption from Project Cerberus. A *Lamport* timestamp within `AESEncryption()` `& AESDecryption()` is incorporated before encrypting/decrypting the $mctp$ message. We are also using ECC curve to generate a digital signature (DSA) to sign each encrypted $mctp$ message for UA and FD. The `signature_verification_ecc.h` file helps to verify the figital signatures of UA/FD during update process.

PLDM Command Interface

Project Cerberus (or Cerberus) defines a generic command interface called '*cmd_interface*' for processing requests and responses in a command protocol. The '*cmd_interface_pldm*' extends '*cmd_interface*' to handle PLDM specific commands as shown in Table 1. It inherits the properties and function pointers from '*cmd_interface*' which are then defined during its initialization. Currently '*cmd_interface_ pldm*' only processes PLDM firmware update command types, but can be further extended to process others such as PLDM for FRU commands.

Command Interfaces and MCTP: Cerberus uses MCTP as the protocol for which messages are exchanged throughout a Cerberus managed subsystem. MCTP is a flexible standard that can

encapsulate other protocols such as Cerberuss own command protocol, SPDM. We modified the `mctp_interface_process_packet()` function to handle PLDM command. During the processing of MCTP packets Cerberus will interpret the MCTP header and extract the message type field which descries the type of payload that packet is carrying. The payload is then passed along to its respective command interface for further processing.

PLDM FWUP Manager

The '*pldm_fwup_manager*' is a library for managing the state of a PLDM-based firmware update and allowing other parts of the S-RFUP framework to modify or view the information present in the firmware update commands. An instance of it is passed along to the '*cmd_interface_pldm*' so that during the processing of firmware update commands the information needed to populate or save the fields of the commands can be accomplished.

PLDM FWUP Protocol Commands

The '*pldm_fwup_protocol_commands*' contains functions that perform the actual decoding and encoding of PLDM commands, saving information to or populating message fields with information from the PLDM FWUP manager. For example, '`pldm_fwup_process_request_update_request()`' function is used to process incoming RequestUpdate PLDM commands saving information in the request data to the manager and extract the managers context to generate a RequestUpdate response.

PLDM FWUP Handler

The '*pldm_fwup_handler*' is the main driver code of S-RFUP. The handler mainly calls the API of the PLDM FWUP protocol commands and the MCTP interface API to generate, send, receive, process, and respond to PLDM firmware update commands. The most important fact is that at any time S-RFUP can be operating as either UA, performing firmware update on another device it manages or as the actual FD being updated. As such the '*pldm_fwup_handler*' interface contains two function pointers: `pldm_fwup_handler _run_update_ua()` for updating a

firmware device in the subsystem as an UA and `pldm_fwup_handler_run_update_fd()` for updating S-RFUPs own firmware (FD) as directed by another UA.

Apart form these core libraries, S-RFUP also has '*pldm_fwup_protocol*' library header containing various structures, macros, and enumerations used by the above-mentioned libraries. Table 1 shows the main libraries and some of API that we have designed, modified or used in the protocol. To convert the BIOS/BMC firmware images to a PLDM message we are mainly using `firmware_update.h` of '*libpldm*' library as external.

As explained in section 5.2, the firmware component images reside in the Firmware Update Package where they are retrieved as needed using Cerberus's flash module. Since we are compiling and evaluating on Linux, a virtual flash module was created to simulate that functionality using disk I/O. We have developed a python script `setup_fwup _flash_virtual_disk()`, that located in the '*core/tools*' directory, generates a 4GB binary file divided into sections to simulate different flash regions: one for package data, one for metadata, and one for each two firmware components. These regions are populated with random bytes.

5.3.2 State Transitions

Fig. 5.7 outlines the various PLDM command and states that the Firmware Device (FD) can be during update process. Each circle in the diagram represents a distinct state the FD might be in and, each rectangular boxes represents a PLDM command. Whenever the FD is initialized, or when it undergoes a system reboot or device reset, it starts in the IDLE state. The execution starts from QueryDeviceIdentifiers and ends at ActivateFirmware. Every PLDM command on a successful execution transits to next PLDM command and the associated states also change. For example, if S-RFUP executes RequestUpdate command, on success it will move to GetPackageData and state of FD will change from IDLE to LEARN COMPONENTS. Similarly, if the execution fails or FD throws a compilation code error, the state of FD remains on IDLE. This design helps us to understand the FD's state with each PLDM command, so that a prompt diagnosis can be launched if any error occurs and it also helps to standardise the update process.

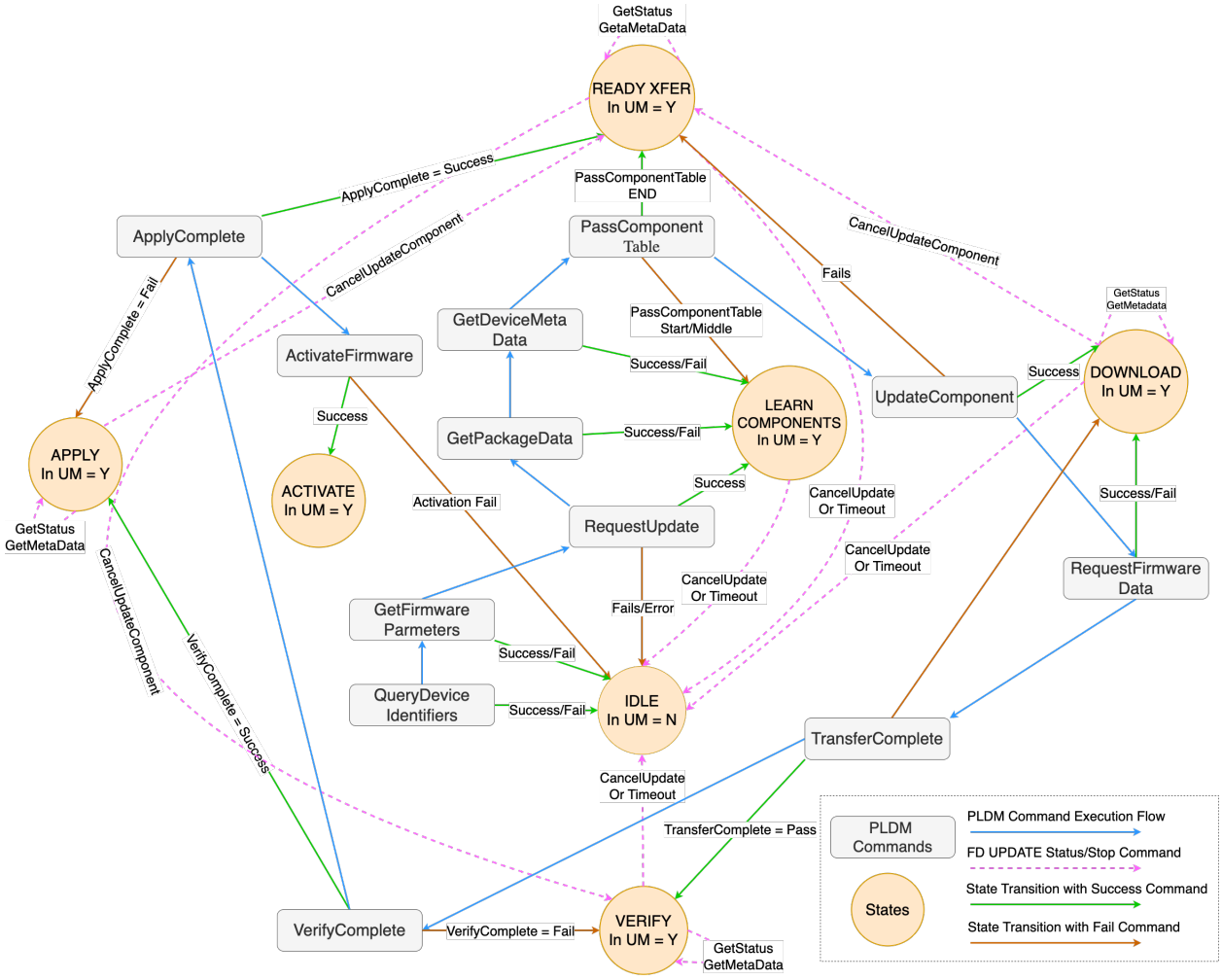

Figure 5.7: State Transition Diagram with PLDM Commands.

5.3.3 S-RFUP Update Flow in Project-Cerberus

The firmware update is performed in a sequential manner. Implementation for parallel operation and message exchange is not addressed in the study. Additionally, the '*pldm_fwup_manager*' must be initialized and the PLDM command interface must obtain a reference to the manager prior to the start of the firmware update process.

Before UA initiates the update process, `keyGeneration()` function of '*pldm_fwup _crypto*' library is responsible for generating the ECC key pair for UA and FD. It loads the length of the key in *key_length*, initializes private key in *privkey*, public key in *pubkey*. `keyExchange()`: function exchanges the public key of UA and FD. On success, `keyExchange()` will initialize *pubkey_cli*

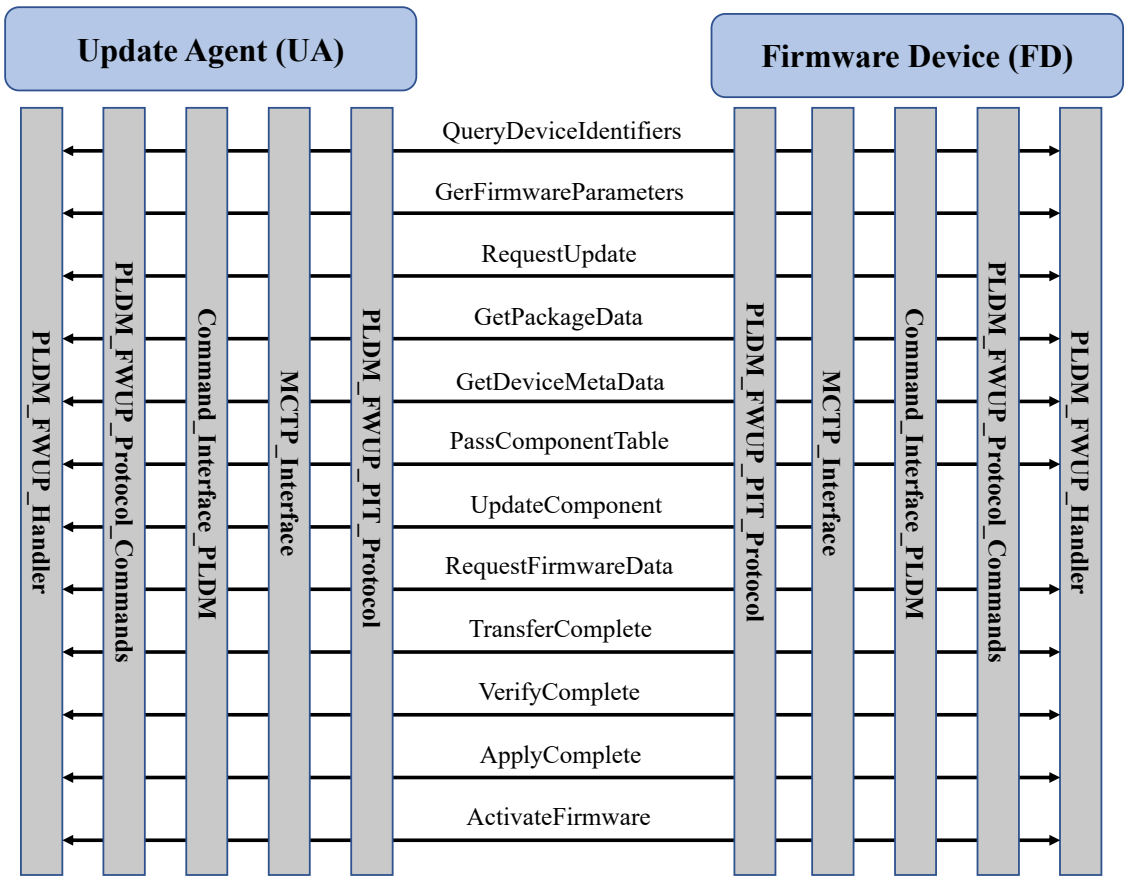

Figure 5.8: Cerberus PLDM Firmware Update Flow.

with the UA's public key and load the *pubkey_serv* variable with a public key received from the FD. The `secretKey()` takes ECC private key from UA and FD, computes the secret key and loads in *secret* parameter.

S-RFUP Operating as UA Flow

1. Initially, Cerberus determines the need to send the inventory commands QueryDeviceIdentifiers and GetFirmwareParameters based on a control boolean provided to `run_update_ua()`.

2. After sending the two inventory commands Cerberus will then issue the RequestUpdate command signaling to the device that a firmware update is eminent. The RequestUpdate command passes some necessary information such as the maximum transfer size and number of outstanding requests the device is allowed to issue to Cerberus. Both of these parameters can be configured in the `platform_config()` of the projects folder.

3. If package data was indicated then the device will communicate to Cerberus that it will send the GetPackageData command in the response to RequestUpdate along with the length of any metadata the device needs Cerberus to retain. The device will proceed to issue the GetPackageData command, if need be, with Cerberus responding with the package data read from flash memory. Once all the package data is transferred Cerberus will then issue the GetDeviceMetaData command, again if need be.

4. After the GetPackageData and GetDeviceMetaData commands, Cerberus will transfer the component table to the device via the PassComponentTable command. During the response to each PassComponentTable command the device will indicate its comparability with the component. The implementation does not assume what to do if there is an error in compatibility.

5. After the component table has been passed to the device, Cerberus will issue the UpdateComponent command telling the firmware device which component will be updated next. Similar to PassComponentTable, the device will respond with compatibility codes.

6. Cerberus will now wait to receive RequestFirmwareData commands from the device. Upon each command Cerberus will respond with the portion of the firmware component image specified by the device.

7. The device will issue the TransferComplete command once it obtains the entire component image or it there was an error during the transfer. It could be that the function simply exits or that Cerberus will issue a CancelUpdateComponent command to the device.

8. The current handler will simply wait for a certain number of milliseconds for the VerifyComplete command and exit if a time out is reached. A more robust mechanism is left up to the reader and could include something like periodically sending the GetStatus command to the device to pull the status of the verification.

9. After verification is completed Cerberus will again wait for the device to apply the firmware image. Just like VerifyComplete a more robust mechanism of waiting for the ApplyComplete command is needed and the apply result that is apart of the command.

10. Steps 5 through 9 are repeated for every firmware component that Cerberus needs to update.

11. After all firmware images have been transferred, verified, and applied, Cerberus will issue the ActivateFirmware command. The command contains a boolean flag telling the device whether to activate any self-contained components specified during the UpdateComponent command.

Please note that before sending the mctp message `AESEncryption()` encrypts the message using a secret key gnerated by `secretKey()`. This function takes secret key, message plus a timestamp, and use AES-GCM-256 method to encrypt the message and loads into *ciphertext* parameter. Each message then signed with a digital signature generate by `generateDSA()` function. Once the FD receives the message it verifies the signature. `AESDecryption()` function takes encrypted message (*ciphertext*), secret key (*secret*) as input, decrypts it and loads the message to the provided *plaintext* buffer. In the current implementation this metadata is written to

a region in flash although depending on the metadat, it could be written to a structure or any other volatile memory.

S-RFUP Operating as FD Flow

1. Cerberus will wait for the first command from the UA. If this command was one of the two inventory commands then Cerberus knows it needs to receive a second inventory command from the UA.

2. After handling the inventory commands, Cerberus processes the RequestUpdate command, deciding based on the UA's indication whether to send the GetPackageData command and specifying the metadata length to be retained, typically stored on flash but adjustable to other storage forms.

3. After the RequestUpdate command Cerberus will proceed to issue the GetPackageData command and respond to the GetDeviceMetaData commands if need be.

4. After receiving package data and transferring metadata, Cerberus will now wait to receive the component table from the UA via PassComponentTable commands.

5. Cerberus now receives the UpdateComponent command indicating which component in the table should be updated next. Cerberus responds back with another compatibility code that requires additional checking.

6. Cerberus will now issue the RequestFirmwareData command with the offset and length of the portion of the firmware image Cerberus is requesting. The offset and length is saved in the manager and upon receiving a response Cerberus will immediately write the portion of the image to flash memory.

7. Once component image has been transferred Cerberus will issue the TransferComplete command. The handler assumes a successful transfer with no additional checks being performed for the result field of TransferComplete.

8. Cerberus will now perform a verification of the received image. Cerberus issues the GetMetaData command with the UA responding with hash codes or signatures of the firmware image. Cerberus would then take these and compare them to the hash codes or signatures it generated with the received component image.

9. After verification is successfully completed, Cerberus will now apply the firmware image. However, because in the current implementation the firmware image is immediately written to flash the apply stage become semi-obsolete. As such, Cerberus will simply send a successful ApplyComplete command to the UA.

10. Steps 5 through 9 are repeated for every component that needs to be updated.

11. Finally, Cerberus expects to receive the ActivateFirmware command. How the firmware is activated depends on the system specifications and user preferences . The estimated time for activation field in Cerberus's response to ActivateFirmware can be configured in the `platform_config()` on the projects folder.

The verification mechanism is left up to the user to implement and subsequently the assignment of the result field in the VerifyComplete command (by default set to success). In our case, S-RFUP issues the GetMetaData command with the UA responding with signatures[5] of the firmware image. S-RFUP would then take these and compare them to the DSA signature it generated with the received component image. Additionally, how the firmware is activated depends on the system specifications and user preferences.

5.3.4 Exception and Error Handling

We have done an extensive software testing for the proposed protocol (S-RFUP). Throughout the development we have introduced several methods to handle errors of the compilation code, time-out exception for each PLDM command and tested the software in various test scenarios that shows the capability of the proposed framework.

[5]DSA with keys derived from ECC curve25519 has been used to generate signatures.

Error completion codes

For each command, we have designed a specific structure to handle the response based on the compilation codes returned by a $pldm$ message from UA/FD. Table 5.2 shows some of the PLDM commands and various scenarios of compilation code and they to handle the error. Let's say if UA send a RequestUpdate $pldm$ message to FD and FD is currently on another firmware update process, the response $pldm$ would contain a compilation_code = *ALREADY_IN_UPDATE_MODE* and a return value of 0x81. If the FD can not do a firmware update right now it will send a response with compilation_code = *RETRY_REQUEST_UPDATE*. Similarly, we have designed error handling capabilities of each command for all possible states of FD.

Timing specification

Command Name	Completion Codes	Return Value	Return By	Descriptions
QDI	PBC	0x00	FD	Executed successfully.
RU	PBC	0x00	FD	Executed successfully.
	AIUM	0x81		Already in update mode.
	UTIU	0x8A		Unable to enter update mode.
	RRU	0x8E		Requests a retry of the RequestUpdate command, needing more time to prepare.
RFD	PBC	0x00	UA	Executed successfully
	ITL	0x83		Image portion > MaxTransferSize .
	CNE	0x88		Command is not expected in the sequence.
	DOFR	0x89		Image portion offset exceeds the range.
	RRFD	0x91		Component image portion is not available.
	CP	0x87		When CancelUpdate initiated by FD previously.

Table 5.2: Command Responses and Descriptions for Firmware Updates. PBC: PLDM_BASE_CODES; AIUM: ALREADY_IN_UPDATE_MODE; UTIU: UNABLE_TO_INITIATE_UPDATE; RRU: RETRY_REQUEST_UPDATE; ITL: INVALID_TRANSFER_LENGTH; CNE: COMMAND_NOT_EXPECTED; DOFR: DATA_OUT_OF_RANGE; RRFD: RETRY_REQUEST_FW_DATA; CP: CANCEL_PENDING;

A timing specification has been designed for every compilation codes and time-out exceptions as illustrated in Figure 5.9. For RequestUpdate response message if compilation_code = *RETRY_REQUEST_UPDATE* sent by FD, it will assign an UA_T4[6] time specification for the pro-

[6]For UA_T4 the $minTime = 1s$ and $MaxTime = 5s$

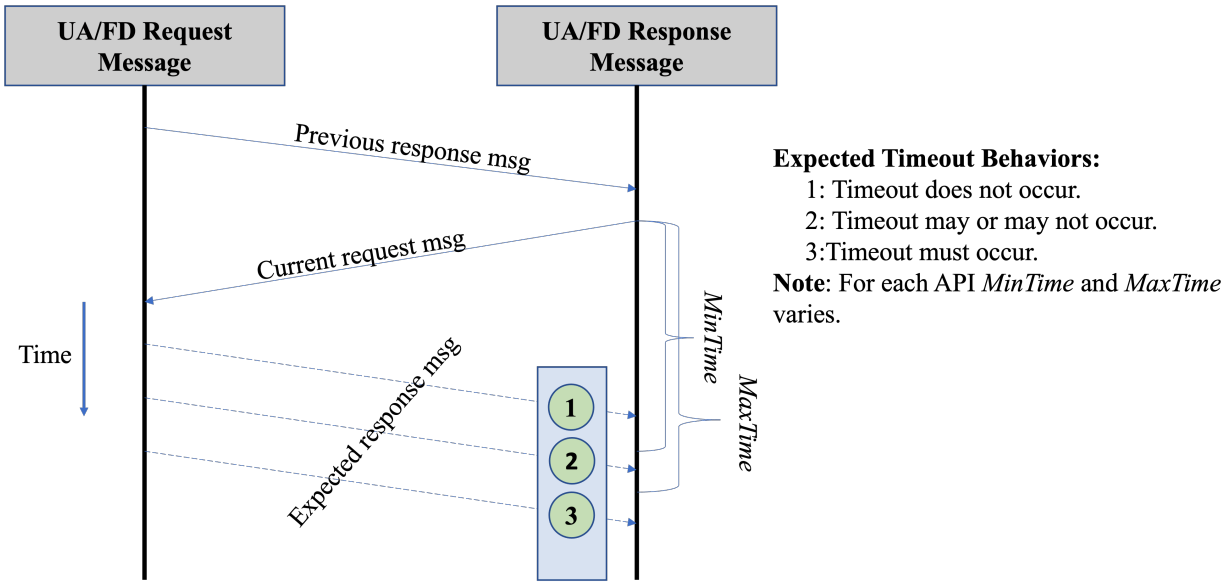

Figure 5.9: Timing specification.

cess. It means the amount of time to wait before UA re-sends a RequestUpdate PLDM command after receiving the previous response. There are also, GetPackageData timeout ($1s \leq$ UA_T5 $\leq 5s$), Update mode IDLE timeout for FD ($60s \leq$ FD_T1 $\leq 120s$) and several others that we have specified. A detailed documentation about the timing specifications will be provided with the source code.

We have also designed several test scenarios to check various failure that can occur during firmware update process such as, if the UA/FD loses connections during update process then the program will wait until the timer specified time (UA_T7 or FD_T5) or if the connection is not back, it will revert back to the previous IDLE state and throws a timeout exception (GT_T1). The S-RFUP is thoroughly tested and validated to handle unexpected behaviours during firmware update process.

5.4 Evaluation and Discussion

We developed various experimental test scenarios to evaluate our framework for correctness, consistency and performance. We run our experiments on 2 virtual Linux servers. The client-side (assumed as FD) has a 5000 MHz 12th Gen Intel(R) Core(TM) i7-12700K processor, x86_64

architecture, 20 CPUs, and UA (as server) in Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz, x86_64 architecture, 12 CPU(s).

All the S-FRUP PLDM firmware update libraries are available on GitHub[7]. All user guidelines, API descriptions, test results, and setup manuals are publicly available on GitHub.

In order to validate that the our protocol is working as it is supposed to, we used Project Cerberus with S-RFUP libraries (server platform) and '*OpenBMC/libpldm*' library as an Update Agent (UA) or server and a HRoT that is a micro-controller containing Project Cerberus with S-RFUP libraries and '*OpenBMC/libpldm*' library as a Firmware Device or client. We introduce '*cmd_chanel_tcp*' that uses a TCP socket for communication between UA and FD. It has `initialize_global_server_socket()` function that works with '*pldm_fwup_crypto*' & '*cmd_interface_pldm*' library to send the encrypted $mctp$ packets to UA/FD. As designed and expected, our protocol delivers the anticipated results, with successful operations observed across both UA and FD.

Figures 5.10 and 5.11 provide visual confirmation of these outputs.

Figures 5.10 and 5.11 show the unit test output for the Update Agent and Firmware Device respectively. Each line represents the exchange and processing of a PLDM command between the UA and FD over the c-socket mentioned previously. The last two lines represent a full uninterrupted firmware update with the second of the two tests not including the inventory commands. As explained previously, the firmware component images reside in the Firmware Update Package where they are retrieved as needed using Cerberus's flash module. Since we are compiling and evaluating on Linux, however, a virtual flash module was created to simulate that functionality using disk I/O. A Python script, located in the '*tools*' directory, generates a 4GB binary file divided into sections to simulate different flash regions: one for package data, one for meta data, and one for each two firmware components. These regions are populated with random bytes.

We tested the PLDM type commands for both successful completion and for error handling. For example, RequestUpdate consists of two tests: the first in which the FD successfully executes

[7]https://github.com/AMIProject0/Project-Cerberus-PLDM/tree/master

```
tokyo:~/Rakesh/PLDM/Project-Cerberus-PLDM/Cerberus-UA/build$ ./cerberus-linux-unit-tests
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_query_device_identifiers_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_firmware_parameters_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_update_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_update_already_in_update_mode
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_pass_component_table_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_pass_component_table_not_in_update_mode
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_update_component_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_update_component_not_in_update_mode
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_transfer_complete_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_transfer_complete_command_not_expected
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_transfer_complete_generic_transfer_error
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_verify_complete_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_verify_complete_command_not_expected
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_apply_complete_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_apply_complete_command_not_expected
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_activate_firmware_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_activate_firmware_activation_not_required
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_status_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_cancel_update_component_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_cancel_update_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_firmware_data_50_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_firmware_data_100_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_firmware_data_500_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_request_firmware_data_1_mb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_package_data_50_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_package_data_100_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_package_data_500_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_package_data_1_mb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_device_meta_data_50_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_device_meta_data_100_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_device_meta_data_500_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_device_meta_data_1_mb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_meta_data_50_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_meta_data_100_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_meta_data_500_kb_success
pldm_fwup_protocol_ua_commands: pldm_fwup_protocol_ua_commands_test_get_meta_data_1_mb_success
pldm_fwup_handler_ua: pldm_fwup_handler_ua_test_run_update_ua
pldm_fwup_handler_ua: pldm_fwup_handler_ua_test_run_update_ua_no_inventory_cmds
linux: linux_teardown
.....................................

OK (39 tests)
```

Figure 5.10: UA/Server side terminal output.

the command and indicates so to the UA and the second in which the FD responds that it is already in update mode. Additionally, for the multi-part transfer commands and the full firmware update, we have tested the frameworks capability with different firmware image payload sizes. A more detailed discussion, where we illustrated the firmware packages, errors, and time to execute each PLDM command in our system with 50KB image size and also for various payloads will be next.

Table 5.3 shows PLDM commands and corresponding various package header and payload values for a 50 KB update. Due to space constrained we could not show all the PLDM commands values. A detailed report will be provided with the source code. The time mentioned in the Table 5.3 is total time taken for the particular command to finish its operation where UT is time (in milli-seconds) taken for UA and FT is time (in milli-seconds) for FD to execute that PLDM command.

```
tokyo:~/Rakesh/PLDM/Project-Cerberus-PLDM/Cerberus-FD/build$ ./cerberus-linux-unit-tests
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_query_device_identifiers_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_firmware_parameters_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_update_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_update_already_in_update_mode
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_pass_component_table_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_pass_component_table_not_in_update_mode
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_update_component_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_update_component_not_in_update_mode
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_transfer_complete_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_transfer_complete_command_not_expected
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_transfer_complete_generic_transfer_error
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_verify_complete_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_verify_complete_command_not_expected
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_apply_complete_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_apply_complete_command_not_expected
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_activate_firmware_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_activate_firmware_activation_not_required
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_status_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_cancel_update_component_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_cancel_update_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_firmware_data_50_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_firmware_data_100_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_firmware_data_500_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_request_firmware_data_1_mb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_package_data_50_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_package_data_100_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_package_data_500_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_package_data_1_mb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_device_meta_data_50_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_device_meta_data_100_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_device_meta_data_500_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_device_meta_data_1_mb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_meta_data_50_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_meta_data_100_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_meta_data_500_kb_success
pldm_fwup_protocol_fd_commands: pldm_fwup_protocol_fd_commands_test_get_meta_data_1_mb_success
pldm_fwup_handler_fd: pldm_fwup_handler_fd_test_start_update_fd
pldm_fwup_handler_fd: pldm_fwup_handler_fd_test_start_update_fd_no_inventory_cmds
linux: linux_teardown
.....................................

OK (39 tests)
```

Figure 5.11: FD/Client side terminal output.

When UA sends an inventory command QueryDeviceIdentifiers and GetFirmwareParameters to enquire about Firmware identity, it sends back the package header information: { DeviceIdentifierLength: 8, DescriptorCount: 4, Descriptors: { 4321, 8765, 6789, 2109 }, - ComponentCount: 2, ActiveComponentSetVersionStringType: 1, ActiveComponentSetVersionStringLength: 13, PendingComponentSetVersionStringType: 1, ...}. Once UA varifies the header information, it will send a RequestUpdate. The information UA send to FD for RequestUpdate is listed in table. Each time the PLDM sends 1024 byte data – MaximumTransferSize: 1024. When the FD request for image data through RequestFirmwareData command, it will request for a ComponentImagePortion which is MaximumTransferSize. In response UA sends the data noted as Length: 1024 and Offset: 50176 (data left to send). The time mentioned in the table is total time taken for the particular command to finish its operation.

As outlined in section 5.2, the protocol functions are responsible for interpreting and constructing the PLDM messages encapsulated in the MCTP payloads and their values are shown in UA

Request & FD Response column of Table 5.3. These functions uses the *'OpenBMC/libpldm'* for easy serialization and deserialization of the PLDM messages.

For example, the FD must generate a RequestFirmwareData request type PLDM message containing a specified length and offset of the component image. These two values are assigned by the FWUP Manager which who's reference is passed to `pldm_fwup_generate_request_firmware_data_request()` along with a reference to the MCTP payload. This function subsequently calls libpldm's `encode_request_firmware_data_req()` which performs header packing, type punning, and endianness conversion. The result is the MCTP payload containing the correct PLDM header, the length, and the offset. Similarly, when the UA receives the RequestFirmwareData command it will need to deserialize the MCTP payload using libpldm's `decode_request_firmware_data_req()` extracting the length and offset which the UA then uses to read that portion of the firmware image from the flash.

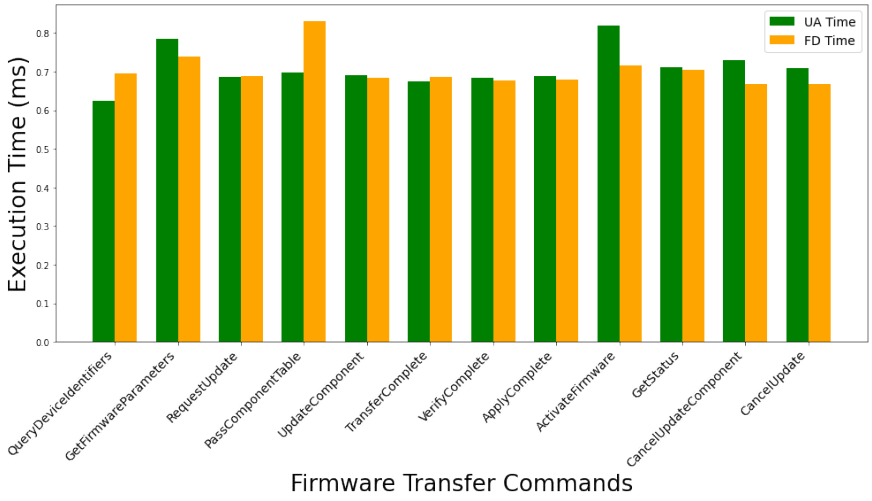

Figure 5.12: UA Time vs FD Time for Different Inventory and Update Commands (50 kB)

Figure 5.15 shows the time taken for both UA and FD for inventory and update type PLDM commands. Also, Figure 5.16 shows the execution time of each PLDM command of transfer type. By comparing Figures 5.15 and 5.16, it is evident that the execution time of transfer type commands are higher than inventory and update type commands.

Command	UA Request	UT (ms)	FD Response	FT (ms)
QDI	None	NA	CompletionCode: 0 DeviceIdentifiersLength: 8 DescriptorCount: 4 Descriptors: {4321, 8765, 6789, 2109}	0.695
RU	MaximumTransferSize: 1024 NumberofComponents: 2 MaximumOutstanding TransferRequests: 1 PackageDataLength: 5120 ComponentImageSetVersion StringType: 1 ComponentImageSetVersion StringLength: 13 ComponentImageSetVersion String: cerberus_v2.0	0.687	CompletionCode: 0 FirmwareDeviceMeta DataLength: 5120 FDWillSendGetPackage DataCommand: 1	0.689
RFD	Offset: 50176 Length: 1024	35.997	CompletionCode: 0 ComponentImagePortion: MaximumTransferSize bytes	36.275

Table 5.3: Test Results for Inventory Commands for a 50KB Update. | QDI: QueryDeviceIdentifiers; RU: RequestUpdate; PCT: PassComponentTable; RFD: RequestFirmwareData;

Table 5.3 shows various PLDM command and their values for a 50KB image size. As described in section 3.2.1 figure 5.3, each Firmware Update Package consists for Package Header and Payload. Due to space constrained we could not show all the PLDM commands values. When UA sends an inventory command QueryDeviceIdentifiers and GetFirmwareParameters to enquire about Firmware identity, it sends back the package header information: {DeviceIdentifiersLength: 8, DescriptorCount: 4, Descriptors: { 4321, 8765, 6789, 2109 }, - ComponentCount: 2, ActiveComponentSetVersionStringType: 1, ActiveComponentSetVersionStringLength: 13, Pending-

Test	UA Request	UA Time (ms)	FD Response	FD Time (ms)
QueryDeviceIdentifiers	Request: None	NA	Response: CompletionCode: 0 DeviceIdentifiersLength: 8 DescriptorCount: 4 Descriptors: {4321, 8765, 6789, 2109}	0.695
GetFirmwareParameters	Request: None	NA	Response: CompletionCode: 0 CapabilitiesDuringUpdate: 0 ComponentCount: 2 ActiveComponentSetVersionStringType: 1 ActiveComponentSetVersionStringLength: 13 PendingComponentSetVersionStringType: 1 PendingComponentSetVersionStringLength: 13 ActiveComponentSetVersionString: cerberus_v1.0 PendingComponentSetVersionString: cerberus_v2.0 ComponentParameterTable: {10, 0, 45, 115, . . . }	0.739

Table 5.4: Test Results for Inventory Commands for a 50KB Update

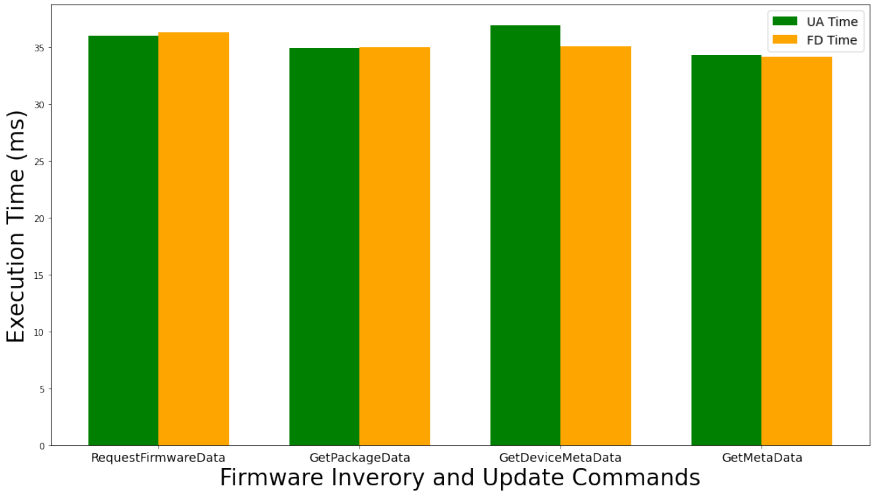

Figure 5.13: UA Time vs FD Time for Different Transfer Commands (50 KB)

ComponentSetVersionStringType: 1, ...}. Once UA varifies the header information, it will send a RequestUpdate. The information UA send to FD for RequestUpdate is listed in table. Each time the PLDM sends 1024 byte data – MaximumTransferSize: 1024. When the FD request for image data through RequestFirmwareData command, it will request for a ComponentImagePortion which is MaximumTransferSize. In response UA sends the data noted as Length: 1024 and Offset: 50176. (data left to send). The time mentioned in the table is total time taken for the particular command to finish its operation.

Test	UA Request	UA Time (ms)	FD Response	FD Time (ms)
RequestUpdate	Request: MaximumTransferSize: 1024 NumberofComponents: 2 MaximumOutstandingTransferRequests: 1 PackageDataLength: 5120 ComponentImageSetVersionStringType: 1 ComponentImageSetVersionStringLength: 13 ComponentImageSetVersionString: cerberus_v2.0	0.687	Response: CompletionCode: 0 FirmwareDeviceMetaDataLength: 5120 FDWillSendGetPackageDataCommand: 1	0.689
PassComponentTable	Request: TransferFlag: 4 ComponentClassification: 9 ComponentIdentifier: 29490 ComponentClassificationIndex: 190 ComponentComparisonStamp: 3780935211 ComponentVersionStringType: 1 ComponentVersionStringLength: 15 ComponentVersionString: middleware_v2.0	0.697	Response: CompletionCode: 0 ComponentResponse: 0 ComponentResponseCode: 0	0.832
UpdateComponent	Request: ComponentClassification: 10 ComponentIdentifier: 29485 ComponentClassificationIndex: 187 ComponentComparisonStamp: 3780935208 ComponentImageSize: 5120 UpdateOptionsFlags: 1 ComponentVersionStringType: 1 ComponentVersionStringLength: 13 ComponentVersionString: firmware_v2.0	0.691	Response: CompletionCode: 0 ComponentCompatibilityResponse: 0 ComponentCompatibilityResponseCode: 0 UpdateOptionFlagsEnabled: 1 EstimatedTimeBeforeSendingRequestFirmwareData : 1	0.685
TransferComplete	Request: TransferResult: 0	0.676	Response: CompletionCode: 0	0.686
VerifyComplete	Request: VerifyResult: 0	0.683	Response: CompletionCode: 0	0.677
ApplyComplete	Request: ApplyResult: 0 ComponentActivationMethodsModification: 0	0.689	Response: CompletionCode: 0	0.68
ActivateFirmware	Request: SelfContainedActivationRequest: 1	0.82	Response: CompletionCode: 0 EstimatedTimeForSelfContainedActivation: 1	0.716
GetStatus	Request: None	NA	Response: CompletionCode: 0 CurrentState: 1 PreviousState: 0 AuxState: 3 AuxStateStatus: 0 ProgressPercent: 101 ReasonCode: 0 UpdateOptionsFlagsEnabled: 1	0.705
CancelUpdateComponent	Request: None	NA	Response: CompletionCode: 0	0.669
CancelUpdate	Request: None	NA	Response: CompletionCode: 0 NonFunctioningComponentIndication: 0 NonFunctioningComponentBitmap: 0	0.668

Table 5.5: Test Results for FD and UA Commands for 50KB Update

Test	UA Request	UA Time (ms)	FD Response	FD Time (ms)
RequestFirmwareData	Request: Offset: 50176 Length: 1024	35.997	Response: CompletionCode: 0 ComponentImagePortion: MaximumTransferSize bytes	36.275
GetPackageData	Request: DataTransferHandle: 50176 TransferOperationFlag: 1	34.939	Response: CompletionCode: 0 NextDataTransferHandle: 0 TransferFlag: 4 PortionOfPackageData: MaximumTransferSize bytes	35.021
GetDeviceMetaData	Request: DataTransferHandle: 50176 TransferOperationFlag: 1	36.9	Response: CompletionCode: 0 NextDataTransferHandle: 0 TransferFlag: 4 PortionOfMetaData: MaximumTransferSize bytes	35.102
GetMetaData	Request: DataTransferHandle: 50176 TransferOperationFlag: 1	34.347	Response: CompletionCode: 0 NextDataTransferHandle: 0 TransferFlag: 4 PortionOfMetaData: MaximumTransferSize bytes	34.187

Table 5.6: Test Results for 50 KB Transfers

Figure 5.15 shows the time taken for both UA and FD for inventory and update type PLDM commands. Also, Figure 5.16 shows the execution time of each PLDM command of transfer type. By comparing Figures 5.15 and 5.16, it is evident that the execution time of transfer type commands are higher than inventory and update type commands.

We have tested our framework with various image sizes:{ 50KB, 100KB, . . . , 500MB, 1GB }.Figure 5.17 shows a bar chart of execution time vs different firmware image sizes with and without inventory commands (as it is not not always required for inventory commands). For, 50 KB to 1 MB the percentage increases in times are 55.68% (for UA) and 55.56% (for FD). If we compare the 50KB to 1 GB sizes, the percentage increases in times are 60194.30% (UA) and 60190.13% (FD). The growth rate is exponential. But it is reasonable as, for 1GB file, it takes only 128 seconds which is acceptable compared to state-of-the-art firmware update methods. By developing S-RFUP, we address the research question (RQ3) of how we can secure and standardize remote firmware updates across diverse device platforms while minimizing security vulnerabilities.

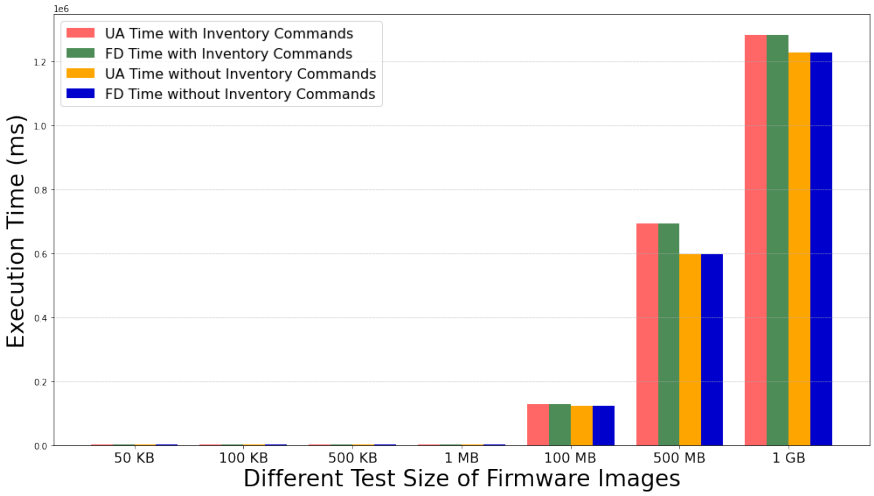

Figure 5.14: UA Time vs FD Time for Firmware Update Tests (with and without Inventory Commands).

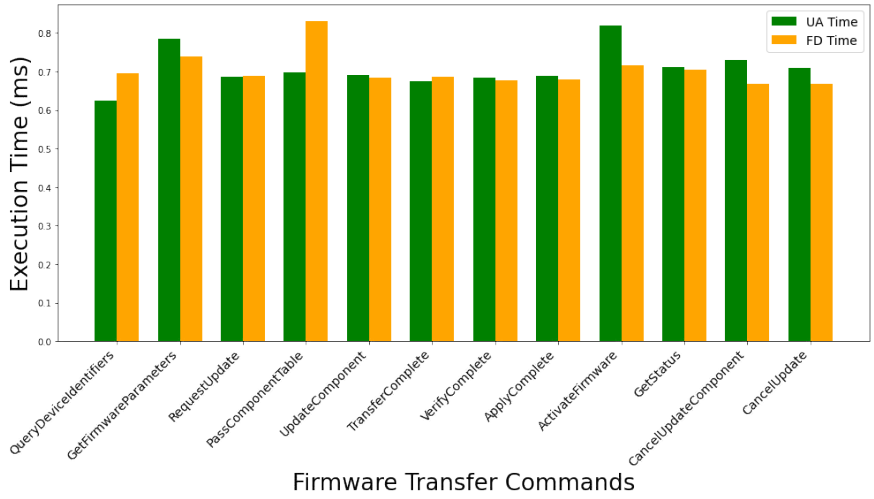

Figure 5.15: UA Time vs FD Time for Different Inventory and Update Commands (50 kB)

5.5 Security Analysis

The proposed secure firmware update protocol (S-RFUP) effectively mitigates a range of security threats through a combination of robust encryption techniques, rigorous verification procedures, and systematic error handling mechanisms. The following analysis details how the identified threats are addressed using this approach and how the essential security properties are maintained. To prevent service unavailability (due to dos attack) and ensure continuous operation during the firmware update process, the protocol incorporates several exceptions and error handling mechanisms. These mechanisms are designed to catch and manage unexpected errors or attacks that

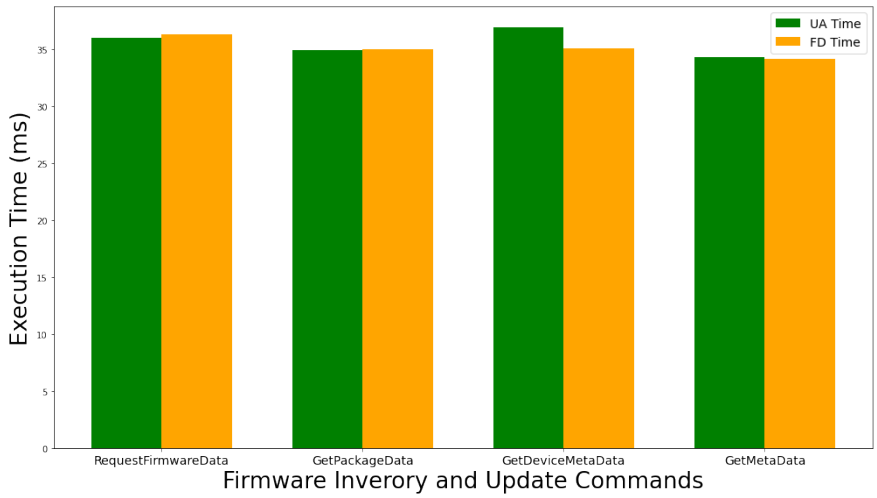

Figure 5.16: UA Time vs FD Time for Different Transfer Commands (50 KB)

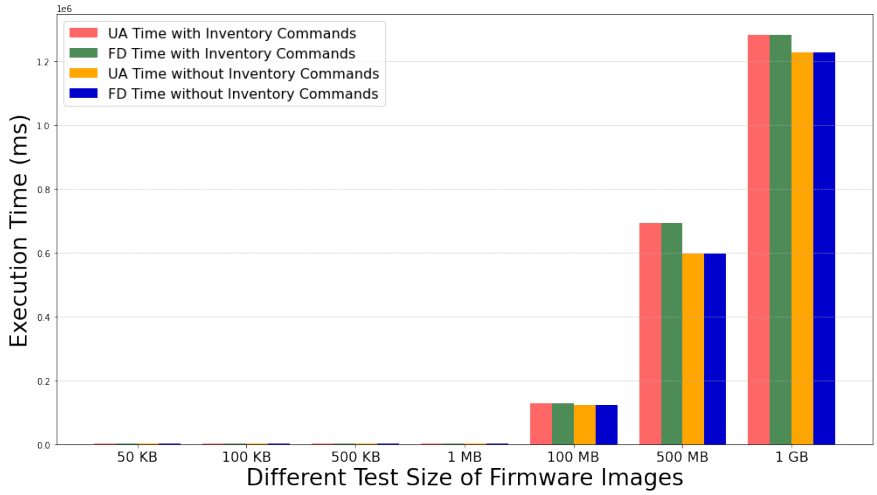

Figure 5.17: UA Time vs FD Time for Firmware Update Tests (with and without Inventory Commands)

could lead to server or service unavailability. By handling such scenarios promptly, the system avoids unwanted interruptions or crashes, maintaining service availability and reliability.

The communication between the UA and FD is protected using advanced encryption methods. Specifically, the Management Component Transport Protocol (MCTP) messages are encrypted using the S-RFUP 'crypto' library framework, which utilizes the Advanced Encryption Standard – Galois/Counter Mode – with a 256-bit key (AES-GCM 256). This encryption method is resistant to a variety of attacks, including *Known Plaintext Attack, (KPA)*, *Chosen Plaintext Attack (CPA)*, *Chosen Ciphertext Attack (CCA)*, and *Ciphertext-Only Attack (COA)*. By encrypting all communi-

cations, the protocol ensures that any intercepted data remains inaccessible to attackers, protecting sensitive information such as firmware/device information. Thus, data integrity is maintained by ensuring as data packets are delivered to the Recipient without any alterations. The protocol achieves this through the use of AES-GCM encryption, which includes built-in integrity checks to verify that the data has not been tampered with during transmission.

To mitigate replay attacks, the protocol employs unique session keys and *Lamport* timestamps. During each session, the UA and FD generate a new shared AES-GCM key using the ECDH key agreement protocol. This ensures that each session is encrypted with a unique key, preventing attackers from reusing intercepted messages in a different session. Thus, by employing *Lamport* timestamps we can verify the freshness of messages, ensuring that old messages cannot be replayed to disrupt the firmware update process, this gives the assurance of data freshness.

The protocol incorporates the Digital Signature Algorithm (DSA) with keys derived from ECC curve to authenticate legitimacy of the entities. This ensures that the recipient can verify the authenticity of the sender, protecting against impersonation attacks. By using digital signatures, the protocol ensures that only legitimate UA and FD entities can participate in the firmware update process, effectively preventing MITM attacks where an attacker could intercept and alter communications. This process confirms the authenticity of the data source and the integrity of the data itself. The use of ECDH for key exchange, combined with AES-GCM for encryption, provides robust protection against *Adaptive Chosen-Plaintext* and *Chosen-Ciphertext Attacks*. By generating a new secret key for each session, the protocol ensures that attackers cannot use previous encryption or decryption to infer the current encryption key. This dynamic key management system enhances security by preventing key reuse and complicating any attempts to compromise the communication through adaptive attacks.

This comprehensive security framework provides robust protection against eavesdropping, replay & MitM attacks, credential theft, and adaptive attacks, ensuring a secure and reliable remote firmware update mechanism. S-RFUP successfully maintains the essential security properties of data integrity, data authentication, data confidentiality, and data freshness, providing a secure,

and reliable process for remote updates. The security analyses in Sections 4.5 and 5.5 address the last research question (RQ4), which examines the extent to which the proposed Protection in Transit (PIT) and Secure Remote Firmware Update Protocol (S-RFUP) enhance security and standardization during transit and remote firmware updates, as well as how their effectiveness can be empirically validated.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The exponential growth in hardware integrated circuits (ICs) inevitably raises parallel concerns regarding device security. Through our research, we have addressed a critical aspect of this concern - the protection of products during transit through third-party mediums. The protocol we have devised successfully safeguards devices from the potential introduction of trojans - a principal reason for the increasing focus on Hardware Root of Trust (HRoT) in contemporary industrial discourse.

Our modified version of the Cerberus embedded framework, PIT-Cerberus, extends security capabilities to any microcontroller or microprocessor, enhancing device security significantly. The robustness of our solution lies in its ability to protect the device at its most vulnerable state - during transit - by ensuring that only a verified user can unlock and boot the device's BIOS, thereby thwarting any attempts at introducing trojans.

In this work, we propose S-RFUP as a uniform framework for secure remote firmware updates across mutitude of platforms. S-RFUP builds upon Project Cerberus hardware root of trust capabilities by integrating the industry-standard protocols PLDM & MCTP and conventional cryptographic protocols to ensure a secure, reliable, interoperable and easily manageable firmware update process. The implementation has been rigorously tested, validating its resilience against various security concerns and demonstrating its robustness in a controlled environment. We plan to open-source S-RFUP libraries.

6.2 Future Work

Looking ahead, we intend to further enhance the utility and security features of PIT-Cerberus. A notable direction of our future work involves porting the hardware-agnostic PIT-Cerberus to a

microchip-specific I2C protocol, to establish serial communication for performing the Key Exchange Scheme. We remain committed to continuously refining our protocol to address an ever-evolving landscape of hardware security challenges, thus ensuring the safeguarding of devices in an increasingly interconnected world.

Future work involves enhancing the protocol's performance and security through implementing parallel firmware updates to increase efficiency, porting the hardware-agnostic S-RFUP to the microchip-specific I2C protocol, and validating the protocol across different firmware ecosystems to ensure robust performance and compatibility.

Bibliography

[1] Jack Ganssle. *The firmware handbook*. Elsevier, Burlington, CO, USA, 1st edition, 2004.

[2] Mikhail Krichanov and Vitaly Cheptsov. UEFI virtual machine firmware hardening through snapshots and attack surface reduction. In *2021 Ivannikov Ispras Open Conference (ISPRAS)*, pages 30–36. IEEE, 2021.

[3] David Cooper, William Polk, Andrew Regenscheid, Murugiah Souppaya, et al. BIOS protection guidelines. *NIST Special Publication*, 800:147, 2011.

[4] Anibal L Sacco and Alfredo A Ortega. Persistent BIOS infection. In *CanSecWest Applied Security Conference*, 2009.

[5] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[6] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL:TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073, 2020.

[7] Lucian Constantin. New exploits can bypass Secure Boot and modern UEFI security protections, August 12, 2022.

[8] Alina A Zharkova. Application Encryption with Trusted Platform Module to Implement Standards in Windows 11 Environment. In *2023 Seminar on Information Computing and Processing (ICP)*, pages 239–241. IEEE, 2023.

[9] Lawrence Abrams. How to bypass the Windows 11 TPM 2.0 requirement, July 2, 2021.

[10] Dimitar Georgiev Vrachkov and Dimitar Georgiev Todorov. Research of the systems for Firmware Over The Air (FOTA) and Wireless Diagnostic in the new vehicles. In *2020 XXIX International Scientific Conference Electronics (ET)*, pages 1–4. IEEE, 2020.

[11] Mohammad Shafeul Wara and Qiaoyan Yu. New replay attacks on zigbee devices for internet-of-things (iot) applications. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–6. IEEE, 2020.

[12] Phi Tuong Lau and Stefan Katzenbeisser. Firmware-Based DoS Attacks in Wireless Sensor Network. In *European Symposium on Research in Computer Security*, pages 214–232. Springer, 2023.

[13] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities. In *USENIX Security Symposium*, 2023.

[14] Yipeng Zhang, Ye Li, and Zhoujun Li. Aye: A trusted forensic method for firmware tampering attacks. *Symmetry*, 15(1):145, 2023.

[15] Zachry Basnight, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76–84, 2013.

[16] Levon Keleman, Danijel Matić, Miroslav Popović, and Ivan Kaštelan. Secure firmware update in embedded systems. In *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*, pages 16–19. IEEE, 2019.

[17] Ang Cui, Michael Costello, and Salvatore Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *20th Annual Network & Distributed System Security Symposium*, 2013.

[18] Ryan Tsang, Doreen Joseph, Qiushi Wu, Soheil Salehi, Nadir Carreon, Prasant Mohapatra, and Houman Homayoun. FANDEMIC: Firmware Attack Construction and Deployment on Power Management Integrated Circuit and Impacts on IoT Applications. In *NDSS*, 2022.

[19] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91):1–91, 2015.

[20] Rakesh Podder, Jack Sovereign, Indrajit Ray, Madhan B Santharam, and Stefano Righi. The PIT-Cerberus Framework: Preventing Device Tampering During Transit. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*, pages 584–595. IEEE, 2024.

[21] Rakesh Podder, Mahmoud Abdelgawad, Indrakshi Ray, Indrajit Ray, Madhan Santharam, and Stefano Righi. Correctness and Security Analysis of the Protection in Transit (Pit Protocol). *Available at SSRN 4980331*, 2024.

[22] Yutian Gui, Ali Shuja Siddiqui, and Fareena Saqib. Hardware based root of trust for electronic control units. In *SoutheastCon 2018*, pages 1–7. IEEE, 2018.

[23] Microsoft. Project Cerberus, 2018.

[24] Rakesh Podder, Tyler Rios, Indrajit Ray, Presanna Raman, and Stefano Righi. S-RFUP: Secure Remote Firmware Update Protocol. In *International Conference on Information Systems Security*, pages 42–62. Springer, 2025.

[25] Bryan Kelly. Project Cerberus Security Architecture Overview Specification. *Open Compute Project*, 2017.

[26] DMTF. Platform Level Data Model (PLDM) for Firmware Update Specification 1.0.1. *DSP0267*, 2009.

[27] Rafal Wojtczuk and Alexander Tereshkin. Attacking intel bios. *BlackHat, Las Vegas, USA*, 2009.

[28] Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM rootkits: a new breed of os independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–12, 2008.

[29] Alexander Maassen. Network bluepill-stealth router-based botnet has been ddosing dronebl for the last couple of weeks, 2009.

[30] Barnaby Jack. Jackpotting automated teller machines redux. *Black Hat USA*, 2010.

[31] Charlie Miller. Battery firmware hacking. *Black Hat USA*, pages 3–4, 2011.

[32] Andrei Costin. Hacking MFPs. In *The 28th Chaos Communication Congress*, 2011.

[33] Steve Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Jeremiah Blocki, Kevin Fu, and Dawn Song. Take Two Software Updates and See Me in the Morning: The Case for Software Security Evaluations of Medical Devices. In *HealthSec*, 2011.

[34] Silvie Schmidt, Mathias Tausig, Matthias Hudler, and Georg Simhandl. Secure firmware update over the air in the internet of things focusing on flexibility and feasibility. In *Internet of Things Software Update Workshop (IoTSU). Proceeding*, 2016.

[35] Zachry Basnight, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76–84, 2013.

[36] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE communications surveys & tutorials*, 18(3):2027–2051, 2016.

[37] John F Miller. Supply chain attack framework and attack patterns. *The MITRE Corporation, MacLean, VA*, 2013.

[38] Andreas Fuchs, Christoph Krauß, and Jürgen Repp. Advanced remote firmware upgrades using TPM 2.0. In *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30-June 1, 2016, Proceedings 31*, pages 276–289. Springer, 2016.

[39] Aurélien Vasselle, Philippe Maurine, and Maxime Cozzi. Breaking mobile firmware encryption through near-field side-channel analysis. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, pages 23–32, 2019.

[40] Shadaab Kawnain Bashir, Rakesh Podder, Sarath Sreedharan, Indrakshi Ray, and Indrajit Ray. Resiliency graphs: Modelling the interplay between cyber attacks and system failures through ai planning. In *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, pages 292–302. IEEE, 2024.

[41] Indrajit Ray, Sarath Sreedharan, Rakesh Podder, Shadaab Kawnain Bashir, and Indrakshi Ray. Explainable ai for prioritizing and deploying defenses for cyber-physical system resiliency. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 184–192. IEEE, 2023.

[42] Rakesh Podder and Sudipto Ghosh. Impact of white-box adversarial attacks on convolutional neural networks. In *2024 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, pages 1–9. IEEE, 2024.

[43] Ian Haken. Bypassing Local Windows Authentication to Defeat Full Disk Encryption. *Black Hat Europe*, 2015.

[44] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*, pages 1362–1380. IEEE, 2019.

[45] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *HotSec*, 2006.

[46] Ryan Tsang, Doreen Joseph, Qiushi Wu, Soheil Salehi, Nadir Carreon, Prasant Mohapatra, and Houman Homayoun. FANDEMIC: Firmware Attack Construction and Deployment on Power Management Integrated Circuit and Impacts on IoT Applications. In *NDSS*, 2022.

[47] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72, 2010.

[48] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. Upkit: An open-source, portable, and lightweight update framework for constrained iot devices. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*, pages 2101–2112. IEEE, 2019.

[49] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. A firmware update architecture for internet of things. *Internet Requests for Comments, RFC Editor, RFC*, 9019, 2021.

[50] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. RapidPatch: firmware hotpatching for Real-Time embedded devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2225–2242, 2022.

[51] Christian Niesler, Sebastian Surminski, and Lucas Davi. HERA: Hotpatching of Embedded Real-time Applications. In *NDSS*, 2021.

[52] Youngcheul Wee and Taehwa Kim. A new code compression method for FOTA. *IEEE Transactions on Consumer Electronics*, 56(4):2350–2354, 2010.

[53] Silu Sun. Design and implementation of partial firmware upgrade, 2019.

[54] Samip Dhakal, Fehmi Jaafar, and Pavol Zavarsky. Private blockchain network for iot device firmware integrity verification and update. In *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, pages 164–170. IEEE, 2019.

[55] Bernardino Pinto Neves, Victor DN Santos, and António Valente. Innovative Firmware Update Method to Microcontrollers during Runtime. *Electronics*, 13(7):1328, 2024.

[56] Neha Jain, Swapnil G Mali, and Suhas Kulkarni. Infield firmware update: Challenges and solutions. In *2016 International Conference on Communication and Signal Processing (ICCSP)*, pages 1232–1236. IEEE, 2016.

[57] Solon Falas, Charalambos Konstantinou, and Maria K Michael. A modular end-to-end framework for secure firmware updates on embedded systems. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(1):1–19, 2021.

[58] Dustin Frisch, Sven Reißmann, and Christian Pape. An over the air update mechanism for esp8266 microcontrollers. In *Proceedings of the ICSNC, the Twelfth International Conference on Systems and Networks Communications, Athens, Greece*, pages 8–12, 2017.

[59] Farouk Mahfoudhi, Ashish Kumar Sultania, and Jeroen Famaey. Over-the-air firmware updates for constrained NB-IoT devices. *Sensors*, 22(19):7572, 2022.

[60] Ming Chang Lu, Qi Xian Huang, Min Yi Chiu, Yuan Chia Tsai, and Hung Min Sun. PSPS: A Step toward Tamper Resistance against Physical Computer Intrusion. *Sensors*, 22(5):1882, 2022.

[61] Tal Moran and Moni Naor. Basing cryptographic protocols on tamper-evident seals. *Theoretical Computer Science*, 411(10):1283–1310, 2010.

[62] FIPS Pub. Security requirements for cryptographic modules. *FIPS PUB*, 140:140–2, 1994.

[63] Todd W Arnold, Carl Buscaglia, Felix Chan, Vincenzo Condorelli, John Dayka, William Santiago-Fernandez, Nihad Hadzic, Michael D Hocker, Michael Jordan, Thomas E Morris, et al. IBM 4765 cryptographic coprocessor. *IBM Journal of Research and Development*, 56(1.2):10–1, 2012.

[64] Aman Bakshi and Yogesh B Dujodwala. Securing cloud from ddos attacks using intrusion detection system in virtual machine. In *2010 second international conference on communication software and networks*, pages 260–264. IEEE, 2010.

[65] Rakel Haakegaard and Joanna Lang. The elliptic curve diffie-hellman (ecdh). 2015.

[66] Morris J Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac, 2007.

[67] Rakesh Podder and Ranjit Kumar Barai. Hybrid Encryption Algorithm for the Data Security of ESP32 based IoT-enabled Robots. In *2021 Innovations in Energy Management and Renewable Resources (52042)*, pages 1–5. IEEE, 2021.

[68] Elaine Barker. Guideline for using cryptographic standards in the federal government: Cryptographic mechanisms. *NIST special publication*, pages 800–175B, 2016.

[69] Rakesh Podder and Jack Sovereign. Project Cerberus with PIT protocol.

[70] John Viega, Matt Messier, and Pravir Chandra. *Network security with openSSL: cryptography for secure communications*. O'Reilly Media, Inc., 2002.

[71] Internet Software Consortium. Libsodium documentation: Introduction, 2013.

[72] Peter Gutmann. Cryptlib encryption toolkit. 2008.

[73] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

Appendix A

License

Colorado State University LaTeX Thesis Template

by Elliott Forney – 2017

This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.