Politecnico di Torino

Cybersecurity Engineering

A.a. 2024/2025

Graduation Session December 2025

Design and Implementation of a Secure Software Patching Mechanism for the Space Rider Avum Orbital Module

Supervisors:

Stefano Di Carlo (DAUIN)

Alessandro Savino (DAUIN)

Maurizio Ronzoni (AVIO)

Candidate:

Simone D'Addio

Abstract

Historically, civilian space missions have employed limited security mechanisms. Telecommands, telemetry, and scientific payload data have often been transmitted over unencrypted and unauthenticated Radio Frequency (RF) channels. While this operational model was once considered acceptable, the growing threat landscape makes such an approach increasingly dangerous.

In response to these evolving needs, this thesis focuses on the design and implementation of a secure software patching mechanism for the Avum Orbital Module (AOM), part of Space Rider, the European Space Agency's reusable space transportation system. The project aims to enhance the spacecraft's ability to safely receive and apply software updates, reinforcing both its resilience and mission reliability.

The developed solution employs the AES-GCM algorithm to ensure the confidentiality, integrity, and authenticity of software patches. To maintain compliance with ESA standards, the mechanism extends the Packet Utilization Standard Service 6 (PUS6), which defines how spacecraft commands and telemetry packets are managed. By securing the update process, the system effectively mitigates the risks associated with unauthorized or corrupted software patches. Additionally, an anti-replay mechanism was implemented, ensuring that even valid, intercepted ciphertexts cannot be fraudulently or repeatedly applied. This process effectively authenticates the patch payload, along with associated data (AAD), strengthening contextual integrity.

A significant contribution of this work involved the validation of the cryptographic algorithm directly on the target spaceborne hardware. Since no prior verification was available on the specific hardware, this activity was crucial to confirm the correctness and robustness of the AES-GCM implementation under realistic operational conditions.

Additionally, a codebase analysis was carried out to map potential weaknesses by correlating the MISRA C guidelines with the MITRE Common Weakness Enumeration (CWE) framework, with the goal of exploring their relationships and supporting the potential extension of the internal coding standard.

Through these activities, the thesis contributes to introducing and strengthening cybersecurity capabilities and advancing the protection of critical spaceborne software systems, enabling a safer transition from traditionally unprotected space operations towards robust, security-aware mission architectures.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the last decade, the increasing reliance of modern societies on space-based services, such as global navigation, Earth observation, satellite communications, and scientific exploration, has turned space systems into strategic assets. Their disruption could have cascading effects on national security, critical infrastructure, and the global economy. Consequently, the cybersecurity of space systems has evolved from a niche concern to a central pillar of modern space policy.

The global cybersecurity landscape has become increasingly complex, characterized by a surge in both the sophistication and frequency of attacks. Advanced Persistent Threats (APTs), supply chain compromises, and zero-day vulnerabilities now routinely target high-value infrastructures.

The space sector, traditionally perceived as isolated, is no longer immune to these dynamics. Recent incidents, such as jamming and spoofing of satellite signals, ransomware targeting ground infrastructures, and unauthorized access to telemetry and telecommand systems, have demonstrated the tangible risks facing space operations. Some notable attack vectors include [1]:

- Ground Segment Attacks: Ground stations and mission control centers remain prime targets for network exploitation, data corruption, and unauthorized access. Outdated software, weak authentication, or physical intrusions can allow attackers to seize control of spacecraft or disrupt telemetry and telecommand flows.

- Space Segment Compromise: Spacecraft are susceptible to command intrusions, signal replay, and malware injection. Due to the lack of physical access, any unauthorized control or payload manipulation can result in mission failure or even create orbital collision risks.

- User Segment Vulnerabilities: Terminals, antennas, and modems used by end-users can be exploited through malware, ransomware, or spoofing attacks.

Insecure configurations or hardcoded credentials may expose communication links and satellite data.

- Communication Channel Threats: Uplink and downlink signals are exposed to jamming, spoofing, eavesdropping, and replay attacks. These may degrade signal integrity, inject false commands, or enable data interception, undermining mission confidentiality and availability.
- Cloud Infrastructure Risks: As satellite operations increasingly rely on cloud-based systems for data storage and coordination, threats such as data breaches, account hijacking, insider attacks, and Advanced Persistent Threats (APTs) emerge, especially in multi-tenant or misconfigured environments.
- Supply Chain Compromise: The growing use of Commercial-Off-The-Shelf (COTS) components introduces vulnerabilities such as firmware backdoors, counterfeit hardware, or malicious updates. A compromised supplier can embed persistent threats that remain undetected into orbit.

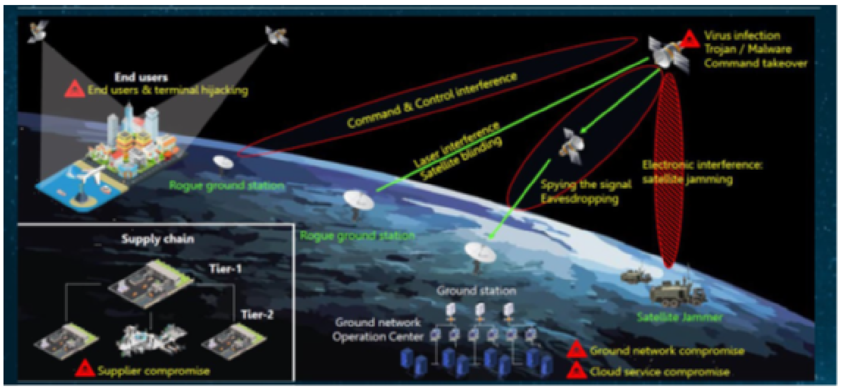


Figure 1.1: Potential Threats in Space Systems[2]

These risks collectively affect the availability, integrity, confidentiality, authenticity, and trustworthiness of space data and operations. Furthermore, the long lifecycle and physical inaccessibility of satellites exacerbate exposure to known vulnerabilities and complicate patch management once deployed.

In Europe, the growing awareness of these threats has led to the establishment of dedicated cybersecurity strategies and frameworks. The EU Cybersecurity Act, the NIS2 Directive, and sector-specific initiatives promoted by the European Space Agency (ESA) and the European Union Agency for Cybersecurity (ENISA) have emphasized the need for resilience, risk management, and continuous assurance in space missions.

Despite these advancements, significant challenges remain. Space systems often rely on legacy technologies with long development cycles, which limits their ability to adopt modern security controls. Additionally, the distributed nature of the space supply chain introduces numerous potential attack vectors, from firmware tampering to insecure interfaces between contractors. Consequently, ensuring cybersecurity in this domain requires coordinated efforts between governmental agencies, industrial partners, and academia , combining technical innovation with policy alignment.

Historically, space missions were designed under the assumption that isolation from terrestrial networks provided inherent protection. However, the growing interconnection between ground and space segments and the adoption of commercial-off-the-shelf (COTS) components have expanded the attack surface. Threat actors can now target the entire mission lifecycle, from supply chain vulnerabilities to in-orbit operations, challenging traditional risk models.

Within this context, cybersecurity must be regarded as a mission enabler rather than an afterthought. Guaranteeing the confidentiality, integrity, and availability of space assets requires an integrated approach that spans hardware design, software development, communication protocols, and mission operations. This evolution is also reflected in standards such as the ESA Cybersecurity Framework and the ECSS policies, which aim to harmonize best practices across the European space ecosystem. In particular, the ECSS-E-ST-40C standard[3], which governs space software engineering, has undergone a critical revision in 2025, and it represents a necessary and fundamental strategic shift, elevating cybersecurity from an optional consideration to a mandatory, integrated engineering discipline within space system development. The standard's scope is comprehensive, applying to Space system product software developed across all elements of a space system, including the space, the launch service segment, and the ground segment. The software-specific cybersecurity mandates are enforced through the definition of some aspects, as follows:

- Security in requirement engineering: the standard mandates the rigorous capture of specific security requirements during the initial phase, ensuring they are explicitly defined.
- Security in Design and Architecture: Security analysis must directly influence architectural and design choices, through the Software Security Analysis Report, a document that must be used and revised during the definition of the detailed design. This ensures that all the identified security risks are definitively addressed by the chosen design architecture.
- Security in Production and Verification, by introducing software code verification activities implemented by Static Analysis tools, enhancing both code quality and security.

Moreover, ECSS-E-ST-80C [4] completes the picture by introducing security principles in space systems, defining concepts well known in the cybersecurity domain, such as attack surface reduction, defence in depth, least privilege, and need-to-know, and highlighting the most common threats affecting space missions, such as denial-of-service, jamming, and replay attacks.

Particular attention must be devoted to on-board software, which plays a decisive role in mission control and autonomy. Since in-orbit assets cannot be easily accessed or repaired, the ability to securely update and reconfigure software throughout the mission lifecycle is essential. Any unauthorized or corrupted update could compromise guidance, communications, or safety functions. For this reason, secure patching mechanisms have become a cornerstone of modern spaceborne cybersecurity, combining cryptographic robustness with operational reliability.

These risks demonstrate that threats to space systems can originate from multiple vectors (cyberspace, physical interference or supply chain), and this challenges the outdated notion that space systems are inherently safe due to isolation, thus the need for a holistic cybersecurity approach, targeting supply chain, ground infrastructure, communication links and on board systems.
That is why it is important to have robust software and hardware design processes that include security features. Manufacturers and companies should be aware of the long lifetime of spacecrafts and enhance flexibility to address cyber threats over the lifetime of the vehicle.

1.1 Scope

Recognizing the relevance of these issues, my thesis investigated the analysis, design and implementation aspects of a secure software patching mechanism for the Avum Orbital Module (AOM), subsystem of Space Rider, the ESA reusable space transportation system. This was possible by first studying the overall system architecture, both Hardware and Software, and then designing the necessary security requirements. The proposed solution was developed in alignment with the European Space Agency's efforts to have a stronger stance on cybersecurity matters. The result contributes to increasing the system's resilience and adaptability while preserving its safety and mission-critical reliability. Specifically, the secure software patching has been implemented with AES-GCM algorithm, ensuring confidentiality, integrity and authenticity. This is implemented maintaining compliance with ESA standards by customizing the Packet Utilization Standard Service 6 (PUS6), which is a standard that addresses the utilization of packets for the purpose of remote monitoring and control of the spacecraft. Ensuring the authenticity and integrity of patches is pivotal to mitigate risks associated with unauthorized or corrupted updates.

Additional activities included the validation of the cryptographic algorithm, a task of paramount importance given that the supplier had not conducted implementation testing on the specific target hardware. This step was crucial to verify both the correctness and robustness of the algorithm under actual operational conditions, ensuring its compliance with functional and security requirements. Furthermore, a codebase analysis was performed to identify and map potential vulnerabilities according to the MITRE Common Weakness Enumeration (CWE) framework. This activity provided a broader understanding of the system's security posture, highlighting areas where architectural or implementation-level weaknesses could emerge.

1.2 Space Rider mission

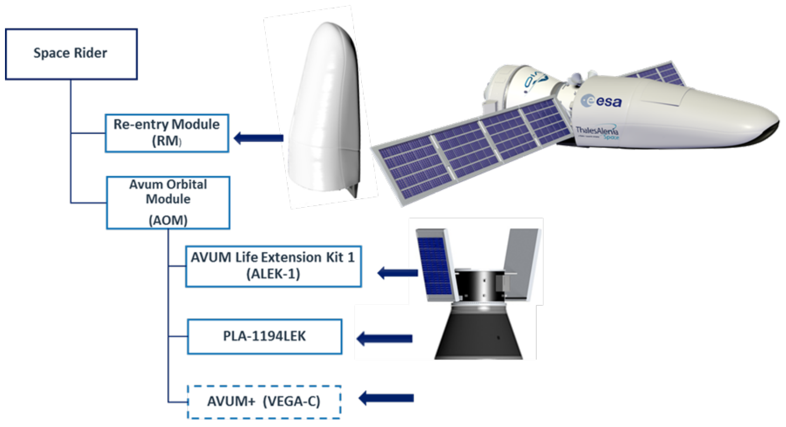


Figure 1.2: Space Rider components

The Space Rider program (Space Reusable Integrated Demonstrator for Europe Return) is developed by the European Space Agency (ESA). It is an uncrewed robotic laboratory which is planned to stay in low orbit for about two months. It "aims to provide Europe with an affordable, independent, reusable end-to-end integrated space transportation system for routine access and return from low orbit. It will transport payloads for an array of applications, orbit altitudes and inclinations"[5]. This program will allow for experiments in microgravity, in-orbit technology demonstration and validation for applications, and surveillance applications, such as Earth disaster monitoring. One of its main features is its ability to bring back experiments and other payloads from orbit, enabling the retrieval of data for detailed post-mission analysis. The purpose of the mission is also to give access to space to entities that are not necessarily directly involved in the aerospace sector, but wish to conduct experiments in such extreme conditions.

The Space Rider system is organized into two segments:

- Flight segment, composed of:
  - AOM (AVUM Orbital Module), responsible to interface with the VEGA-C Launcher, to accommodate the Solar Panels and to perform all the actuation maneuvers required during the orbital phase. It is physically made of the AVUM and of the AVUM Life Extension Kit (ALEK).
  - RM (Re-entry Module), which embarks the payload and ensures the re-entry and precision landing. It provides communication with ground stations for data download and commands uplink.
  - Payloads
- Ground segment, which is not under the Flight segment responsibilities, composed of:
  - VegaC Launch Complex
  - VCC (Vehicle Control Center), whose roles and responsibilities include supporting the Space Rider system for routine access to and return from low orbit, as well as ensuring the monitoring and control of the spacecraft throughout all mission phases, and the Payloads Ground Control Center.
  - Ground Stations Network
  - Operational Simulator
  - Landing Facilities

The purpose of the Ground Segment is to support the Space Rider System mission throughout the different phases of its lifecycle, from pre-launch to RM landing and AOM destructive re-entry. This component is particularly relevant within the scope of this thesis, as the patching mechanism originates in the Ground Segment, where the patch is ciphered and subsequently transmitted to the Flight Segment.

AOM, which is designed by AVIO, acts as a service module during the orbital phase, and performs functions like power supply, battery recharging, communication between AOM and RM, while RM, designed by Thales Alenia Space, is designed to be an operative and reusable space platform. The two modules are interconnected and exchange information, such as telecommands, which are essential for the correct operation of the system. This program is part of a broader strategy to strengthen Europe's independent access to space, and to foster innovation and technological advancement. Space Rider also opens up new possibilities for both scientific research and commercial exploitation in space, thus making it a high-value target for various adversaries.

The on board software controls navigation, communication and critical flight systems, so a compromised patch can jeopardize not only the mission but also the safety of any other co-orbital asset dependent on the spacecraft's operation, or, even worse, it could enable remote control, disrupt autonomous operations or introduce backdoors.

Chapter 2

System Overview

The Space Rider Avum Orbital Module (AOM) represents a complex, highly integrated system where hardware and software components coexist within a constrained and safety-critical environment. Understanding the overall architecture is essential to appreciate the challenges and design decisions behind the implementation of secure software patching and onboard cybersecurity functionalities. This chapter provides a comprehensive overview of the AOM system, describing both its hardware and software architecture, the hypervisor-based partitioning model, and the mechanisms that ensure operational safety, reliability, and data security. The discussion highlights how technical and organizational choices contribute to building a resilient space system capable of supporting long-duration missions in orbit.

2.1 Hardware

At the core of the AOM hardware architecture lies the On-Board Data Handling (OBDH) subsystem, which serves as the computational backbone of the spacecraft. It coordinates all the major activities of the platform, from telemetry and telecommand processing to payload management and system health monitoring. The OBDH integrates multiple functional units, including the On-Board Computer (OBC), mass memories, communication buses, and redundant interfaces that interconnect with other spacecraft modules.
The central processing element of the OBDH is the GR740 System-on-Chip (SoC), a fault-tolerant quad-core processor based on the LEON4 SPARC V8 architecture, specifically designed for space applications. This processor provides a balance between computational power and reliability, supporting deterministic execution for real-time tasks and featuring integrated interfaces such as SpaceWire, CAN, MIL-STD-1553, and UART. These interfaces ensure interoperability with various

spacecraft subsystems while adhering to European Cooperation for Space Standardization (ECSS) communication requirements.

In parallel with the processor, the system employs a Companion FPGA, which handles peripheral functions and custom logic for the management of low-level interfaces, timing synchronization, and redundancy control. The FPGA plays a crucial role in managing the communication between the AOM and the Re-entry Module (RM), ensuring robust data transfer and fault isolation in case of subsystem anomalies.
Given the harsh space environment, the hardware design also integrates a number of fault-tolerance mechanisms. Memory integrity is ensured through Error Detection and Correction (EDAC) techniques, which detect and correct any kind of error caused by cosmic radiation. Furthermore, critical components are implemented in redundant configurations, following a cross-strap redundancy principle that enables each subsystem to switch seamlessly to its backup channel in case of failure. This approach eliminates single points of failure and enhances system availability, which is crucial for long-duration missions. Additionally, the design includes non-volatile memories (NVM) dedicated to the storage of the cryptographic keys and application images, and MRAM memories, which support the memory scrubbing feature. While the Scrubber mechanism is used to preserve memory integrity in volatile components, it is not supported on NVM. The onboard computer's redundant architecture enables a patching mechanism where software updates are written to the redundant memory area while preserving the nominal one. This allows the system to restore the previous application image if the new patch fails, ensuring system availability and mission continuity.

This architectural redundancy, combined with radiation-hardened electronics and error-correction schemes, establishes the foundation upon which the onboard software and security layers operate.

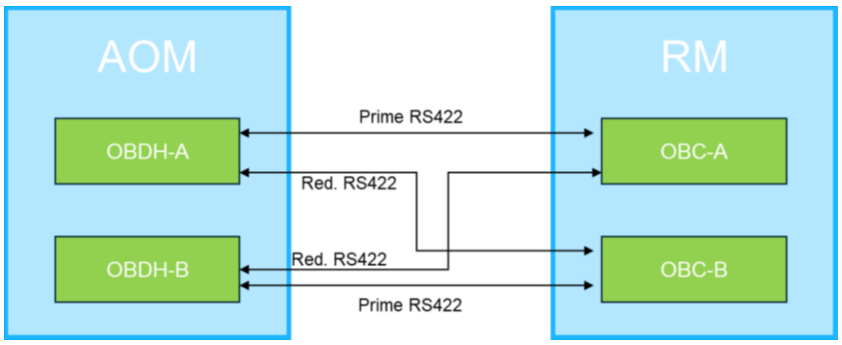


Figure 2.1: Cross-strap redundancy between AOM and RM subsystems

2.1.1 Security aspects

From a security perspective, the onboard computer is equipped with dedicated hardware mechanisms to ensure the protection of telecommands (TC) and telemetry (TM). Specifically, telecommands received from ground are decrypted, and telemetry data generated in flight are encrypted using hardware-implemented cryptographic algorithms, guaranteeing high reliability and performance. Specifically, this is executed by an FPGA provided by the GR740 SoC, ANACOND, which implements hardware decryption using AES-256.
To be clear, the software patching mechanism relies on a purely software-based encryption and decryption process, which operates on top of the already encrypted TC/TM channel. This results in a "double encryption", where hardware cryptography ensures secure communication, while software-level encryption protects the integrity and confidentiality of onboard software updates.

2.2 Software

The AOM on-board software (OBSW) is built upon a layered architecture that separates low-level hardware-dependent components from mission-specific applications. This modular approach increases maintainability.
The architecture is composed of three primary layers:

- Basic Layer, which includes the Real-Time Operating System (RTOS), device drivers, and board support packages. Its function is to provide a standardized software interface to the hardware, allowing upper layers to operate independently from specific processor or peripheral implementations. This layer ensures portability and abstraction from the hardware components. It includes the BootSW function, which resides in a dedicated PROM memory and is responsible for the initialization of hardware resources such as the processor and the memory (RAM). During the bootstrap sequence, the BootSW loads the Application Software (APSW) into memory and prepares the system for nominal operations. The Low Level I/O Device Drivers (Devices Interface) have the purpose of guaranteeing a Low level Software interface to manage I/O devices supported by the GR740.

- Service Layer, which provides shared services to higher-level applications. It acts as an intermediary between the basic system components and the mission-oriented functionalities.

- Application Layer, which implements mission functionalities such as telecommand processing, data handling, and payload operations. Additionally, this layer integrates the security functions responsible for patch authentication and decryption, ensuring software integrity and confidentiality.

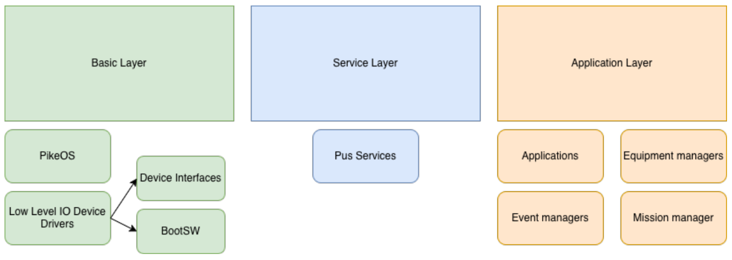


Figure 2.2: AOM Layered Architecture

This layered organization ensures clear separation of concerns between hardware control, system services, and mission-specific logic. A crucial aspect of the software architecture is its deterministic real-time behaviour. Tasks with different priorities must coexist without interfering with each other, and their execution order must be predictable even under fault conditions. This requirement justifies the adoption of a hypervisor-based system, described in the following section.

2.2.1 PikeOS

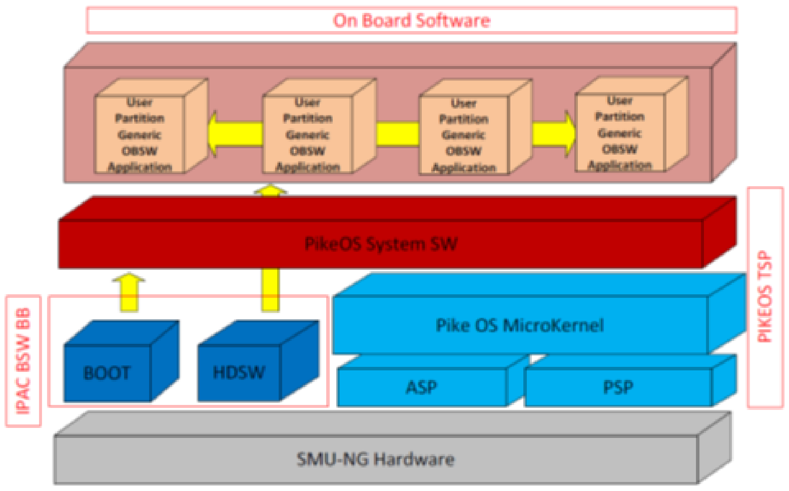

On Board Software
User Partition Generic OBSW Application
User Partition Generic OBSW Application
User Partition Generic OBSW Application
User Partition Generic OBSW Application
PikeOS System SW
IPAC BSW BB
BOOT
HDSW
Pike OS MicroKernel
ASP
PSP
PIKEOS TSP
SMU-NG Hardware

Figure 2.3: PikeOS Architecture

The main features that characterize PikeOS are Resource partitions and Time partitions:

- **Resource partitions** are one of the basic security mechanisms to support multiple virtual machines on top of PikeOS. They can be thought of as containers within which applications are executed. They define the system resources that their application can use and provide protection domains between different applications. It is important to note that an application running in a partition is completely unaware of applications in other partitions and of system resources to which it does not have access. The segregation of resources, achieved through mechanisms provided by the hypervisor and the IOMMU, enables a mixed-criticality approach. This allows software components with different criticality level (A/B/C/D), to execute safely on the same OBDH, while mantaining strict isolation and preventing any interference between partitions.

- **Time partitioning** is a mechanism for allocating CPU time amongst the partitions. It ensures that all the partitions get a predefined amount of execution time and to prevent any thread from starving others, even in the case of a faulting thread.

The power of PikeOS partitioning stems from the fact that time partitions and resource partitions are independent. Time partitioning is not just a case of allocating a certain amount of CPU time to each resource partition; multiple resource partitions can belong to the same time partition, while it is also possible in some circumstances that different threads from the same resource partition belong to different time partitions. It is the main concept at the base of safety and security critical applications.

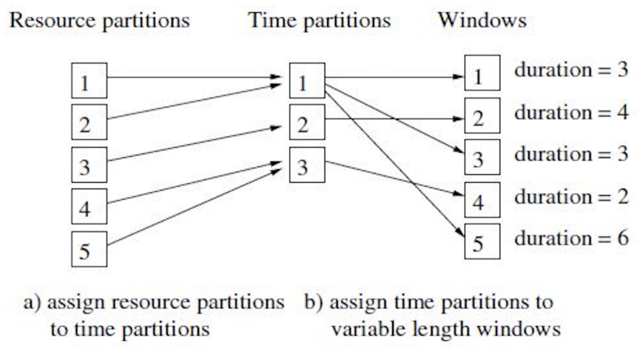


Figure 2.4: Example of resource and time partitioning interaction

This mechanism results in a scheduling that operates on two dimensions. Specifically, at the start of each time window, the scheduler activates the time partition assigned to that window. This time partition becomes the current time partition, and all the applications within the partition are eligible to be scheduled on the CPU. In the example 2.4, the system has five partition resources and three time partitions: this illustrates the many to one relationship between resource and time partitions.

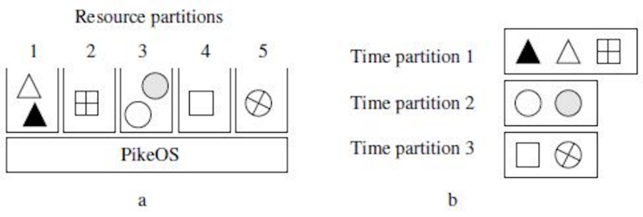


Figure 2.5: Assigning Resource Partitions to Time Partitions

Figure 2.5 shows how applications from the five resource partitions would be mapped into the three time partitions. Seen this way, a time partition is simply the set of applications from the resource partitions assigned to it.

In this system's case, to obtain the best degree of determinism possible, each partition has exactly one thread: the scheduler has the job to choose which thread to schedule based on the current time window.
PikeOS offers inter partition communication capabilities by implementing queuing ports and shared memory regions.
Specifically, there are two kinds of ports that enable interpartition communication:

- Queuing ports, which operate on a FIFO with a maximum message size.
- Sampling ports, which use a single message buffer that is atomically updated by a write operation. It also mantains the age of a message measured from the time of reception, which can be compared to the port's refresh rate. This allows to determine if a message is valid or not.

Another feature that is offered by PikeOS is the Health Monitoring (HM): it is a core mechanism designed for fault detection and deterministic recovery, crucial for maintaining system safety and resilience in mixed-criticality environments. HM operates across multiple abstraction layers, like user, partition, and module (global system), to ensure that detected errors, such as hardware exceptions, missed deadlines, or severe timing overruns, are handled according to a strict, pre-configured policy defined in the VMIT. A central safety principle enforced by HM is fault containment: errors originating from applications, particularly untrusted code, must be configured to trigger actions only within their local scope, typically resulting in a partition restart or termination. This is vital because configuring a local fault to execute a Module Level action (such as a full system reboot) would directly violate temporal and spatial partitioning integrity, potentially allowing a single non-critical fault to compromise the availability and safety state of the entire system.

PikeOS enables the effective segregation of software components based on their criticality. This classification is defined by the ECSS (European Cooperation for

Space Standardization) and assigns each software component a criticality category. The category corresponds to the severity of the most critical failure mode among all possible failure modes for that component.
From a security standpoint, PikeOS allows applications with different criticality levels to coexist on the same platform while maintaining strict isolation. This feature is particularly advantageous in space systems, where hardware resources are limited, and yet both safety-critical and non-critical tasks must share the same computational environment. Each partition is allocated its own resources and is assigned a criticality level as defined by the ECSS software classification:

- Category A: Software with the highest criticality. It implements functions where failure could result in catastrophic consequences for the mission.
- Category B: High criticality software where failure results in major mission impact but less severe than Category A.
- Category C: Medium criticality software where failure could cause minor mission impact or degradation of functionality.
- Category D: Low criticality software, such as operational procedures or non-critical functions where failure has minimal or no impact on mission success.

Naturally, the higher the criticality, the more rigorous the testing activities must be.

2.3 Security Functionalities

The security model implemented within PikeOS and the AOM software architecture is based on the principle of defense-in-depth, achieved through multiple layers of control.
Spatial and temporal separation ensures that applications are confined to their assigned partitions, preventing any form of data leakage or unauthorized interference. Communication between partitions occurs only through well-defined communication objects, such as shared memory regions, regulated by strict access control policies. This prevents unprivileged applications from modifying or corrupting configuration tables, which define the mapping of partitions, communication channels, and scheduling parameters.
The system's configuration files effectively act as a security policy, defining the boundaries of interaction between partitions and the privileges associated with each.

Security integrity is further maintained through rigorous configuration requirements defined in the Virtual Machine Initialization Table (VMIT), which is a crucial file for partitions configuration. This includes restricted resource allocation, such as the mandatory protection of physical memory and I/O access rights from untrusted domains, often requiring hardware mechanisms like IOMMU to constrain Direct Memory Access (DMA). Additionally, PikeOS provides configurable countermeasures against modern threats, including processor-level attacks exploiting speculative execution side channels (e.g. Spectre), which require careful integration and software development practices. The hypervisor thus serves as a high-assurance security domain responsible for managing and mediating all critical shared resources.

Additionally, PikeOS's security stance is centered on the principle of Robust Partitioning, which guarantees absolute spatial and temporal segregation between partitions at the platform level. It is very important to analytically and systematically prove this segregation by identifying and demonstrating coverage for all known potential Interference Channels possible. Interference Channels are a key concept for PikeOS because they relate to how applications in one partition could potentially compromise the robust partitioning of the hypervisor.

These design elements collectively enforce security mechanisms to reduce the likelihood of compromise even in the presence of malicious or faulty components.

2.4 Telemetry and Telecommand

The basic operation [6] of nearly any spacecraft relies heavily on continuous interaction with ground stations for control, command, communication, and data return. In many cases, this interaction is supported by a high degree of onboard autonomy. The Command and Data Handling (C&DH) system is responsible for managing all data sent to and received from the spacecraft.

A space link refers to the communication channel established either between a spacecraft and its corresponding ground system, or between two spacecraft. The primary data types transmitted over a space link are Telemetry (TM) and Telecommand (TC) data. A TC packet is the data unit that is used to carry a service request from an application process on the ground to an application process on-board. A TM packet is the data unit that is used to carry a service report from an application process on-board to an application process on the ground.

Together, the TM downlink and TC uplink form the essential communication interface between the spacecraft and ground operators.

On the uplink, the C&DH system receives and decodes all incoming commands and data, which are related to both platform and payload operations. These telecommands (TCs) are then routed to the appropriate subsystems or executed directly at the platform level. Typically, the C&DH system does not process payload-specific commands; instead, these are forwarded in encapsulated form directly to the payload. TC data can be broadly classified into:

- Direct spacecraft reconfiguration commands
- Application-specific commands

On the downlink, the C&DH system collects various types of data—either generated by platform subsystems or produced by scientific payloads—and multiplexes them into transfer frames for transmission to ground. Telemetry (TM) data can include:

- Spacecraft housekeeping (HK) data
- Orbital position data
- Scientific payload data
- Telecommand reception status (CLCW)
- Memory dump data

The Space Rider telecommanding architecture is based on a comprehensive set of standardized and layered command services. These services are functionally organized into three layers:

- Data Management Service
- Data Routing Service
- Channel Service

Within this architecture, the AOM OBC receives commands from the RM TC Data Routing Service. A telecommand (TC) packet is transmitted to the AOM OBDH system only after it has been completely and correctly received by the RM module. To support this routing mechanism, each TC packet is tagged with a MAP ID, which instructs the RM OBC to forward the packet to the AOM OBC.
The correct reception of each Transfer Frame by the Space Rider spacecraft is actively monitored. If a frame is rejected, retransmission mechanisms ensure that TC packets are delivered in sequence and without loss.
The TC Data Management Service standard defines how TC and telemetry (TM) packets are utilized for remote monitoring and control of spacecraft subsystems and payloads. In addition, the Packet Utilization Standard (PUS) complements these lower-layer standards by defining the application-level interface between ground systems and onboard software. Within the PUS framework, each application process is uniquely identified by an APID (Application Process ID), which specifies:

- the service user profile, determining which service requests the application is allowed to issue, and
- the service capability set, defining the services the application process is able to provide.

Once the MAP ID identifies the AOM OBDH as the destination of a TC packet, the APID is then used to determine which specific service is requested by or required from the AOM OBDH.
PUS defines various types of services that support spacecraft operations and management. Among all the services:

- Service 1, Request verification: it provides the capability for checking that a request received on-board has not been corrupted during transmission, or for checking the availability of the service that executes that request.
- Service 3, Housekeeping: it provides the visibility of any on-board parameter.
- Service 5, Event reporting: it provides the capability to report information of operational significance, such as on-board failures and anomalies, or initiation, progress and completion of activities.
- Service 6, Memory Management: it provides the capability for loading, dumping and checking the content of memory areas that exist on the spacecraft. This service is what has been modified to include and implement the secure software patching process.

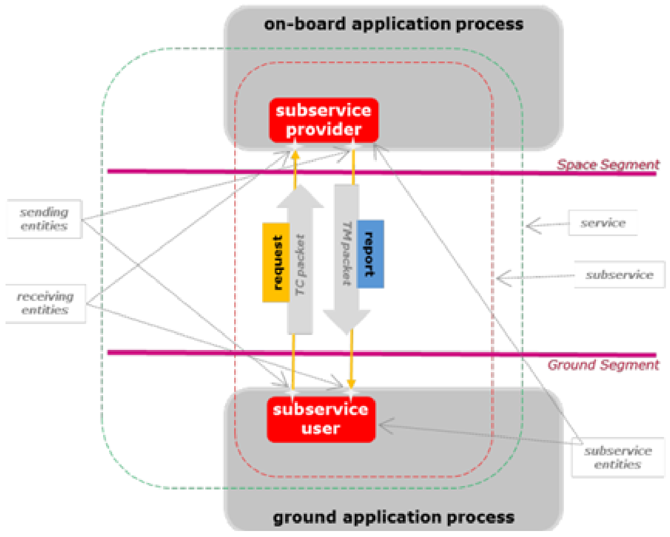


Figure 2.6: TC and TM exchange[7]

Chapter 3

Secure Software Patching

Ensuring the confidentiality, integrity, and authenticity of software patches in spaceborne systems is essential for both mission success and cybersecurity. Any alteration to onboard software has the potential to affect guidance, navigation, communication, and payload control subsystems. Therefore, secure patching mechanisms must guarantee that only authorized, untampered, and verified updates are accepted by the spacecraft, even under the most adverse operational conditions.
The secure patching mechanism for Space Rider is based on the AES-GCM algorithm, an Authenticated Encryption with Associated Data (AEAD) mode providing combined encryption and authentication in a single, efficient operation.
This design ensures end-to-end protection, addressing both confidentiality (protection against eavesdropping) and data authenticity (resilience against tampering or spoofing). This implementation complies with ESA standards by extending Packet Utilization Standard (PUS) Service 6, which is traditionally responsible for memory management operations and has also been used for managing patching operations. The integration of AES-GCM introduces a state-of-the-art cryptographic assurances, transforming a crucial maintenance function into a trustworthy and verifiable process.
The secure patching design leverages the WolfSSL cryptographic library, selected for its compact footprint and proven reliability in resource-constrained, safety-critical environments.
This combination of standards-based design and lightweight cryptography ensures suitability for the hardware constraints of the onboard computer. The patching mechanism operates within the AOM system and interacts closely with the OBSW (onboard software), the PikeOS hypervisor, and the underlying Hardware architecture. Therefore, its design had to respect not only cryptographic robustness, but also real-time constraints and fault-tolerance requirements inherent to spaceborne environments.

3.1 Secure Software Patch Functionality

The actual security feature is executed by an application running on the AOM OBC that decrypts and authenticates AOM on-board software patches encrypted and authenticated on ground before transmitting it to the spacecraft. This application is installed in a partition managed by the hypervisor. The ciphering function, implemented with AES-GCM authenticated encryption with associated data (AEAD), has the following security properties:

- Confidentiality: encrypted data will not leak any information about the sensible plaintext data without the correct decryption key.
- Integrity and authenticity: to detect accidental or intentional modifications of both the encrypted data and any additional authenticated data that is not encrypted.
- Authenticated Encryption with Associated Data: it guarantees both encryption of the plaintext and authentication of the ciphertext along with additional authenticated, but unencrypted, data.

In addition to these properties, the security mechanism also features an anti-replay mechanism by managing a patch counter register.

3.2 Key steps of the Patching mechanism

Patching is the only way to modify the software once the space vehicle has been launched and together with software maintenance must be given proper consideration during the whole software life cycle.
During the maintenance phase of the onboard software, the updated code shall be validated before deploying it to the operational environment. This is achieved by using simulation platforms, as they give developers visibility and control over the system behaviour: any state can be reached and any condition can be triggered. The patching process is about this phase too.
This is the general overview of the secure patching process:

1. Avio develops the onboard software patch in its protected software facility. Avio encrypts and authenticates the patch and authenticates the additional authenticated data. The AAD are composed of the key index, Initialization vector and Patch counter. The key index parameter identifies which of the available keys of 256 bits in the key database #1 is to be used.
2. Avio sends the encrypted patch and the authenticated associated data to the simulation facility.

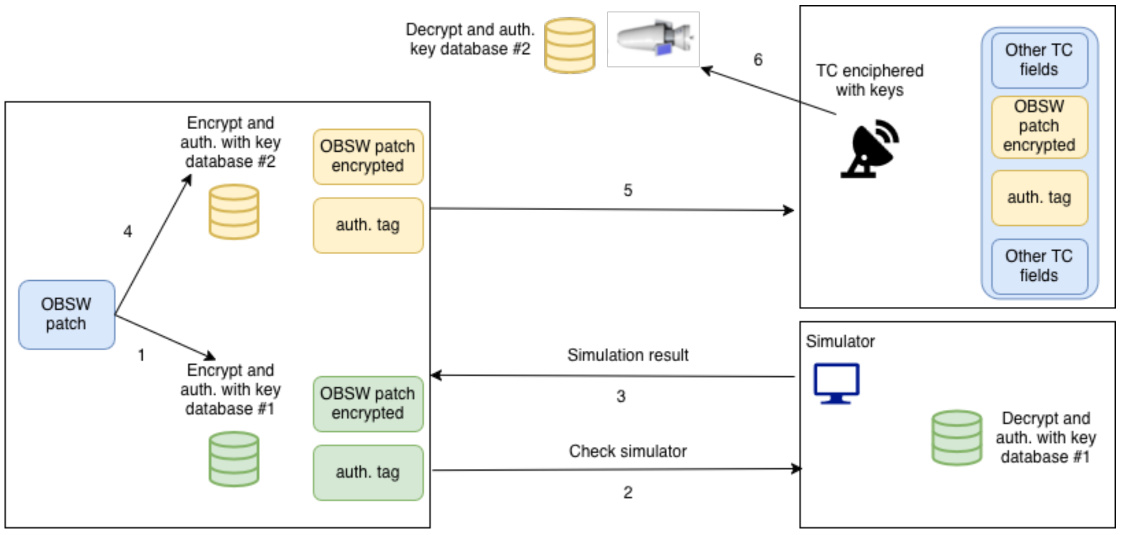


Figure 3.1: Patching mechanism context

3. The patch gets injected in the simulator, which contains the same pre-shared key database #1. The simulator is able to emulate the onboard software functionalities, included the secure patch mechanism. The simulated onboard software installs the patch only after successfully decrypting and authenticating it, authenticating the AAD and checking the validity of the patch counter. In case the simulations results are positive, the process can go on.

4. Avio encrypts and authenticates the patch already validated by the simulator and authenticates the associated data, which are made up of key index, Initialization Vector and Patch counter. This time around, the key index identifies the selected key from database #2 which is a database residing on the spacecraft.

5. Avio sends the encrypted patch and the additional authenticated data to the ground facility that will be in charge of transmitting it to Space Rider. This phase can include an additional layer of security, like encrypting the TC packets sent, but it is out of scope of Avio responsibilities.

6. The AOM OBSW decrypts and authenticates the patch with the key whose index is selected according to the received key index, from the pre-shared onboard keys database #2. The AOM OBSW (onboard software) accepts the patch if the patch and associated data authentication succeeds and install the patch if the counter value is valid.

3.3 Authenticated Encryption

The Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) is the de facto standard for Authenticated Encryption with Associated Data (AEAD). The primary goal of an AEAD algorithm is to combine the guarantee of three fundamental properties into a single process: Confidentiality, Integrity, and Authenticity. AES-GCM, specified in NIST Special Publication (SP) 800-38D[8], is the most frequently used symmetric block cipher mode of operation globally, employed in critical communication protocols such as TLS and IPsec. Its universal adoption and continued validity demonstrated over twenty years of cryptographic analysis confirm its robustness for intended uses.

3.3.1 Internal Architecture: CTR Mode, GHASH function and Initialization Vector

GCM is inherently a composite algorithm, consisting of two main functional components that work in parallel: the Counter Mode (CTR) for encryption and the universal hashing function GHASH for authentication.

- CTR Mode - Encryption: AES, operating in Counter Mode, generates a pseudo-random keystream by encrypting an incremental counter. This architecture is essential for GCM's efficiency, as the keystream generation does not depend on the plaintext blocks, thus allowing highly pipelined and parallelized implementations. Counter Mode is recognized as the most effective method for high-speed encryption.

- GHASH - Authentication: the authentication component uses the GHASH function, a universal hashing mechanism defined over a binary Galois field $GF(2^{128})$. This function sequentially processes both authenticated but unencrypted data (Associated Data) and the ciphertext (C). The output of GHASH is then combined via an XOR operation with the first keystream block encrypted by AES in CTR mode, finally producing the Authentication Tag. The structural separation between the CTR encryption pipeline and the GHASH authentication pipeline is the fundamental reason for GCM's exceptional throughput, as the two processes are independent except for the final tag encryption step.

- Initialization Vector: the Initialization Vector (IV), or Nonce, is a public parameter, typically 96 bits long in AES-GCM, whose function is not to maintain secrecy but to guarantee uniqueness for every encryption operation performed with the same secret key. GCM's security critically depends on the fact that a key is never reused with the same initialization vector. This uniqueness condition is an absolute and fundamental security requirement for the algorithm's integrity.

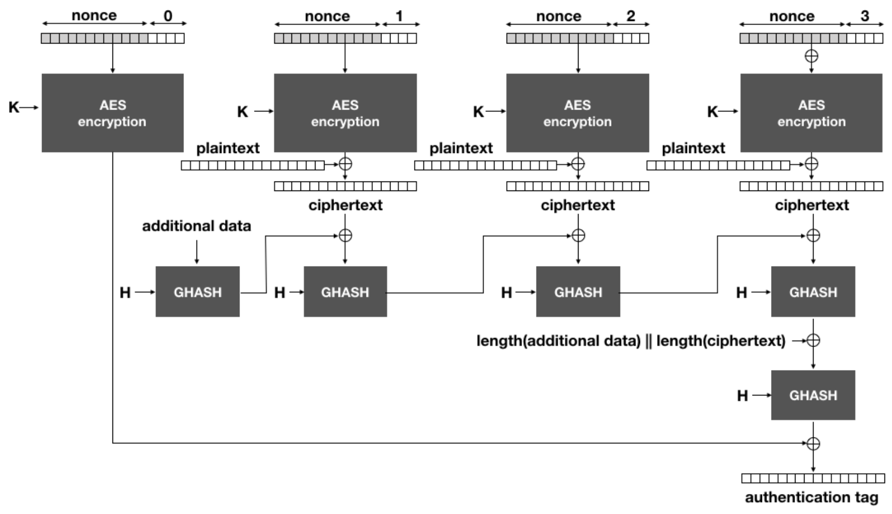


Figure 3.2: Algorithm internal mechanics

3.4 Cryptographic Design Decision: Adoption of AES-GCM

The choice of a symmetric-key algorithm like AES-GCM was driven by the possibility to share a set of keys out-of-bounds, at the launch site. This option[9] significantly reduces the complexity of the key management problem, completely removing all the advantages an asymmetric key algorithm could have had, since there is no longer a need for establishing a key over an insecure medium. Furthermore, symmetric cryptosystems are generally less complex and less computationally expensive. For all these reasons, symmetric key cryptography is the preferred solution for the classic point-to-point space mission communication security scenarios.

Furthermore, AES-GCM was the preferred choice because of several considerations:

- Performance: Galois/Counter Mode enables parallelizable encryption and authentication, minimizing latency, which is a crucial factor for real-time systems.

- Proven security: AES is endorsed by NIST and already widely adopted in space, aviation and defense applications, ensuring compatibility and trust.

- AEAD flexibility: By authenticating additional non-encrypted fields (AAD), such as patch counters or key indices, AES-GCM strengthens state awareness and contextual integrity during patch verification.

3.4.1 Intrinsic weaknesses

Despite its performance and community support, AES-GCM exhibits operational weaknesses that impose stringent requirements on state and key management. These weaknesses primarily stem from its vulnerability to operational error (misuse) and mathematical limits related to the reduced IV length. The most significant and serious weakness is its sensitivity to IV reuse. If the same IV is used to encrypt two different messages with the same key, the security failure is catastrophic:

- Loss of Confidentiality: a single IV reuse leads to the disclosure of the XOR operation between the two plaintexts. If an attacker obtains ciphertexts C_1 and C_2 (encrypted with the same Nonce IV) and knows or guesses one of the plaintexts (e.g., P_1), they can completely decrypt the other (P_2) via the relationship $P_2 = (C_1 \oplus C_2) \oplus P_1$.

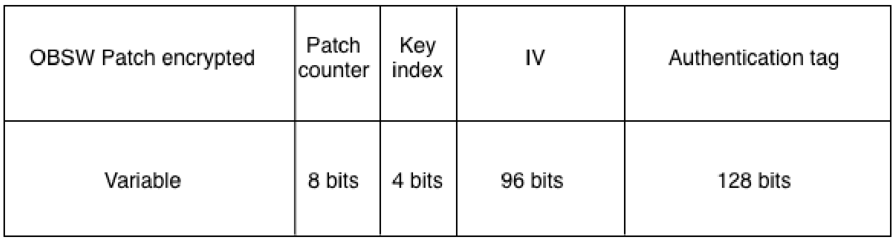
- Forgery Vulnerability: in addition to the loss of confidentiality, IV reuse compromises authenticity. The attacker can narrow the field of possible keys to a limited number of options and, through polynomial root-finding [10], generate valid forgeries with a high probability of success. This makes GCM's authenticity guarantee extremely fragile in case of operational error.

Operational limits

The standard IV size of 96 bits imposes rigid limits on the volume of data that can be encrypted with a single key, especially when using random IVs. Due to the birthday paradox, even though there are 2^{96} possible IV values, the probability of encountering a collision (i.e., reusing the same IV under the same key) increases much faster than intuition suggests. In cryptography, a collision probability below 2^{-32} is generally considered acceptable. Applying the birthday bound, this probability is reached after approximately 2^{32} random IVs have been generated, far fewer than 2^{96}, but still a very large number in practice. For this reason, NIST imposes an operational limit of about 2^{32} messages encrypted per key when using a random IV construction. In our context, this threshold is far above the expected operational usage, meaning that IV reuse due to random collisions is not a practical concern.

3.5 Crypto application

The application itself is implemented using the WolfCrypt library, a cryptographic software API that also provides FIPS validation. The keys used in this process consist of a set of pre-shared keys, injected through a dedicated interface on the on-board computer at the launch site by authorized employees, and stored in memory, which offers radiation-tolerant and Error Detection and Correction feature. The memory contains a set of cryptographic keys, each 256 bits in length, and each identified by a key index.
The picture 3.3 is the packet format expected from the OBSW.



OBSW Patch encrypted	Patch counter	Key index	IV	Authentication tag
Variable	8 bits	4 bits	96 bits	128 bits

Figure 3.3: Packet format

Patch counter, Key index and Authentication tag constitute the Additional Authenticated Data (AAD), which is additional data sent in clear that will get integrity-checked by the AES-GCM algorithm. The key index is the index which identifies the key stored in the pre-shared keys database recorded in memory, and each communicating entity shares the same keys database. The patch counter value must be bigger than the value stored in memory, and in this case the patch can be applied and the counter register is updated with the new value, otherwise the patch is rejected. This mechanism avoids replay attacks, which occur when an attacker eavesdrops on a secure communication, intercepts it, and then fraudulently delays or resends it to misdirect the receiver into doing what he wants. Authentication and Integrity check is provided by the authentication tag, which is managed by the AES-GCM algorithm.
If either the patch payload or the additional authenticated data are not correctly authenticated, the packet will be discarded, guaranteeing data authenticity by verifying that the ciphertext and associated data were produced by an entity holding the shared secret key (in this case, the legit ground segment).

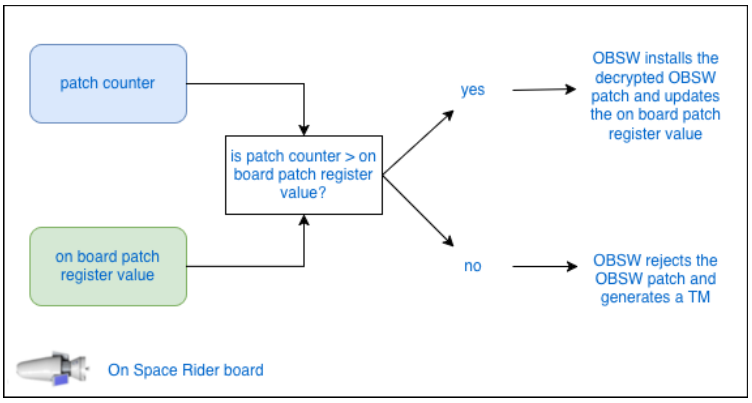

patch counter
is patch counter > on board patch register value?
on board patch register value
yes
no
OBSW installs the decrypted OBSW patch and updates the on board patch register value
OBSW rejects the OBSW patch and generates a TM
On Space Rider board

Figure 3.4: Anti-Replay mechanism

3.6 Strategic importance

The strategic value of securing Space Rider's software patching mechanism extends beyond the mission itself. Space Rider represents a cornerstone of European autonomy in space operations, offering reusable, cost-efficient access to Low Earth Orbit. Any compromise to its onboard software could thus impact not only a single mission, but also Europe's strategic independence, scientific credibility, and commercial competitiveness. Cyberattacks targeting the patching pipeline could result in mission denial, loss of control, or the insertion of a persistent backdoor into flight-critical systems. Given the increasing militarization and commercialization of space, threat actors may include nation-states seeking espionage, criminal organizations pursuing ransom or sabotage, or insiders with privileged access. The risk surface is therefore broad and evolving. By securing the patching mechanism through end-to-end encryption, authentication, and replay protection, the system effectively closes one of the most critical attack vectors in post-deployment software management. This aligns with the ESA Cybersecurity Framework, which emphasizes continuous assurance and defense-in-depth across the mission lifecycle.

3.6.1 Possible attack targets

Below are several possible attack scenarios for introducing malware into a spacecraft's on board software patch during the distribution phase.

- Infrastructure compromise
  - Remote exploitation of vulnerabilities: attackers can exploit vulnerabilities in the ground upload servers or software to gain unauthorized access.

  - Credential theft: if attackers steal or compromise the credentials used to access the ground upload systems and impersonate authorized personnel. This enables them to modify the patch or substitute it with a malicious version.
  - Insider threat: an insider with legitimate access to the ground systems could intentionally modify the patch or bypass the integrity checks.
  - Key injection risks: keys should be injected at the launch site by authorized staff via software uploaded into NVM (non-volatile memory). The main risk is malicious or accidental upload of tampered/unverified software.

- Development intrusion
  - Attacker infiltration: an attacker infiltrates the developer environment or source code repositories. By injecting malicious code early in the software build process, the attacker ensures that the malware becomes part of the patch before it's even signed.
  - Malicious or coerced personnel: an individual with privileged access may intentionally insert malware into the patch.

Each of these scenarios underscores the necessity for a comprehensive, multi-layered security approach spanning the entire software patch lifecycle, from initial development to final distribution. For this reason, end-to-end encryption for patch transmission has been implemented.

The use of WolfSSL's WolfCrypt library is motivated by its lightweight design and

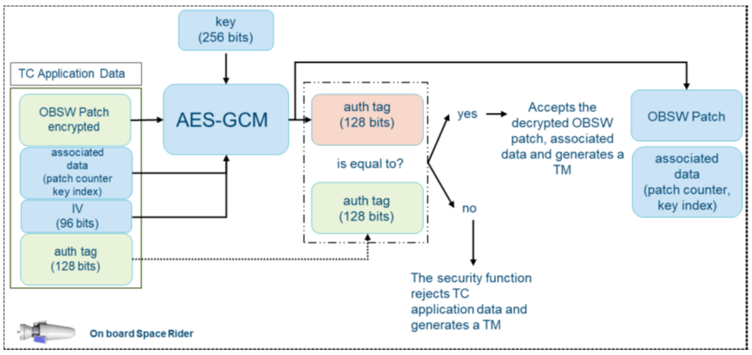

key (256 bits)
TC Application Data
OBSW Patch encrypted
associated data (patch counter key index)
IV (96 bits)
auth tag (128 bits)
AES-GCM
auth tag (128 bits)
is equal to?
auth tag (128 bits)
yes
Accepts the decrypted OBSW patch, associated data and generates a TM
no
The security function rejects TC application data and generates a TM
OBSW Patch
associated data (patch counter, key index)
On board Space Rider

Figure 3.5: Decryption and Authentication decision process

high level of optimization for embedded systems. Its AES-GCM implementation offers the following advantages:

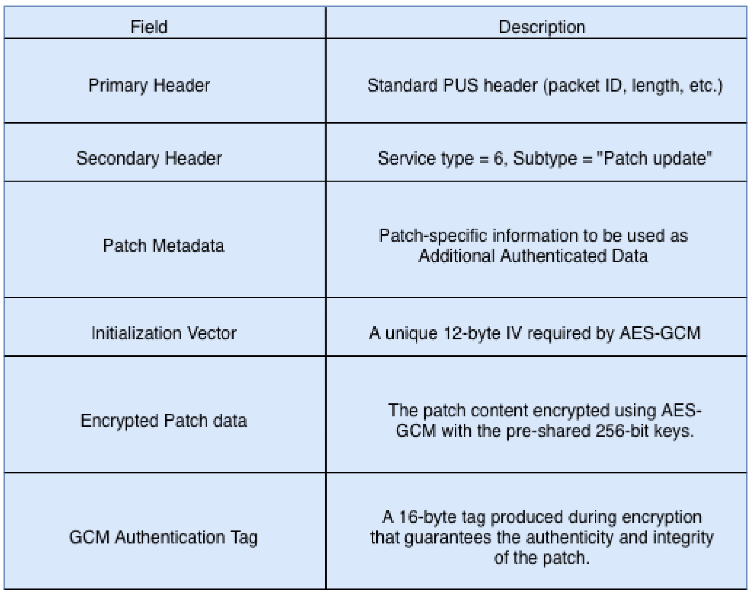
- Low memory footprint, making it suitable for spaceborne hardware with limited resources.

- Compliance with FIPS 140-2 and other relevant standards for use in safety- and mission-critical systems.

3.7 Modified PUS 6 packet structure

The PUS Service 6 is responsible for memory management in a spacecraft's onboard software system. This service plays a crucial role in handling memory related operations, including patching, modification, and verification of software running on the spacecraft's onboard computer. The original PUS Service 6 telecommand includes:

- Primary header, containing the packet ID, length, etc.
- Secondary header, Indicates the service type and subtype. This is used to discriminate between different memory management operations such as memory load or software update.
- Payload



Field	Description
Primary Header	Standard PUS header (packet ID, length, etc.)
Secondary Header	Service type = 6, Subtype = "Patch update"
Patch Metadata	Patch-specific information to be used as Additional Authenticated Data
Initialization Vector	A unique 12-byte IV required by AES-GCM
Encrypted Patch data	The patch content encrypted using AES-GCM with the pre-shared 256-bit keys.
GCM Authentication Tag	A 16-byte tag produced during encryption that guarantees the authenticity and integrity of the patch.

Figure 3.6: Modified PUS Service 6 packet structure

The updated Service 6 packet layout was designed to introduce fields that would be useful for the decryption: patch metadata, initialization vector, authentication tag and the encrypted payload.
Due to PUS Service 6 design constraints, a Telecommand (TC) cannot carry a

payload larger than 986 bytes. Therefore, whenever the encrypted payload exceeds this size, the ground segment must segment it into multiple TCs.

Although the PUS standard defines Service 13 (Large Packet Transfer) to handle the reliable transmission of large packets, including retransmission of individual missing segments,this service is not supported by AOM. As a result, Service-13-style redirection and retransmission mechanisms are unavailable. The solution adopted is to transfer the retransmission responsibility to the ground segment: for each transmitted TC, identified by a sequence number, the spacecraft sends a TM acknowledging whether it was correctly received. The ground segment then identifies the missing TCs and retransmits them selectively. This ensures reliable transfer of large payloads without requiring Service 13.

Additionally, a custom extension of Service 6 was introduced (outside of the PUS standard). This dedicated command explicitly informs the OBSW when the transmission of the entire segmented payload is complete, enabling the decryption process to start safely. Once this command is received, the OBSW stores the full payload in non-volatile memory (NVM) and triggers decryption. This process is handled by a high-priority partition, MODE, which delegates the decryption activity to a lower-priority partition, FLASH. The FLASH partition runs in background mode, which means it is scheduled only when other applications within its time partition are idle. Upon successful decryption, the task of applying (flashing) the patch is transferred back to MODE partition. This design results in a significantly

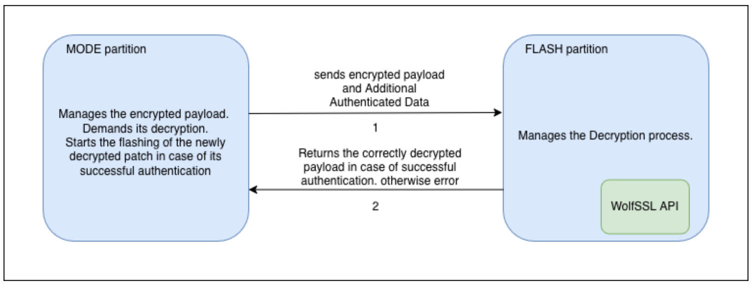

MODE partition
Manages the encrypted payload. Demands its decryption. Starts the flashing of the newly decrypted patch in case of its successful authentication
sends encrypted payload and Additional Authenticated Data
1
Returns the correctly decrypted payload in case of successful authentication. otherwise error
2
FLASH partition
Manages the Decryption process.
WolfSSL API

Figure 3.7: High level view: interaction between MODE and FLASH partition

more efficient use of CPU cycles, by offloading computationally intensive tasks (such as decryption) to idle time, while ensuring that critical operations (such as flashing) are handled promptly by higher-priority components. This architecture provides several advantages in terms of real-time scheduling and fault tolerance.

From a real-time scheduling perspective, offloading the decryption process to the low-priority FLASH partition allows the system to perform computationally expensive operations without affecting the responsiveness of time-critical tasks. Since the FLASH partition is scheduled only when higher-priority applications are idle, this approach ensures optimal CPU utilization without interfering with

mission-critical software. The actual flashing of the patch, which is time-sensitive and potentially hazardous if interrupted, is delegated to a high-priority partition, guaranteeing its execution within a predictable and bounded time frame.
In terms of fault tolerance, the classification of the FLASH partition as Category C reflects a design philosophy where potential failures in the decryption process do not lead to catastrophic consequences. By isolating the patch processing logic within this non-critical partition, the system limits the impact of potential software errors or transient faults. Furthermore, by separating the decryption and flashing stages into distinct partitions with different priority levels and safety classifications, the overall robustness of the patching mechanism is enhanced. This separation of concerns reduces the risk of a single point of failure and contributes to system reliability, especially in spaceborne environments where recovery options are limited.

Chapter 4

Implementation

This chapter describes the practical implementation of the secure software patching mechanism introduced in the previous chapter. The work focused on the development of the software module responsible for patch authentication and decryption within the AOM OBSW. The implementation required the creation of a dedicated PikeOS partition, the configuration of shared memory for inter-partition communication, and its integration with the existing OBSW framework.

4.1 Software Requirements Specification

In space software engineering, system quality and mission reliability are directly dependent on the quality and rigor of the requirements that drive the development process. Without clear, verifiable, and traceable requirements, it becomes impossible to ensure correctness of the final product or to demonstrate that it satisfies mission objectives. This principle is particularly critical when cybersecurity is involved: security cannot be added retroactively once the architecture has been defined. Instead, it must be specified, engineered, and verified from the very beginning of the lifecycle. The introduction of cybersecurity mandates in ECSS-E-ST-40C [3] institutionalizes this paradigm by elevating security to a first-class engineering requirement, to be handled with the same level of rigor as functional performance, reliability, and safety.

In this context, the detailed specification of the secure patching functionality was defined through a formal Software Requirements Specification (SRS) document, following ESA and Avio standards. The requirements were written in compliance with ECSS-E-ST-40C [3], ensuring full bidirectional traceability across system design, implementation, verification, and validation.

Each requirement is uniquely identified through a structured ID naming convention, enabling traceability to higher-level system and mission requirements. The SRS

defines the functional behavior of the secure patching module integrated within the AOM OBSW, including operational constraints, error handling mechanisms, and cybersecurity safeguards.
The specification addresses four primary functional domains:

- **Cryptographic configuration and key management**
- **Patch authentication and decryption**
- **Counter validation and replay protection**
- **Memory access control and operational modes**

These requirements collectively ensure that the patching mechanism operates securely, deterministically, and in compliance with the cybersecurity provisions established by ECSS and ESA for space-grade onboard software.

4.1.1 Cryptographic configuration and key management

The decryption function relies on a pre-initialized portion of MRAM whose purpose is to contain configuration parameters, such as the Patch counter value and the 256-bit symmetric keys, each referenced by a unique index. The OBSW shall select the proper key according to the index received within the incoming telecommand. The key management requirements specify that:

- Keys are stored in a radiation-tolerant memory region protected by Error Detection and Correction mechanism: in order to protect configuration data, the Memory Controller provides an EDAC function capable to correct up to 2 wrong 4-bits Symbols in the same 32-bits Codeword.
- The memory region is read-only for the application layer to prevent tampering. In fact, this is implemented by a Hardware interlock, requiring an explicit write-enable command before accessing the memory.
- if the EDAC system detects uncorrectable errors, the OBSW shall attempt recovery from the redundant memory.

If recovery fails, the patching process is aborted and a Telemetry (TM) event shall be sent to notify ground segment.
These mechanisms ensure both key integrity and system resilience in the event of radiation-induced data corruption.

4.1.2 Patch Authentication and Decryption

Upon reception of a patch Telecommand (TC), the OBSW extracts three essential parameters:

- the Key Index,
- the Initialization Vector,
- the Patch Counter value.

These parameters are used during the AES-GCM decryption process. In particular, the Key Index and the Patch Counter value must be included as Additional Authenticated Data (AAD), to ensure their integrity and protect them against tampering.
The AES-GCM decryption and authentication process is performed using WolfSSL's WolfCrypt library, which provides an implementation optimized for embedded systems. The adoption of WolfSSL's WolfCrypt library should be regarded as an implementation hypothesis. The development work described in this thesis has been performed using WolfSSL's APIs.
The module ensures that:

- AES-GCM is used as the only supported decryption scheme.
- both ciphertext and AAD are verified for authenticity before patch installation.

If authentication succeeds, the patch payload is stored in a Non-volatile Memory (NVM) for further installation. In case of failure, the patching sequence is immediately terminated, and a Telemetry (TM) event report is generated and issued.
This mechanism guarantees end-to-end integrity and prevents unauthorized or corrupted patches from being installed.

4.1.3 Counter validation and Replay Protection

The SRS also specifies a robust anti-replay system based on an increasing patch counter stored in memory.
The OBSW compares the counter received with the patch against the stored value:

- If the new counter value is greater, the patch is accepted and processed.
- If the new counter value is equal to or lower, the patch is rejected and the event is reported to ground via a TM report.

After successful patch installation, the onboard counter register is updated to match the latest value received from ground.This simple but effective mechanism ensures chronological integrity, preventing rollback or downgrade attacks, and maintaining synchronization between ground and onboard states.

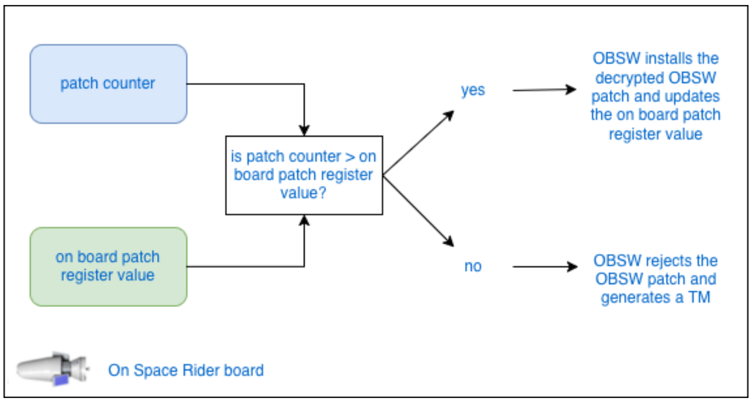

patch counter
on board patch register value
is patch counter > on board patch register value?
yes
no
OBSW installs the decrypted OBSW patch and updates the on board patch register value
OBSW rejects the OBSW patch and generates a TM
On Space Rider board

Figure 4.1: Anti-Replay mechanism

4.1.4 Memory Access Control and Operational Constraints

In accordance with safety and mission assurance requirements, the secure patching functionality can operate only under specific conditions and modes. This restriction prevents accidental or malicious modification of software during critical mission phases.
Additionally, the OBSW shall also:

- Inhibit memory load or dump commands targeting critical memory areas.
- Notify the ground segment of any attempt to access or modify such restricted memory regions.
- Select the appropriate NVM Flash bank (nominal or redundant).

These requirements implement defense-in-depth at both software and system levels, ensuring that even if the cryptographic layer is secure, operational policies still enforce safety and integrity during patch installation.

4.1.5 Error Handling and Fault Recovery

The SRS includes explicit requirements for fault detection, correction and reporting. In case of authentication or decryption failure, the module shall:

- Abort the patching operation,
- Generate a TM message to notify the ground,
- Log the failure cause (counter error, authentication or decryption failure, EDAC correction error).

If the EDAC mechanism cannot correct memory faults, the OBSW shall attempt decryption using the redundant memory. Persistent failures result in termination of the process and transmission of a diagnostic TM.

4.1.6 Summary of Key Requirements

The formal requirements define a tightly controlled and secure patching workflow where:

- Key management, decryption and validation are clearly separated.
- All cryptographic operations are deterministic and traceable.
- Error conditions are explicitly handled and reported.
- Operational safety is enforced by limiting patch execution to predefined mission modes.

4.2 Partition Design, Configuration and Communication

The implementation of the secure patching functionality required the creation of a dedicated PikeOS partition within the AOM On-Board Software (OBSW). This new partition, named FLASH, was designed to host all cryptographic operations related to patch decryption and authentication, ensuring full isolation from mission-critical tasks and adherence to real-time and cybersecurity constraints.
The integration process followed the PikeOS system definition workflow, which is managed by the Virtual Machine Initialization Table (VMIT). The VMIT represents the core configuration file that defines the system structure, including:

- the list of partitions (e.g., MODE, FLASH),
- each partition's scheduling and priority,
- memory requirements and allowed memory regions,
- communication channels (message queues, sampling ports),
- file access permissions and shared memory mappings.

In other words, the VMIT acts as the single source of truth for system configuration. During system startup, PikeOS reads the VMIT and deterministically instantiates the partitions, allocates memory segments, and configures inter-partition communication. This guarantees deterministic initialization, controlled data flow, and strict enforcement of security boundaries.

```xml
<Partition Name="FLASH" Identifier="19" >
    <MemoryRequirementTable>
        <MemoryRequirement Name="RAM" Type="VM_MEM_TYPE_RAM" Size=
    "xxxxxxx"/>
    </MemoryRequirementTable>
    <FileAccessTable>
        <FileAccess FileName="shm:MODE" AccessMode="VM_O_MAP
    VM_O_RD"/>
        <FileAccess FileName="shm:FLASH" AccessMode="VM_O_MAP
    VM_O_RD VM_O_WR"/>
    </FileAccessTable>
    <QueuingPortTable>
        <QueuingPort Name="recv_queue_from_PUS_to_FLASH" Type="
    VM_PORT_QUEUING" Direction="VM_PORT_DESTINATION" MaxMessageSize
    ="xxxx" />
        <QueuingPort Name="snd_queue_from_FLASH_to_PUS" Type="
    VM_PORT_QUEUING" Direction="VM_PORT_SOURCE" MaxMessageSize="
    xxxx" />
    </QueuingPortTable>
</Partition>
```

Listing 4.1: VMIT structure: definition of the FLASH partition

The XML code snippet 4.1 is extracted from the project's VMIT. In the configuration interface, each partition is associated with a dedicated *Partition* tab, where its parameters and interactions with other partitions can be defined and controlled.

4.2.1 Partition Design and Configuration

The creation of the FLASH partition aimed to provide an isolated execution environment dedicated to the secure patching process. This separation ensures that any malfunction or fault in the decryption process remains confined, without affecting higher-criticality tasks such as attitude control or telemetry management. The code that will reside in this partition is classified as Category C, following ECSS-E-ST-40C[3], and the partition is configured as a low-priority entity within the PikeOS scheduling plan. Within the VMIT, the partition is declared as a virtual machine instance, and its parameters include:

- Memory segments for code, stack, and data, statically allocated to guarantee predictability.
- Definition of input and output shared memory regions, specifying the direction of communication (source and destination partitions).
- Access control policies ensuring strict isolation between partitions.

The VMIT thus acts as the authoritative source defining the spatial and temporal boundaries of the partition, serving as a single configuration point from which the PikeOS system image is generated.

4.2.2 Inter-Partition Communication Design

The FLASH partition primarily communicates with MODE partition, as it is the high priority partition chosen to install the patch once the authentication and decryption phase are successfully over.
This interaction is established through shared memory regions, which is defined in the VMIT. Each communication object is explicitly declared with its ownership, access rights, and direction, ensuring that no implicit or unauthorized data exchange occurs between partitions.
The VMIT configuration specifies the following properties for each shared region:

- Virtual memory address and size allocation.
- Source and destination partitions.
- Access rights (read/write).

As seen in 4.1, the shared memories, which are used to exchange data between the two partitions FLASH and MODE, have different direction of communication: for the FLASH partition, the shared memory `shm:MODE` is read only, while the shared memory `shm:FLASH` has write enabled. Moreover, FLASH partition cannot access any other shared memory, enforcing isolation policies offered by PikeOS.

Within its configuration, queueing ports are also defined for FLASH, establishing a communication channel with the PUS partition. Queuing ports provide an asynchronous communication mechanism, where messages sent to a port are stored in an internal FIFO queue until the receiving process retrieves them. Only the partition owning the port can invoke send/receive operations on it. When a message is sent, it is placed into the FIFO buffer associated with the source port; the separation kernel later transfers that message to the buffer of the destination port according to the channel configuration. PikeOS, being fully compliant with ARINC-653 (Avionics Application Software Standard Interface), defines a system event that moves messages from source to destination ports independently of partition execution. This isolates communication from scheduling and avoids interference between partitions.

Possible Security shortcomings of queuing ports

The paper[11] demonstrates that the original ARINC-653 specification contains hidden covert channels, meaning that the state of a queuing buffer (full/not full)

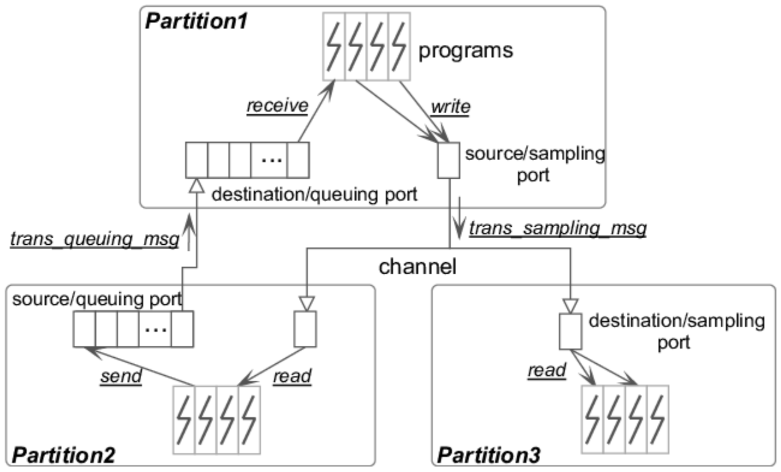


Figure 4.2: Queuing port based Communication[11]

could be exploited to leak information between partitions. This violates the requirement that partitions must not influence one another outside the predefined communication policy.
To eliminate this covert channel, the suggestion is to modify the queuing port behavior so that when a buffer is full, messages are silently discarded, instead of returning an error to the sender.

4.2.3 Execution Flow

The secure patching execution flow, starting from the reception of the telecommand (TC), involves multiple PikeOS partitions, each responsible for a different stage of the operation:

- **FLASH**: executes the decryption and authentication of the received patch and handles any cryptographic error conditions.
- **MODE**: interfaces with the memory areas where cryptographic keys and application software reside and manages the storage of the incoming encrypted APSW chunks.
- **RM**: receives telecommands and sends telemetry messages. It interfaces directly with RM subsystem, which exchanges TCs and TMs with the ground segment.
- **PUS**: validates and interprets the TCs received from RM and generates TMs according to the ECSS-PUS standard.

The complete interaction chain shown in Figure 4.3 proceeds as follows:

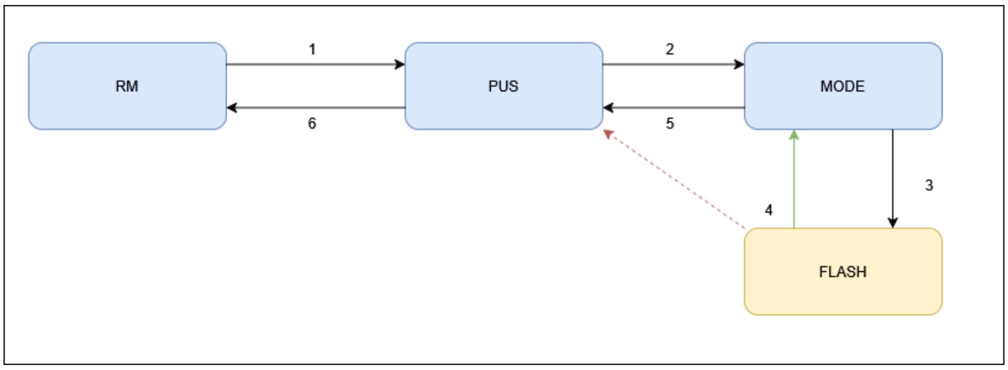


Figure 4.3: Execution flow

1. RM receives the TC (Service 6) from the ground segment and forwards it to PUS.

2. PUS validates the TC format and parameters. It immediately generates a TM(1,1) or TM(1,2) to notify RM (and consequently the ground segment) whether the TC is valid or rejected. If the TC is accepted, PUS forwards it to MODE. MODE then begins collecting the APSW encrypted chunks.

3. Once the final Service 6 TC is received, indicating that the full encrypted APSW image has been transferred, MODE notifies FLASH that the encrypted payload is available in the shared memory region.

4. FLASH performs the decryption and authentication of the received payload. If an error occurs (e.g., failed authentication, invalid parameters), FLASH requests PUS to generate a TM(5,2) containing the rejection reason. Otherwise, upon successful decryption, FLASH requests PUS to generate a TM(5,1), meaning authentication and decryption successful, and writes the decrypted image into the shared memory `shm:FLASH`.

5. MODE retrieves the decrypted APSW image and sends a request to PUS to generate TM(1,8), notifying the ground segment of the successful completion of the telecommand.

6. PUS generates and forwards the TMs.

4.2.4 FLASH partition

Once the `FLASH` partition has been correctly configured, as described in Chapter 4.2, the next step consists in implementing the software components, in particular the integration of wolfSSL's *WolfCrypt* module. The integration required adapting the

WolfCrypt library to include only the functionalities relevant to this project, namely the AES-GCM cryptographic routines. For this purpose, both the `Makefile` and the include directives were modified so that only the necessary source files were compiled. WolfCrypt offers a wide range of cryptographic algorithms, but in a safety and security-critical environment such as this one, it is essential to minimize the codebase and remove all unused features in order to reduce complexity, attack surface, and resource usage.
After this process, the development environment was fully set up and ready for implementation. The following sections illustrate the cryptographic functions that have been used.

```c
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Listing 4.2: Initialization Function

The function **`wc_AesInit()`** initializes the AES context structure used by WolfCrypt. It prepares the internal state of the AES object and allows the caller to optionally specify a memory heap and device identifier to use with crypto callbacks or async hardware. If the initialization fails, the returned error code is propagated to ensure that no cryptographic operation is performed on an uninitialized context. In this case, since only static memory use is allowed, the memory heap is not defined, and so is the device ID: the whole functionality is implemented in software.

```c
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

Listing 4.3: Key management

The function **`wc_AesGcmSetKey()`** loads the AES-256 key into the AES context. In this case, the key is a pre-shared symmetric key stored onboard and used to decrypt the incoming APSW ciphered image. By separating key loading from context initialization, WolfCrypt allows flexibility in key management and enforces explicit key assignment before any encryption or decryption attempt. This function returns an error code in case of a wrong size key assignment.

```c
int wc_AesGcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

Listing 4.4: Decrypt function

Finally, the function **`wc_AesGcmDecrypt()`** performs the authenticated decryption using AES-GCM. It takes as input the ciphertext, the Initialization Vector (IV), the authentication tag, and Additional Authenticated Data (AAD), which contains the Patch Counter value and the key index. During execution, the function verifies the authenticity and integrity of both the ciphertext and the metadata (AAD). If authentication fails, it means that the packet, Additional data, Initialization Vector or tag has been tampered, returning an error code, and no plaintext is produced, ensuring that corrupted or malicious patches are never applied.

```
int wc_AesFree(
    Aes * aes
)
```

Listing 4.5: Free function

In the end, the function **`wc_AesFree`** releases the AES context previously initialized with **`wc_AesInit()`**. It performs all necessary cleanup operations on the internal data structures used by the AES object, such as clearing sensitive material from memory and releasing any resources allocated during the cryptographic operation. Calling this function ensures that no residual key material or intermediate state remains in memory after the decryption procedure completes, contributing to both memory safety and secure key handling.
This is true on HOST environment, but not on the real target hardware: here resource allocation is strictly controlled and the wolfCrypt library is configured to avoid dynamic allocation, hence this method results in a no-op. Moreover, the standard math library is used on the HOST environment, whereas on the real hardware the system relies on *libmcs*[12], a mathematical library provided by ESA. libmcs is designed for embedded and safety-critical systems and is pre-qualified for ECSS software criticality category B.

4.3 Cryptographic Validation and NIST KAT Implementation

The operational environment of the Spacecraft On-Board Computer (OBC), characterized by non-standard, radiation-tolerant hardware and the use of a Real-Time Operating System like PikeOS, imposes more stringent cryptographic validation requirements compared to implementations on COTS (Commercial Off-The-Shelf) platforms. Validation was not limited to the WolfCrypt library but extended the tests to the integration on the target hardware to ensure that the execution of the AES-GCM algorithm produced correct and compliant results under all operational conditions.

4.3.1 The Importance of Validation on Non-Standard Hardware

Flight hardware is optimized for resilience and low power, not for general-purpose performance, and often features vendor-specific architectures for the processor (e.g., SPARC LEON architecture) and memory modules. In this context:

- Architectural Differences: The implementation of cryptographic algorithms, though standard at the source code level, can be affected by specific compiler optimizations for the architecture, or by potential hardware errors that could deviate execution.
- Memory Management: Memory restrictions imposed by PikeOS and the use of protected memories like MRAM with EDAC (Error Detection and Correction) must coexist with cryptographic operations, making it necessary to verify that the manipulation of input buffers (Ciphertext, AAD, Tag) and output (Plaintext) is not corrupted by the environment.
- In-Orbit Reliability: Verification of the algorithm's correctness on a specific architecture is a fundamental prerequisite for Mission Assurance.

Verification using NIST KAT on the real hardware closes the validation loop, proving that the specific software implementation and cryptographic library behave as expected on the target architecture.

4.3.2 Implementation of NIST Known Answer Tests (KAT)

To validate the AES-GCM implementation, a test suite based on the Known Answer Tests (KATs) defined by the National Institute of Standards and Technology (NIST) was developed. These tests involve executing the algorithm with specific sets of known input data (Key, IV, AAD, Authentication Tag, Ciphertext) and comparing the generated output (Plaintext) with pre-calculated, published NIST output values. The implementation and verification process was as follows:

- Selection of Test Vectors: Relevant test vectors for the AES-256 operation in GCM mode (specifically the decrypt tests) were extracted from the **Cryptographic Algorithm Validation Program**, published by NIST.
- Test Driver Development: A customized build containing the software module responsible to do the cryptographic testing was created and executed on the target platform.
- Execution on Hardware: The WolfCrypt AES-GCM implementation was invoked on the target OBC to perform the decryption and authentication operation for each set of vectors.
- Result Verification: After execution, the generated Plaintext, if the expected result of the decrypt operation was successful, was compared bit-by-bit with the expected value (Known Answers) from NIST.

```
[Keylen = 256]
[IVlen = 96]
[PTlen = 128]
[AADlen = 0]
[Taglen = 120]

Count = 0
Key =
   e51f5a7db5a4fe8c5dc78a105ad263ac81d740d2f26034040fae0cce0722e515

IV = e874dab5de4319958379d42d
CT = 82424f7297e2c24f835f10c9605c3b4f
AAD =
Tag = be1d7f4f51bb87ebbf2c5e567948a7
FAIL
```

Listing 4.6: KAT

The code snippet in Listing 4.6 shows one of the NIST Known Answer Test (KAT) vectors, in which all parameters (Key, IV, Ciphertext, AAD, and Tag) are provided with fixed lengths. The test suite consists of multiple vectors, and each vector is structured into a block containing 15 individual test cases. In this specific example, the first test case is *expected to fail* if the cryptographic implementation is correct, since its purpose is to verify that the system properly detects invalid authentication data and rejects the message.

Chapter 5

Integration of MISRA C and CWE for the Modernization of SG34

The objective of this chapter is to analyze the relationship between MISRA C and MITRE CWE, and to determine whether cybersecurity-oriented classifications (CWE) can be mapped onto coding rules used in embedded systems (MISRA C). The goal of this study is to update and modernize the internal coding standard SG34, making it both safety-oriented (MISRA C philosophy) and security-aware (CWE vulnerability taxonomy).
MISRA is widely used in safety-critical embedded software. CWE provides a vulnerability classification recognized internationally in cybersecurity.
However:

- MISRA does not explicitly classify vulnerabilities,
- CWE does not prescribe coding rules.

By studying these elements together, the scope is to evaluate how SG34 can be supported by a set of security standards.

5.1 SG34 Coding Standard

The SG34 internal coding standard defines the software development rules adopted by Avio, covering all phases from requirements analysis to design, coding, testing, delivery, and maintenance. It applies to all on-board software, both specifically developed and reused components.

The standard mandates consistency and traceability across lifecycle phases, prescribing uniform naming conventions for constants, variables, operations, and types. It enforces rules that promote readability, maintainability, and determinism, including limits on loop nesting depth, forbidding dynamic memory allocation and goto statements, requiring single return points in functions, and avoiding equality comparisons between floating-point values. Coding rules are language-independent but include language-specific profiles for Ada, C, and Assembly, reflecting the high reliability constraints of space-grade real-time embedded systems. Overall, SG34 ensures uniformity, verifiability, and safety, reducing integration effort and improving review and testing efficiency.
It includes rules for the entire software lifecycle:

Table 5.1: Key SG34 Rules by Development Phase

Phase / Area	**SG34 rules include...**
Requirements & Design	document structure, traceability, architecture constraints
Coding	indentation, naming convention, modularity, cyclomatic complexity limits, no recursion or hidden control flow
Testing	unit test planning, integration tests, coverage requirements
Configuration & Documentation	versioning, reviews, code ownership

Many of the existing rules aim at portability, style consistency, readability, and maintainability. However, SG34 is outdated today especially because it lacks explicit connection to security concepts.
Updating SG34 requires adding:

- safety in C → enforced via MISRA C rules
- security awareness → via CWE classifications

5.2 MISRA C: Reliable Coding for Embedded Systems

MISRA C is a widely adopted set of guidelines that defines a safe and secure subset of the C programming language for high-integrity software development. Its mission is to provide "world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software"[13]. Originally focused

on safety-critical embedded applications, MISRA C has evolved into a reference also for cybersecurity-oriented development, addressing vulnerabilities inherent to the C language. The guidelines achieve this by restricting unsafe language constructs and enforcing disciplined coding practices, with the goal of creating a subset of C “in which the opportunity to make mistakes is either removed or reduced”. Recent addenda demonstrate MISRA’s proactive alignment with other security standards: MISRA C:2023 Addendum 3[13] shows full coverage of CERT C secure coding rules, strengthening the claim that MISRA C is applicable not only to safety but also to security. Similarly, MISRA C:2025 Addendum 5[14] maps MISRA rules to weaknesses in MITRE’s Common Weakness Enumeration (CWE), specifically addressing memory-safety related weaknesses through explicit, implicit, or restrictive guidelines.
Together, these documents confirm that MISRA C provides structured, enforceable rules that reduce the risk of typical C vulnerabilities and support the development of robust, secure, and portable software.
The analysis of the MISRA addenda shows that many security-related vulnerabilities are already mitigated by MISRA rules, either explicitly, implicitly, or through restrictive prohibitions of unsafe language constructs. The goal of this analysis is to identify which CWE classes are covered by the MISRA ruleset, thereby strengthening and expanding the security-oriented metrics of the static analysis activities performed internally at AVIO.
MISRA C is organized into:

- Directives: organizational or process constraints
- Rules: syntactic and semantic restrictions on the language itself (what is allowed or forbidden)

5.2.1 Areas Covered

MISRA rules span almost every critical aspect of developing embedded C software. They are grouped into thematic categories:

Table 5.2: Coding Standard Categories, Focus, and Purpose[15]

Category	**Focus Area**	**Purpose**
Directives (D.x)	Process-level compliance (documentation, verification, justification)	Ensure traceability and reviewability of code decisions
General Rules (1.x, 2.x)	Language behavior, avoidance of undefined or unspecified behavior	Ensures code behaves identically on all platforms
Declarations and Initialization (8.x, 9.x)	Variable lifetime, scope, initialization requirements	Prevents uninitialized memory usage and improves determinism
Arithmetic and Type Rules (10.x, 12.x)	Type conversions, integer promotion, signed/unsigned operations	Prevents overflow, truncation, loss of precision
Pointers and Memory (11.x, 18.x)	Pointer casts, pointer arithmetic, dynamic memory	Eliminates unsafe pointer operations and memory risks
Control Flow (13.x, 14.x)	if/else, loops, switch, recursion	Avoids dead code, invariant conditions, and hidden control flow
Expressions (12.x)	Operator precedence, side effects, sequential points	Ensures clarity and avoids ambiguous expressions
Preprocessor Use (19.x)	#define, macros, conditional compilation	Limits use of preprocessor to avoid untraceable logic
Standard Library (20.x)	Limits usage of C standard library	Avoids non-deterministic or non-portable library functions

Some MISRA rules prohibit entire language features when they introduce risk:

- **Dynamic memory allocation** (malloc, free) → not allowed in freestanding/embedded environments
- **Recursion** → forbidden due to unpredictable stack usage
- **Implicit type conversion** → forbidden due to hidden loss of precision

This restrictive philosophy is justified in high-integrity embedded software, where predictability is more valuable than flexibility.

5.2.2 Why MISRA is used as the embedded development Standard

MISRA is considered as the standard in high-criticality systems for three reasons:

- **Determinism**: forbidden behaviors prevent undefined or compiler-dependent execution.
- **Portability**: code behaves the same across compilers, architectures, and toolchains.
- **Static analyzability**: MISRA is designed to be automatically verifiable with tools such as Polyspace, which supports rule checking and compliance reporting.

Furthermore, MISRA Addendum 5[14] maps MISRA guidelines to CWE weaknesses, proving that MISRA inherently covers memory safety and security concerns.

5.3 CERT C: Missing link between MISRA and CWE

The SEI CERT C Coding Standard[16] defines rules for developing safe, reliable, and secure software in the C language. Its purpose is to eliminate undefined and insecure behaviors that may lead to exploitable vulnerabilities. Each rule specifies mandatory requirements, together with non-compliant and compliant code examples, helping developers understand how to write secure C code and how static analysis tools can detect violations. CERT C focuses explicitly on security and targets developers of systems that must be reliable and resistant to attack. MISRA C:2023 Addendum 3[13] maps CERT C rules to MISRA rules, showing that CERT C serves as a bridge between MISRA's safety-oriented guidelines and security-oriented vulnerability taxonomies such as CWE. By exploiting this mapping, it becomes possible to identify which security issues (classified as CWEs) are already mitigated through MISRA rules.

5.4 CWE: Common Weakness Enumeration

The Common Weakness Enumeration[17] (CWE) is a community-developed classification system of software and hardware weakness types, maintained by The MITRE Corporation, which aims to provide a unified language for describing the root causes of vulnerabilities. At its core, CWE defines a weakness as "a condition in software, firmware or hardware, that when introduced during development or operation, may enable or contribute to an exploitable vulnerability"[18]. The CWE list is structured as a hierarchical taxonomy, with entries ranging from very abstract categories to specific language or technology-dependent variants, enabling developers, testers and static analysis tools to map individual issues to standardized identifiers. The current CWE repository includes hundreds of distinct weakness types and supports multiple "views" to aid navigation and analysis. One of the key uses of CWE is enabling measurement and tracking of weaknesses across tools, codebases and architectures, thereby supporting metrics such as how many instances of particular weakness classes have been identified or mitigated. By embedding CWE identifiers into static analysis outputs or compliance dashboards, organizations can extend their assessments from generic "non-compliance" flags toward measurable "weakness-type coverage" metrics.

5.5 Synthesis: MISRA - CERT C - CWE

The integration of MISRA C, CERT C, and CWE enables a unified methodology to evaluate both safety and security in C code. MISRA C acts on the language itself, eliminating unsafe or undefined constructs, while CERT C defines secure coding practices and bridges MISRA guidelines to concrete vulnerability classes. CWE provides the taxonomy to classify weaknesses and quantify their security impact. Their analysis followed a two-step approach:

1. the direct correspondence between MISRA C and CWE, as officially provided in the MISRA Addendum 5[14], was reviewed to establish a baseline understanding of how MISRA rules inherently address known software weaknesses. This initial mapping highlights MISRA's role in mitigating several CWE categories, particularly those related to memory safety, control flow integrity, and type correctness.

2. A reverse-correlation was developed, between CERT C, MISRA and CWE. Since CERT C explicitly links its secure coding rules to CWE identifiers, it serves as a methodological bridge, enabling the derivation of additional and more granular correspondences between MISRA rules and CWE weakness classes. This cross-standard correlation was reached also thanks to MISRA Addendum 3 [13], which focuses on mapping CERT C and MISRA rules together.

The MISRA Addendum 5 [14] defines several coverage types, and each mapping is assigned one of them.

5.5.1 Analysis of selected MISRA-CWE mappings

CWE-119: Improper Restriction of Memory Buffer Bounds

This weakness represents one of the most critical categories of memory-safety vulnerabilities in C. It occurs when operations exceed allocated buffer limits, leading to memory corruption or code execution. MISRA C mitigates this class of

Table 5.3: Type of coverage [14]

Type of coverage	**What it means**	**interpretation in practice**
Explicit	The weakness is directly addressed by one or more MISRA rules	A rule exists that exactly specifies what to not do
Implicit	MISRA rules prevent the weakness, but the CWE is not named explicitly	Following the rules naturally avoids the weakness
Restrictive	MISRA solves the weakness by prohibiting the dangerous language feature entirely	E.g. banning dynamic memory in critical code removes entire CWE classes
Partial	MISRA covers some but not all aspects of the weakness	Additionalk control outside MISRA may be required
Collection	Multiple MISRA rules together mitigate the CWE	No single rule solves it, but a combination of rules does

weaknesses through a collection of rules addressing arrays, pointers, and dynamic memory. Rules R.9.1 and R.11.1–R.11.6 impose strict boundaries on array indexing and pointer conversions, while R.18.1 limits pointer arithmetic to valid object ranges. Additional safeguards from R.21.6, R.21.17, and R.21.18 restrict unsafe standard library functions known to cause buffer overflows. Since dynamic memory allocation is typically forbidden in high-critical embedded systems, MISRA encourages static allocation and compile-time sizing, reducing the attack surface and eliminating unpredictable memory behavior. Collectively, these rules provide strong coverage under the Collection classification, preventing out-of-bounds memory access at its root.

CWE-120/121/122: Buffer Overflow, Stack Overflow, Heap Overflow

The CWE series 120–122 extends CWE-119 to specific memory regions, emphasizing how incorrect input handling or pointer misuse can overwrite adjacent memory on the stack or heap. MISRA mitigates these weaknesses primarily through R.18.1, which restricts pointer manipulation, and through the R.21.x family, which discourages dangerous library functions such as strcpy and sprintf. Furthermore, R.1.3 prohibits undefined behavior, ensuring that memory access patterns remain well-defined. In high-critical software, dynamic heap allocation is often disallowed, which further reduces exposure to heap-based overflows and eliminates run-time fragmentation and non-determinism. MISRA's rules therefore enable predictable memory usage and reduce the attack surface associated with buffer overflows.

Table 5.4: MISRA C rules mapped to CWE IDs [14]

CWE Weakness	**Mapped MISRA Rule**	**Type of coverave**	**Strength**
CWE-119: Improper Restriction of Memory Buffer Bounds	R.9.1 + R.11.1 + R.18.1 + R.21.6/21.17/21.18 + R.22.2 + D.4.7 + R.1.3	Collection	Strong
CWE-120/121/122: Buffer overflow / stack overflow / heap overflow	R.18.1 + R.21.x rules + R.1.3	Implicit	Strong
CWE-415: Double Free	R.22.2 + D.4.x + R.1.3	Partial / Restrictive	Strong
CWE-476: NULL Pointer Dereference	D.4.1(defensive coding)	Implicit	Weak
CWE-590: Freeing stack memory	R.22.2	Explicit	Strong
CWE-822: Pointer to invalid memory location	R.11.1-6 + D.4.7 + D.4.14	Explicit	Strong
CWE-780-789(memory/pointer handling variants)	R.18.1 + R.21.6/21.17/21.18	Partial / Restrictive	Strong

CWE-415: Double Free

This weakness occurs when the same dynamically allocated memory region is freed more than once, potentially resulting in heap corruption or arbitrary code execution. MISRA reduces exposure to this risk by limiting or outright prohibiting dynamic allocation in safety-critical systems. Rule R.22.2 enforces that memory obtained via allocation functions must be released exactly once, and directives such as D.4.x encourage runtime checks before deallocation. Combined with R.1.3, which prohibits undefined behavior, these rules ensure controlled memory lifecycles. In many high-reliability environments, dynamic allocation is avoided altogether, meaning that the conditions leading to double-free situations are structurally eliminated.

CWE-476: NULL Pointer Dereference

Dereferencing a null pointer can lead to crashes or unpredictable system behavior. MISRA addresses this issue implicitly through D.4.1, which promotes defensive programming practices such as validating pointer values before dereferencing. Although the coverage is considered weak, it still ensures that software designed under MISRA principles systematically validates all inputs and references. This approach transforms potential null-pointer dereferences into detectable, controlled errors rather than fatal faults, aligning with MISRA's emphasis on predictable system behavior.

CWE-590: Freeing Stack Memory

This CWE targets incorrect deallocation of non-heap memory, for example when using `free()` on stack-allocated variables. MISRA explicitly forbids this pattern under R.22.2, which requires that only memory obtained via dynamic allocation functions (e.g., `malloc`, `calloc`) may be released. The coverage is explicit and strong, since the rule directly addresses this dangerous practice. By eliminating the possibility of freeing invalid memory regions, MISRA ensures runtime stability and guards against memory corruption that could otherwise arise from improper deallocation.

CWE-822: Pointer to Invalid Memory Location

Invalid pointer references often occur after deallocation, incorrect typecasting, or pointer arithmetic errors. MISRA offers explicit and strong coverage through R.11.1–R.11.6, which prohibit unsafe pointer conversions, and through D.4.7 and D.4.14, which mandate defensive programming and pointer validity checks. These rules ensure that pointers always reference valid, well-defined objects throughout their lifecycle. This alignment between MISRA rules and CWE-822 demonstrates MISRA's capability to enforce memory safety at the language-usage level, thereby preventing access to stale or non-existent memory.

CWE-780–789: Memory/Pointer Handling Variants

These CWEs encompass several specialized memory-safety and pointer-handling issues, such as improper pointer scaling or incorrect object referencing. MISRA mitigates these through R.18.1 and the standard-library restrictions defined in R.21.6, R.21.17, and R.21.18. The Addendum identifies this mapping as partial/restrictive with strong coverage, reflecting the fact that MISRA not only defines rules to prevent unsafe pointer arithmetic but also disallows the use of dangerous library functions that could introduce these vulnerabilities. Collectively, these constraints foster robust pointer discipline and help maintain strict control over memory access semantics.

5.6 CERT C: additional cross-check mapping

The additional mappings presented in this section were derived through a reverse-correlation methodology. Although a direct mapping between MISRA C and CWE was already analysed, this correspondence alone does not fully capture the semantic links between coding rules and weakness classes, due to the different scopes and abstraction levels of the two frameworks. For this reason, the analysis was complemented by a CERT C–based approach, leveraging the fact that CERT rules explicitly associate coding practices with well-defined CWE categories. Starting from each relevant CERT C rule, the corresponding MISRA C rules enforcing the same constraint were identified. Since CERT C rules already include a systematic mapping to MITRE CWE categories, this intermediate step allowed the final derivation of an indirect but reliable relationship between MISRA C and the CWE weakness taxonomy. Through this "CERT C → MISRA C → CWE" cross-check process, it becomes possible to determine not only whether MISRA covers a specific weakness, but also how strongly (Explicit, Implicit or Restrictive), enabling a deeper assessment of MISRA C as a security-coding standard.

Table 5.5: MISRA C rules mapped to CERT C and CWE IDs

CERT C Rule	**CWE Description Summary**	**Related MISRA C Rule ID**	**Related CWE IDs**
INT31-C, FLP34-C	Integer coercion / truncation errors, incorrect numeric conversion	10.1, 10.3, 10.4, 10.6, 10.7	192, 197, 681
ARR30-C	Out-of-bounds access, incorrect pointer/index use, buffer sizing issues	18.1, 21.17, 21.18	119, 125, 788
EXP33-C	Out-of-bounds read, improper initialization	9.1, 21.3	125, 665
EXP39-C, INT36-C	Strict aliasing violation, incorrect pointer–integer conversion	11.1, 11.2, 11.3, 11.7	119, 123, 125, 466, 587
MEM31-C, MEM34-C	Memory/file descriptor leaks, freeing non-heap memory	22.1, 22.2, Dir 4.12	401, 415, 416, 590

5.6.1 Integer Conversion Safety - INT31-C & FLP34-C $\rightarrow$ CWE 192, 197, 681

CERT C Rule INT31-C demands that integer conversions must not result in lost or misinterpreted data[16]. This imperative is addressed by a stringent family of MISRA C rules, collectively providing an Explicit Strong defense[13].

- MISRA C Rules 10.1 through 10.4, 10.6, and 10.7 rigorously control implicit and explicit type conversions to prevent truncation, sign errors, and loss of precision. For example, Rule 10.1 dictates that "Operands shall not be of an inappropriate essential type"[13]. This proactively manages the risk that a value exceeding the representable range of a smaller type is truncated (CWE-197), or that an incorrect conversion logic is used (CWE-681).
- The same strong ruleset applies to floating-point conversions (FLP34-C), explicitly covered by MISRA Rules 10.3, 10.4, 10.5, and 10.8.

5.6.2 Memory and Pointer Integrity

Pointer-related vulnerabilities form the bulk of critical exploits (CWE-119, CWE-125). MISRA's role here is to impose rigorous, explicit constraints on pointer manipulation.

Out-of-Bounds Access: ARR30-C $\rightarrow$ CWE 119, 125, 788

CERT C Rule ARR30-C prohibits forming or using out-of-bounds pointers or array subscripts[16]. This maps directly to memory safety vulnerabilities.
MISRA C Rule 18.1 ("A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand"[13]) explicitly confines pointer arithmetic to valid array boundaries. This rule, alongside related rules 21.17 and 21.18, provides Implicit Strong protection[13]. The implicit nature stems from MISRA defining this as an integrity or portability rule, rather than an explicit "buffer overflow prevention" measure, but the practical effect against CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) is robust.

Uninitialized Memory Access EXP33-C $\rightarrow$ CWE 125, 665

Reading uninitialized memory (EXP33-C) can lead to data exposure (CWE-125) or unpredictable control flow (CWE-665, Improper Initialization)[13].
MISRA C Rule 9.1 mandates that "The value of an object with automatic storage duration shall not be read before it has been set"[13]. This is backed by Dir 4.1 and

Rule 21.3, yielding an Explicit Strong coverage[13]. This is a fundamental safeguard against a class of errors highly prioritized by safety and security standards.

Incompatible Type Access and Aliasing (EXP39-C & INT36-C → CWE 119, 123, 125)

The practice of accessing a variable through a pointer of an incompatible type (strict aliasing violation, EXP39-C) triggers Undefined Behavior and can lead to memory corruption or data leaks[13].
MISRA addresses this explicitly and strongly through the Rule 11.x family. Rules 11.1, 11.2, 11.3, and 11.7 strictly govern pointer conversions (e.g., preventing conversion between a pointer to an object type and a pointer to a different object type). This comprehensive restriction on type punning ensures the compiler cannot perform detrimental optimizations based on false non-aliasing assumptions, thus mitigating the root cause of CWE-119 and CWE-125 exploits stemming from pointer misuse. CERT C Rule INT36-C (converting a pointer to integer or integer to pointer) is also managed by this family of rules (11.1, 11.2, 11.4, 11.6, 11.7, Dir 1.1)[13].

Dynamic Memory Management (MEM31-C, MEM34-C → CWE 401, 415, 416, 590)

MISRA adopts a high-level, design-based restriction for memory management.
MISRA C Directive 4.12 requires that "Dynamic memory allocation shall not be used"[13]. This prohibition immediately grants a Restrictive Strong coverage against the entire class of heap-based memory vulnerabilities. This highly effective security measure eliminates issues such as Double Free (CWE-415), Use After Free (CWE-416), Memory Leak (CWE-401, addressed by MEM31-C), and freeing non-heap memory (CWE-590, addressed by MEM34-C, via Rule 22.2)[13].

5.7 Summary

From a comparative standpoint, SG34 already provides substantial coverage of many practices later formalized in MISRA C and linked, through CERT C and CWE, to concrete security weaknesses. Core SG34 constraints, such as *the prohibition of dynamic memory allocation, the enforcement of deterministic control flow (no recursion, no hidden control flow), disciplined pointer usage, mandatory initialization, and strict rules on documentation and traceability*, are conceptually aligned with MISRA directives and rules. Through the MISRA–CERT–CWE mappings, these practices can be seen as indirectly mitigating critical weakness classes such as buffer overflows, pointer misuse, double free, improper initialization, and use of invalid memory. However, SG34 remains mostly safety-oriented and does not explicitly address security taxonomies, nor does it systematically restrict dangerous standard library functions or type conversions as rigorously as MISRA and CERT C. As a result, SG34 offers strong but implicit coverage for many CWE categories, while leaving gaps in areas such as vulnerability classification, strict handling of integer and floating-point conversion, aliasing, preprocessor safety, and the explicit treatment of security-specific behaviors. Aligning SG34 with MISRA, CERT C, and CWE therefore turns it from a purely safety-driven coding standard into a framework that also provides measurable security coverage.

Table 5.6: Summary of SG34 coverage w.r.t. MISRA C / CERT C / CWE

Area	**Related MISRA / CERT / CWE concepts**	**SG34 coverage**
Prohibition of dynamic memory allocation	MISRA Dir 4.12, R.22.x; CERT MEM31-C, MEM34-C (CWE-401, 415, 416, 590)	Strong coverage (heap-related weaknesses structurally removed)
Deterministic control flow (no recursion, limited nesting, no hidden flow)	MISRA 13.x, 14.x; CERT EXP rules (CWE-480, CWE-691)	Strong, implicit coverage of control-flow related weaknesses
Pointer discipline and restricted pointer arithmetic	MISRA 11.x, 18.x; CERT ARR30-C, INT36-C (CWE-119, 125, 466, 822)	Partial coverage of memory-safety issues
Mandatory initialization and deterministic data lifetime	MISRA 9.x; CERT EXP33-C (CWE-665, 125)	Strong coverage against uninitialized use
Naming, readability, modularity, documentation and traceability	MISRA general guidelines and directives	Strong coverage for safety, indirect benefit for security
Restrictions on unsafe C standard library functions (string and memory APIs)	MISRA 21.x; CERT STR/MSC rules (various CWE-119/120/121/122)	Weak coverage in SG34 (not explicitly codified)
Integer and floating-point conversion safety (overflow, truncation)	MISRA 10.x; CERT INT31-C, FLP34-C (CWE-190, 192, 197, 681)	Partially covered by general SG34 principles
Null pointer checks and defensive pointer validation	MISRA D.4.1; CERT EXP34-C (CWE-476)	Weak / implicit coverage (recommended, but not fully formalized)
Strict aliasing and type-punning control	MISRA 11.x; CERT EXP39-C, INT36-C (CWE-119, 123, 125)	Partial coverage; SG34 does not define an explicit "essential types" model
Explicit vulnerability taxonomy and security metrics	CERT → CWE mapping, MISRA Addenda 3[13] and 5[14]	Not covered: SG34 does not reference CWE or security coverage metrics

Chapter 6

Polyspace Static Analysis Tools in Embedded C Verification

In an embedded systems verification workflow, Polyspace Bug Finder and Polyspace Code Prover serve as complementary static analysis tools to improve code safety and reliability without executing the software. Both tools analyze C (and C++) source code for onboard software to detect defects early and verify correctness, which is crucial in safety-critical domains like automotive or aerospace. Unlike dynamic testing, Polyspace analysis does not run the code on hardware, but it inspects the code statically (using mathematical methods) to find problems and prove the absence of certain errors[19]. Below the purpose and analysis methods of each tool and the types of defects they target.

6.1 Polyspace Bug Finder: Static Analysis for Potential Defects

Polyspace Bug Finder is a static code analyzer designed to scan embedded C/C++ code for potential bugs, vulnerabilities, and coding standard violations at an early development stage, without executing the code. It examines the code's structure and flows to catch issues as soon as the code is written or integrated. Internally, Bug Finder performs semantic static analysis of the program's control flow, data flow, and interprocedural interactions to spot anomalies[20]. This means it tracks how data moves through the program and how functions call each other, looking for anything suspicious, for example, variables used before initialization, null pointers that might be dereferenced, or array indices that could go out of bounds. Because it's static (no actual running of the program), it explores these paths symbolically to flag potential run-time errors (like overflows or divide-by-zero) and logic bugs in the code. The analysis is relatively fast and tries to minimize false positives, so developers can quickly focus on real issues.

6.1.1 Purpose and defect detection

The primary purpose of Bug Finder is to act as a bug-hunting tool for developers: essentially an automated code review that catches runtime error conditions, dangerous code patterns, and bad practices before the code ever runs. It identifies a broad range of defects such as numeric errors (e.g. overflow or underflow possibilities), invalid pointer usage (null or dangling pointer dereferences), buffer and array overruns, division by zero risks, and data flow problems like use of uninitialized variables. All of these are detected through static analysis algorithms that symbolically evaluate how the code could execute in different scenarios. Importantly, no actual code execution or test vectors are needed; Bug Finder infers problems by analyzing the code itself, so it can find subtle bugs that might only surface under rare conditions. By catching these issues early, developers can fix them in the development cycle rather than during testing or deployment.

6.1.2 Coding standards and metrics

In addition to finding bugs, Polyspace Bug Finder also checks the code against industry coding standards and best practices, which is especially relevant for embedded software quality. It has built-in rules for several guidelines, and any violations of these rules, such as misusing language features, unsafe constructs, or deviating from style mandates, are reported. This helps ensure the code not only is bug-free but also compliant with standards required for certification. Furthermore, Bug Finder gathers code quality metrics like cyclomatic complexity, code size, and other maintainability metrics. These metrics give insight into the complexity of functions and can signal high risk modules. The tool outputs comprehensive reports listing all the bugs found, coding rule violations, and metrics. Overall, Bug Finder's role is preventative: it performs a wide sweep for potential problems and enforce good coding practice, improving code quality before deeper verification is done.

6.1.3 Bug Finder analysis in the project

A static analysis on a subset of the project has been conducted using Polyspace Bug Finder. The tool scanned the codebase and reported 83 findings distributed across the involved testing units. Most issues detected in the application code were

minor and related to structural or maintenance weaknesses such as empty code blocks(CWE-1071), indicating areas where initialization routines were declared but not yet implemented. Some of the scanned issues were about unsafe type conversions (CWE-704, CWE-758) and integer overflows/underflows (CWE-190, CWE-191, CWE-128, CWE-197) occurring when casting between pointer-sized and integer-sized types or when narrowing 32-bit values into 8-bit variables. Importantly, critical classes of memory-safety vulnerabilities such as buffer overflows (CWE-120/121/122) or use-after-free (CWE-416) were not observed in the analyzed files, confirming that the application code adheres to safe memory usage patterns and that the integration of the crypto library does not introduce uncontrolled dynamic memory operations.
Overall, the analysis demonstrated that the project does not present high-risk memory-safety defects.

6.2 Polyspace Code Prover: Formal Verification of Run-Time Errors

Polyspace Code Prover is a more advanced static analysis tool that uses formal methods to mathematically prove properties about the code's correctness. Its main purpose is to verify the absence of certain critical run-time errors in C code by exhaustively analyzing all possible executions, inputs, and values that the program can take without running the program on a target. Under the hood, Code Prover implements abstract interpretation, a formal methods technique that interprets the program with abstract values representing all possible real values, to explore every feasible path and state of the program. This exhaustive analysis is aimed at proving that no runtime error of a certain class can ever occur when the software runs. In other words, while Bug Finder might say "this line might cause a null pointer dereference if X happens," Code Prover goes further to determine conclusively whether such an error can or cannot happen in any scenario. It verifies each operation in the program for correctness with respect to runtime errors, effectively acting like a mathematical proof checker for the code.

6.2.1 Purpose and analysis method

The focus of Code Prover is on deep formal verification of critical code. It is typically used on the most safety-critical components of embedded software, where you need absolute confidence that no catastrophic run-time errors will occur. Code Prover analyzes the program by considering all possible inputs and states, thanks to its formal approach. It performs value range analysis on variables and expressions, deducing the possible range of values each variable can take at each point in the code. By propagating these ranges through the control flow, it can prove facts such as “this index will always stay within array bounds” or “this pointer will never be null when dereferenced,” etc. Importantly, Code Prover checks for runtime errors that include: buffer overflows (array index out of bounds), arithmetic overflows/underflows, division by zero, null or invalid pointer dereferences, out-of-range numeric casts, and other undefined-behavior cases that could crash or corrupt a program. If the analysis can guarantee that none of these errors can happen, it will consider the operation proved safe. This process is exhaustive in the sense that it simulates all feasible control paths and combinations of input values within the program’s constraints. Behind the scenes, the tool builds an exhaustive function call tree and a global memory access map of the program. This means it understands every function call sequence and how memory (global variables, etc.) is accessed across the program, which helps in reasoning about interactions and side-effects across the entire codebase. All this is done through static analysis algorithms based on formal semantics of the language, ensuring that the verification is sound.

6.2.2 Proving absence of defects

The output of Polyspace Code Prover is not just a list of potential issues, but a guarantee level for each operation in the code. Each statement or operation in the source code gets a color-coded result indicating its proven status. In practice, Green code means that Code Prover has proven that this operation cannot fail (e.g. an array access that is proven always in bounds, or a division proven never to have a zero divisor). Red means a definite error: the tool found a combination of inputs or a path that will lead to a runtime error (so a bug that needs fixing). Orange indicates unproven or potentially unsafe: the analysis could not conclusively prove the operation safe, which often means there is a possible but not guaranteed error (or the tool had to make a conservative assumption), and thus it flags it for review. Gray denotes code that wasn’t executed in any feasible path (dead code), such as a function that is never called or an if-branch that is never true. This color-coding provides developers and verification engineers a clear map of the code’s health: ideally, all critical sections should be green (proven safe). If any red or orange remains, those are either real bugs or areas that need further scrutiny or perhaps additional assumptions. Achieving all-green status in Code Prover gives a high level of confidence that the embedded software is free of run-time crashes like overflows and invalid memory accesses.

Because Polyspace Code Prover uses formal verification, it is computationally heavier than Bug Finder (analysis can take longer on large codebases), but it delivers a higher assurance level. It’s often used in later stages of development or on subsets of the code that are safety-critical, after using Bug Finder to clean out the obvious issues. No code execution is required for this verification; the proofs are all done through static mathematical reasoning on the source code. In essence, Code Prover acts like an automated code proof assistant, showing that under all possible operating conditions, certain classes of errors will not happen.

6.2.3 Code Prover analysis in the project

The Polyspace Code Prover analysis reported a total of 159 open runtime checks, all marked in orange, indicating operations for which the tool could not conclusively prove safety. These checks span across data flow (118 issues), numerical operations (8), static memory usage (24), and a few under miscellaneous categories (9). Notably, among the orange results are warnings for illegally dereferenced pointers, potential integer overflows, and invalid shift operations, all of which correspond to high-impact CWEs such as CWE-119, CWE-190, and CWE-681. These unresolved proofs highlight areas where runtime behavior may be unsafe under certain input conditions or data paths, particularly in embedded environments where deterministic behavior is critical. The complete absence of "red" errors suggests that no proven unsafe operations were detected. However, the presence of unresolved orange checks emphasizes the need for refinement. In particular, applying Data Range Specification (DRS) techniques could help convert many of these orange results into green outcomes. As such, Code Prover acts as a formal diagnostic stage, revealing which operations require further adjustments or annotations to attain proof completeness in safety-critical software components.

6.3 Combined Use in the Verification Workflow

Polyspace Bug Finder and Code Prover are often used in tandem to achieve robust static verification for embedded software. In a typical workflow, developers first run Bug Finder on new or modified code to quickly identify and fix all the obvious bugs, dangerous constructs or standard violations. This improves the code quality and ensures compliance with coding standards early on. Subsequently, Code Prover is applied to the cleaned-up code to perform deep formal verification on the most critical components, confirming that no runtime error conditions remain. Together, the tools provide an end-to-end static analysis solution: Bug Finder's fast bug detection and coding rule checking combined with Code Prover's exhaustive proof capabilities. This combination means that by the time the code is ready to deploy on the embedded hardware, it has been both scanned for defects and formally proven for key correctness properties, all without executing a single line on target. This approach greatly increases confidence in the software's reliability.

In the context of onboard embedded software, where failures can be catastrophic, using these tools is now a common practice to meet safety and quality requirements. The results from Polyspace analyses can be used to produce evidence for certification and standards compliance. For example, Polyspace outputs can be turned into reports that demonstrate code quality, list all found issues (with justifications for any that are deemed acceptable), and show which parts of the code have been proven free of run-time errors, which is very useful for auditors or safety engineers. By integrating Bug Finder and Code Prover into the development lifecycle, teams can continuously monitor code for issues and regressions. This ensures that the embedded C code for onboard systems remains robust: potential defects are caught early by static analysis, and the final code is backed by mathematical proofs of correctness for critical safety properties. The end result is highly reliable embedded software, with Polyspace providing a line of defense against runtime bugs long before the code ever runs on the actual device.

Chapter 7

Conclusion

This thesis presented the analysis, design, implementation, and validation of a secure software patching mechanism for the Avum Orbital Module (AOM), a core element of ESA's Space Rider mission. The work addressed a critical challenge in modern space systems: enabling safe, authenticated, and resilient software updates throughout the mission lifecycle, despite the constraints imposed by deterministic real-time architectures, limited computational resources, and the absence of physical access once in orbit.

The project began with a comprehensive exploration of the AOM hardware and software architecture, emphasizing the role of the GR740 processor, the PikeOS hypervisor, and the layered organization of the onboard software (OBSW). Understanding these foundations was essential to ensure that the proposed security mechanism could be integrated without compromising real-time behavior, fault tolerance, or mission-critical operations.

A complete end-to-end update pipeline was then designed, based on AES-GCM as an Authenticated Encryption with Associated Data (AEAD) algorithm. This choice ensured that each patch delivered to the spacecraft benefits from confidentiality, integrity, authenticity, and contextual binding via the use of Additional Authenticated Data (AAD). The mechanism was integrated within a customized extension of PUS Service 6, enabling secure handling of segmented payloads without relying on unsupported PUS large-packet services. A robust anti-replay mechanism, based on a monotonic patch counter, ensured chronological integrity and prevented downgrade or replay attacks.

A key contribution of this thesis was the validation of the AES-GCM implementation directly on the GR740 flight hardware via NIST Known Answer Tests (KATs). Since no prior verification existed for this specific spaceborne platform, this activity was fundamental to guarantee correctness, robustness and suitability for onboard use.

The implementation of the decryption functionality required the creation of a dedicated PikeOS partition, FLASH, configured with strict spatial and temporal isolation requirements and integrated seamlessly within the real-time execution model.

In parallel, the thesis investigated how modern software assurance practices could strengthen the existing SG34 coding standard. By analyzing the correspondence between MISRA C guidelines, CERT C recommendations, and the CWE weakness taxonomy, the work highlighted how many safety-driven MISRA constraints also implicitly mitigate security flaws, particularly those related to memory safety, pointer misuse, and undefined behaviors. This mapping also revealed gaps where SG34 could be extended with explicit security rules, thereby aligning development practices with the new ECSS cybersecurity mandates.

Overall, this work delivered both a practical operational capability (a secure, validated patching mechanism ready for integration into Space Rider) and a methodological contribution, showing how safety-critical coding standards and modern cybersecurity taxonomies can converge toward unified software assurance principles. In doing so, the thesis supports ESA's strategic shift toward treating cybersecurity not as an optional add-on, but as a foundational engineering discipline embedded throughout the mission lifecycle.

The integration of this mechanism demonstrates that spaceborne systems can achieve higher levels of cybersecurity without compromising safety, determinism, or performance. As space missions become increasingly autonomous, interconnected, and reliant on long-duration reusability, such capabilities will be essential to maintain trust, resilience, and operational superiority in an evolving threat landscape.

7.1 Future Directions

While this thesis focused on secure software update mechanisms and coding-level assurance, several promising directions remain open for future development:

Hardware-rooted security

Current cryptographic protection is performed at the software level. An evolution of this design consists in introducing security functions directly within hardware elements:

- Hardware Security Modules (HSM) for key storage and cryptographic operations
- Secure Enclaves or Trusted Execution Environments

- Physically Unclonable Functions (PUFs) to derive cryptographic keys from intrinsic silicon randomness

Moving cryptography from software to hardware increases resistance to exploitation, prevents attacks targeting memory access, and reduces the exposure of cryptographic material.

Secure Boot and chain of trust

Extending the patching mechanism with secure boot would ensure that every software image executed at startup is authenticated based on a root of trust embedded in non-modifiable memory. This would prevent persistent malware and unauthorized firmware replacement, ensuring that only cryptographically signed software can ever run on the spacecraft. Although this mechanism perfectly aligns with the security objectives of this thesis, it was not adopted in this case for specific technical reasons. In particular:

- The hardware platform used (GR740) does not natively support Secure Boot features, unlike other architectures (e.g., many ARM-based SoCs) that provide hardware support for key storage, secure bootloaders, and firmware cooperation for authentication and decryption.
- The selected bootloader (a COTS component) does not implement a secure boot chain nor comparable cryptographic validation capabilities.

Hence the need to implement a cryptographically protected software update mechanism.

Cryptographic agility and post-quantum readiness

AES-GCM is secure today, but long-duration missions require resistance against future cryptographic breakthroughs, hence future work may explore post-quantum–resistant cryptography. Additionally, crypto-agility ensures that future missions can switch algorithms without redesigning the whole architecture.

In-orbit anomaly detection and attack monitoring

With new connectivity models and increasing mission autonomy, security should evolve beyond preventive protections. Future spacecraft may integrate:

- Telemetry-based anomaly detection
- On-board intrusion detection mechanisms
- Forensic logging and tamper-evident audit trails

This shifts cybersecurity from static protection to dynamic resilience.

7.2 Final remarks

Space systems are transitioning from isolated, deterministic machines into networked, intelligent, and long-lived assets. With this transformation comes a fundamental shift: security can no longer be an afterthought imposed late in the design cycle, but must become an intrinsic property of spacecraft architectures. The secure patching mechanism and assurance framework developed in this thesis demonstrate how this shift can begin practically and without compromising safety. As missions grow more autonomous and interconnected, the ability to protect, update, and trust onboard software will become as essential as propulsion, navigation, or thermal control. The results presented here contribute to this emerging paradigm, pointing toward a future where spacecraft are not only reliable and safe, but also resilient, adaptable, and secure by design.

Bibliography

[1] Shah Khalid Khan, Nirajan Shiwakoti, Abebe Diro, Alemayehu Molla, Iqbal Gondal, and Matthew Warren. «Space cybersecurity challenges, mitigation techniques, anticipated readiness, and future directions». In: *International Journal of Critical Infrastructure Protection* 47 (2024). S1874-5482(24)00065-9. DOI: `10.1016/j.ijcip.2024.100724` (cit. on p. 1).

[2] ESA Security Office. *ESA Cyber Security Resilience Achievement.* Plan ESA-ESO-PL-2023-0068. Version Issue 1.0. European Space Agency (ESA), Oct. 2023. URL: `https://connectivity.esa.int/sites/default/files/2025-04/ESA%20Cyber%20Security%20Resilience%20Achievement.pdf` (cit. on p. 2).

[3] *Space Engineering - Software.* Tech. rep. ECSS, April 2025 (cit. on pp. 3, 31, 36).

[4] *Space Engineering - Security in Space Systems Lifecycle.* Tech. rep. ECSS, July 2024 (cit. on p. 4).

[5] ESA. *Space Rider overview.* `https://www.esa.int/Enabling_Support/Space_Transportation/Space_Rider_overview` (cit. on p. 5).

[6] ESA. *Telemetry & Telecommand.* `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Telemetry_Telecommand` (cit. on p. 15).

[7] *Space Engineering - Telemetry and telecommand packet utilization.* Tech. rep. ECSS, April 2016 (cit. on p. 18).

[8] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.* Tech. rep. NIST - National Institute of Standard and Technology, November 2007 (cit. on p. 22).

[9] *SYMMETRIC KEY MANAGEMENT.* Tech. rep. The Consultative Committee for Space Data Systems, February 2022 (cit. on p. 23).

[10] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. «Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS». In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. Available under CC0 1.0 license. USENIX Association. 2016. URL: `https://github.com/nonce-disrespect/nonce-disrespect` (cit. on p. 24).

[11] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. «Reasoning About Information Flow Security of Separation Kernels with Channel-Based Communication». In: *Lecture Notes in Computer Science*. Available on ResearchGate. 2015. DOI: `10.1007/978-3-662-49674-9_50` (cit. on pp. 37, 38).

[12] ESA. *libmcs*. `https://essr.esa.int/project/libmcs-mathematical-library-for-critical-systems` (cit. on p. 41).

[13] The MISRA Consortium Limited. *MISRA C:2023 Addendum 3 – Coverage of MISRA C:2023 against CERT C (2016 Edition)*. Tech. rep. 1 St James Court, Whitefriars, Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Jan. 2025. URL: `https://misra.org.uk/app/uploads/2025/03/MISRA-C-2023-ADD3-CERT.pdf` (cit. on pp. 45, 46, 48, 49, 54, 55, 57).

[14] The MISRA Consortium Limited. *MISRA C:2025 Addendum 5 – Coverage of MISRA C:2025 against the Common Weakness Enumeration (CWE)*. Tech. rep. 1 St James Court, Whitefriars, Norwich, Norfolk, NR3 1RU, UK: The MISRA Consortium Limited, Mar. 2025. URL: `https://misra.org.uk/app/uploads/2025/03/MISRA-C-2025-ADD5.pdf` (cit. on pp. 46, 48–51, 57).

[15] MISRA. *MISRA C:2023 Directives and Rules*. `https://uk.mathworks.com/help/bugfinder/misra-c-2023-reference.html` (cit. on p. 47).

[16] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Tech. rep. 2016 Edition. Pittsburgh, PA, USA: Software Engineering Institute, Carnegie Mellon University, 2016. URL: `https://docenti.ing.unipi.it/~a009435/issw/extra/c-coding-standard.pdf` (cit. on pp. 48, 54).

[17] MISRA. *MITRE CWE*. `https://cwe.mitre.org/` (cit. on p. 48).

[18] *Common Weakness Enumeration (CWE): A Community-Developed Dictionary of Software Weakness Types*. Tech. rep. Introductory Handout, MITRE Corporation. Bedford, MA, USA: MITRE Corporation. URL: `https://cwe.mitre.org` (cit. on p. 48).

[19] Mathworks. *Formal Methods*. `https://uk.mathworks.com/discovery/formal-methods.html` (cit. on p. 58).

[20] NIST. *Source Code Security Analyzers.* `https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers` (cit. on p. 58).