Politecnico di Torino

Politecnico di Torino

Master's degree in Cybersecurity

A.y. 2024/2025

Graduation Session October 2025

Securing Aircraft Engine Control Units: Utilizing Embedded Board Security Features for Enhanced Protection

Supervisors:

Stefano Di Carlo
Alessandro Savino
Luca Schena

Candidate:

Niccolò Lentini

Abstract

In the aerospace domain, where the concept of airworthiness and ensuring its continuity is fundamental, securing Engine Control Units (ECUs) is a critical objective to prevent system compromise with potentially catastrophic consequences. This work focuses on the implementation of robust security mechanisms for embedded avionics ECUs by leveraging the advanced hardware security features available on one from the NXP S32K family of microcontrollers used in both automotive and aerospace applications.

By examining this recent Automotive General Purpose ECU, the study demonstrates how embedded system security can be significantly enhanced through the strategic utilization of built-in hardware capabilities. Special emphasis in this work is placed on securing the software image loading process with appropriate verification and authentication and enforcing strict memory protection policies to ensure both the integrity and confidentiality of system data and code exploiting the cryptographic capabilities of the board.

The study begins with the modelling of a realistic case study for an avionic ECU, establishing a foundation for subsequent security analyses. A detailed threat assessment follows, adhering to most recent aerospace standards and regulations (e.g. DO-178C, DO-356, AIR7368) and employing frameworks like the Common Attack Pattern Enumeration and Classification (CAPEC) and the Embedded System Threat Modelling and Mitigation Methodology (EMB3D) to systematically identify and categorize potential vulnerabilities. Next, a comprehensive analysis of the necessary security measures is conducted to obtain adequate countermeasures against each threat condition.

The implementation part follows a configuration and deployment strategy for hardware-assisted security mechanisms such as secure boot, cryptographic validation of application image, and memory region protection, features that are not only analyzed in terms of their technical configuration but also evaluated for their effectiveness in mitigating all the threats previously identified.

Experimental results confirm that the proposed approach strengthens the ECU's security reducing the attack surface. The use of on-chip security features contributes to a substantial increase in system resilience without introducing significant performance penalties.

In conclusion, this work provides a concrete contribution toward the development of safer and more secure avionics systems by proposing an architecture that exploits the native hardware security features of modern microcontrollers underscoring the critical role of hardware-assisted security in the design of next-generation ECUs.

Table of Contents

Chapter 1

Avio Aero

This thesis work was carried out during an internship at Avio Aero, which provided me the opportunity to work at their headquarters in Rivalta di Torino throughout the entire duration of the internship.

Figure 1.1: GE Avio Aero logo

1.1 Company overview and background

Avio Aero is a GE Aerospace company that represent a global leader in the design, production, and maintenance of advanced propulsion systems for the aerospace industry. Originally established as Fiat Aviazione in 1908, the company has established itself as a key player in the development of innovative technologies for aviation, covering a significant role in the development of Italy's aviation sector during the early 20th century, contributing to both civil and military aviation programs. Avio Aero is headquartered in Turin but operates multiple facilities across Europe.

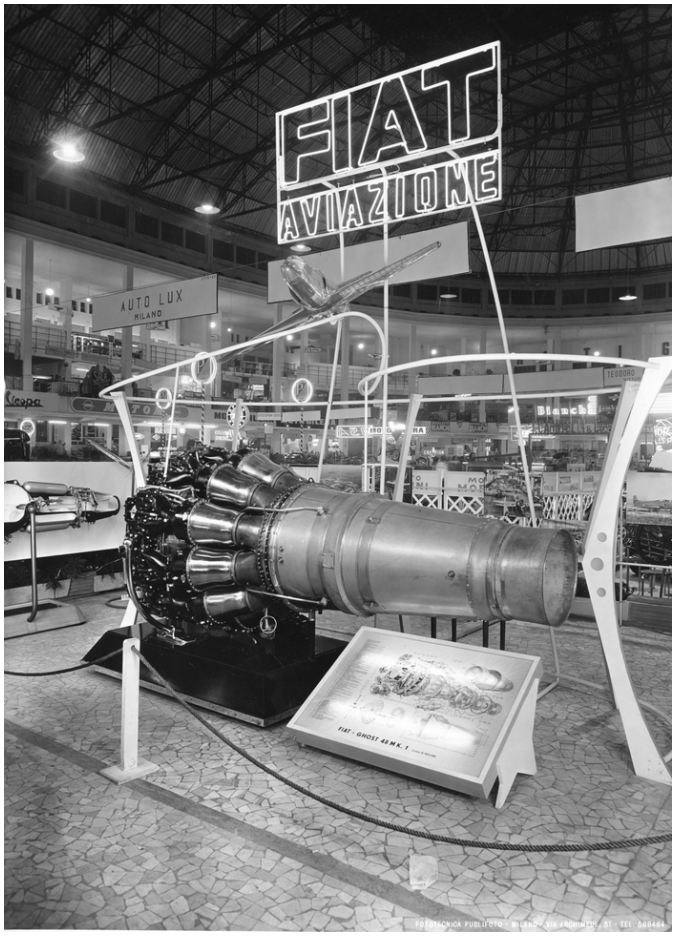


Figure 1.2: Fiat Aviazione stand at Fiera Campionaria di Milano 1952. Source `https://archiviostorico.fondazionefiera.it/oggetti/10412-stand-della-fiat-aviazione-al-salone-dellauto-avio-moto-ciclo-e-accessori-sportivi-nel`

In 1989 the company changed its name in Fiat Avio while in 2003, Fiat Avio underwent a major transformation when it was sold by Fiat Group to a consortium led by the Carlyle Group and Finmeccanica (now Leonardo) and was rebranded as Avio and continued to focus on aerospace propulsion systems, while also expanding into space propulsion technologies.
In 2013, Avio's aerospace division was acquired by General Electric (GE) and became part of GE Aerospace, taking on the name Avio Aero. This acquisition integrated the company into GE's global operations, providing access to GE's resources and expertise in aviation technology allowing Avio Aero to continue develop advanced propulsion systems.
Today, with more than 5,700 employees, Avio Aero is present from the product design and development phase through production and aftermarket services.
The company specializes in the design and manufacturing of critical components for aircraft engines, including turbines, gearboxes, and combustors, as well as advanced systems for both civil and military aviation. Avio Aero is renowned for its expertise in additive manufacturing, leveraging cutting-edge 3D printing technologies to produce complex engine parts with enhanced performance, reduced weight, and improved efficiency. This commitment to innovation aligns with

GE Aerospace's broader mission to deliver safer, more sustainable, and more efficient solutions for the aviation industry.

The company adheres to rigorous standards and certifications to ensure the reliability and airworthiness of its products, contributing to the success of major aerospace programs worldwide. Collaborations with leading aircraft manufacturers and research institutions gave Avio Aero a pivotal role in shaping the future of aviation through continuous technological advancements and a focus on sustainability.

As a trusted partner in the aerospace sector, Avio Aero combines decades of experience with a forward-looking approach to innovation, making it a cornerstone of GE Aerospace's global operations. The dedication to excellence and the ability to adapt to the evolving demands of the industry underscore its position as a leader in propulsion systems and aerospace technology.

1.2 Latest Achievements

During its history the company has achieved several key milestones successfully leading the development o advanced technologies and taking part in international projects.

1.2.1 AMBER Hybrid-Electric Propulsion Project

Avio Aero has launched a hybrid-electric technology demonstration program as part of the European Clean Aviation initiative that focuses on developing a hybrid-electric demonstrator engine that combines traditional gas turbines with electric power systems to improve fuel efficiency and reduce emissions. This effort explores the potential of hybrid-electric propulsion for regional and short-haul aircraft, using advanced technologies like electric motors, power electronics, and energy storage systems. Working in collaboration with European research institutions, universities, and industry partners, Avio Aero is contributing to the aviation industry's push toward sustainability and its goal of achieving net-zero carbon emissions by 2050.

1.2.2 Catalyst engine

The Catalyst engine represents one of Avio Aero's most significant technological achievements in recent years. Developed in collaboration with GE Aerospace and entirely designed and manufactured in Europe, the Catalyst engine is the first turboprop engine of its kind in over five decades on the continent. It integrates advanced technologies such as a fully authority digital engine control (FADEC) system and extensive use of additive manufacturing that enable improved performance, reduced fuel consumption and lower emissions compared to its competitors. In early 2025 the Catalyst Engine received FAA Part 33 certification after a rigorous campaign involving over 2600 hours of testing. The Catalyst engine has also been selected to power the Eurodrone, the European Medium Altitude Long Endurance (MALE) remotely piloted aircraft system, marking a major step toward strategic autonomy in defense application.

These milestones confirm the technical maturity of the engine and demonstrate Avio Aero's capacity to deliver cutting-edge propulsion systems for both civil and military aviation.

Figure 1.3: Catalyst engine. Source `https://avioaero.com/it/media/media-releases/il-catalyst-scelto-da-airbus-per-eurodrone`

Chapter 2

Security Regulations in Aerospace applications

In the aerospace industry, ensuring the safety and security of systems is fundamental, as these systems operate in environments where failure or compromise can have catastrophic consequences. To address these challenges, stringent security regulations and standards have been established to guide the development, certification, and operation of aerospace systems and all the vendors (like Avio Aero) must strictly adhere to these regulations to guarantee the right level of security and safety of the systems.

2.1 Airworthiness & Airworthiness Security

In the process of securing avionic system, two key terms are commonly used, Airworthiness and Airworthiness Security. The first one refers to the ability of an aircraft or its components to operate safely within the defined parameters of its design and certification, ensuring that aircraft meet strict safety standards and are fit for flight. Airworthiness is governed by regulatory authorities such as the Federal Aviation Administration (FAA) in the United States and the European Union Aviation Safety Agency (EASA) in Europe that establish certification processes, maintenance requirements, and operational guidelines to ensure the continued safety of aircraft throughout their lifecycle.
Airworthiness Security, on the other hand, focuses on protecting the airworthiness of an aircraft from intentional threats, such as cyber attacks, sabotage, or other malicious activities that comes from the increasing reliance on software, connectivity and digital systems of a modern aircraft. Airworthiness security ensures that the **confidentiality, integrity and availability (CIA)** of systems essential to safe flight are maintained through cybersecurity standards and certifications, threat assessment and mitigation strategies.

2.2 DO-178C (Software Considerations in Airborne Systems and Equipment Certification)

DO-178C is the primary standard for the development and certification of software used in airborne systems and equipment. It provides a structured framework for ensuring the safety, reliability, and airworthiness of software in aviation and defines objectives for software development processes,

including requirements capture, design, coding, testing, and verification. It introduces the concept of Design Assurance Levels (DALs), which categorize software based on its criticality to flight safety, ranging from Level A (most critical) to Level E (least critical). DO-178C also emphasizes traceability between requirements, design, and testing to ensure comprehensive validation and it includes supplements such as DO-330 (tool qualification), DO-331 (model-based development), DO-332 (object-oriented technology), and DO-333 (formal methods), which address advanced software development techniques. Compliance with DO-178C is essential for obtaining certification from aviation authorities like the FAA and EASA.

Design Assurance Level	Description	Target System Failure Rate	Example System
Level A (Catastrophic)	Failure causes crash, death	$<1 \times 10^{-9}$ chance of failure / flight-hr	Flight controls
Level B (Hazardous)	Failure may cause crash, deaths	$<1 \times 10^{-7}$ chance of failure / flight-hr	Braking systems
Level C (Major)	Failure may cause stress, injuries	$<1 \times 10^{-5}$ chance of failure / flight-hr	Backup Systems
Level D (Minor)	Failure may cause inconvenience	No safety metric	Ground navigation systems
Level E (No Effect)	No safety effect on passengers/crew	No safety metric	Passenger entertainment

Figure 2.1: Design Assurance Levels (DALs). Source `https://eteo.tistory.com/496`.

2.3 DO-326B (Airworthiness Security Process Specification)

DO-326B focuses on the security aspects of airborne systems, addressing the growing concern of Intentional Unauthorized Electronic Interaction (IUEI), defined as human-initiated actions with the potential to affect the aircraft due to unauthorized access, disclosure, use, denial, modification or destruction of electronic information or electronic aircraft system interface. This document is the joint product of two industry committees, the EUROCAE Working Group WG-72 and the RTCA Special Committee SC-216 and it provides a framework for identifying, assessing, and mitigating security risks that could impact the airworthiness of an aircraft. It introduces the concept of Airworthiness Security, which ensures that security vulnerabilities do not compromise the safety and operational integrity of the aircraft. DO-326B outlines processes for threat identification, risk assessment, and the implementation of security controls that address airworthiness security during the aircraft product life cycle, from project initiation until the aircraft Type Certificate.
The purpose of the Airworthiness Security Process (AWSP) is to establish that when the system is subject to IUEI, it will remain in a condition for safe operation. This necessary implies the definition of an acceptability treshold and a complete and correct Risk Assessment. The AWSP

is composed of three major parts:

- Certification Activities: to manage the certification process.
- Security Risk Assessment Related Activities: to evaluate risk based on identified threat scenarios to determine acceptability and to assess the implemented security measures always considering the acceptability of the risk.
- Security Development Related Activites: this is the implementation part of the required security measures.

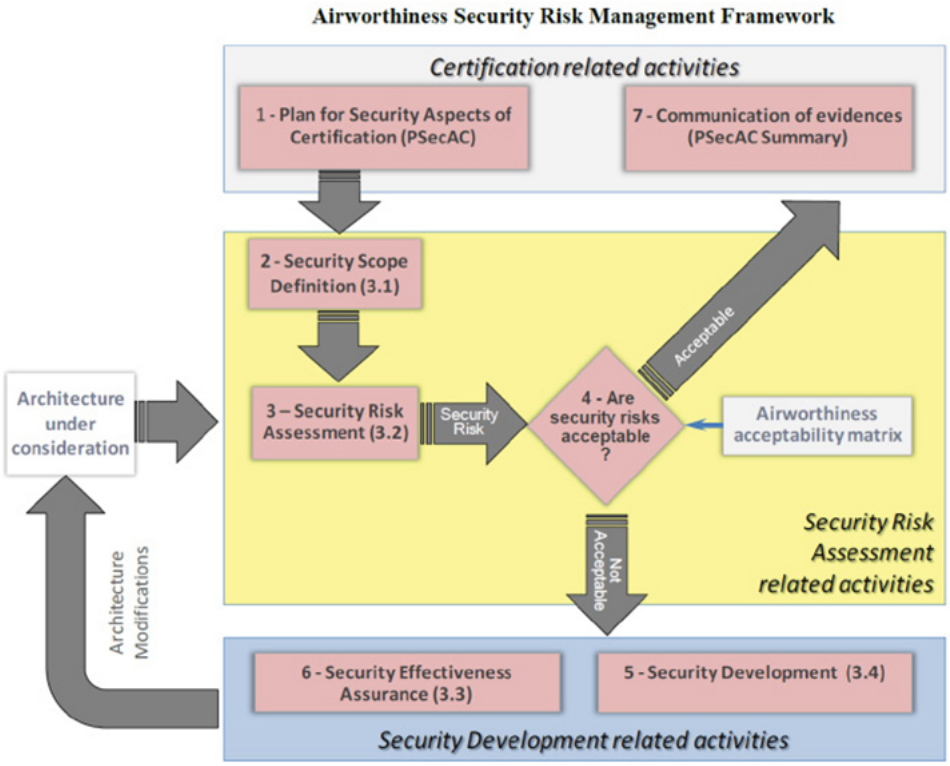


Figure 2.2: DO-326B Management Framework. Source `https://militaryembedded.com/avionics/safety-certification/incorporating-do-326a-security-airworthiness-into-software-development-life-cycle`.

DO-326B provides a structured workflow and defines the activities to be performed, organized into the following steps:

1. Aircraft Security Scope Definition (ASSD): Establish the aircraft's operational environment with respect to information security.
2. Preliminary Aircraft Security Risk Assessment (PASRA): Identify potential threat conditions and scenarios, and evaluate security risks at the aircraft level.

3. System Security Scope Definition (SSSD): Define the system's operational environment concerning information security.
4. Preliminary System Security Risk Assessment (PSSRA): Identify potential threat conditions and scenarios, and evaluate security risks at the system level.
5. System Security Risk Assessment (SSRA): Analyze threat conditions, scenarios, and vulnerabilities to assess the system's security risks.
6. Aircraft Security Risk Assessment (ASRA): Analyze threat conditions, scenarios, and vulnerabilities to assess the aircraft's security risks.

The workflow proposed by DO-326B follows a V-MODEL approach that highlights the presence of a preliminary Security Risk Assessment before the implementation, followed by another Security Risk Assessment after the implementation. The aim of the first one is to identify which part of the system is at an acceptable theoretical security risk level, identifying new security requirements when the security risk is not acceptable.

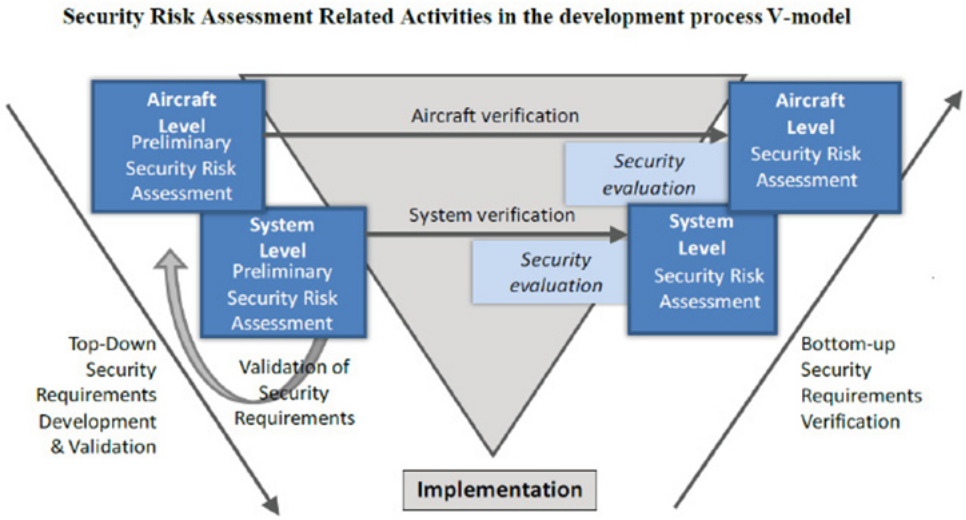


Figure 2.3: DO-326B Security Risk Assessment V-MODEL. Source `https://militaryembedded.com/avionics/safety-certification/incorporating-do-326a-security-airworthiness-into-software-development-life-cycle`.

2.4 DO-356 (Airworthiness Security Methods and Considerations

DO-356 is another critical guidance document, born by the joint work of RTCA Special Committee SC-216 and EUROCAE Working Group WG-72, that addresses the growing need for cybersecurity in aviation systems due to the increasingly interconnected nature and reliance on digital technologies that made the potential for cyber threats broader. DO-356 complements other aviation standards, such as DO-178C and DO-326B, by focusing specifically on the identification, assessment, and mitigation of cybersecurity risks throughout the lifecycle of an aircraft system. It provides methodologies and guidelines to be used within the airworthiness security process defined in DO-326B, for threat modeling, vulnerability analysis, and risk management, ensuring that cybersecurity measures are integrated into the design, development, operation, and maintenance phases. The document also emphasizes the importance of aligning cybersecurity efforts with safety objectives, ensuring that security measures do not inadvertently compromise the reliability or functionality of critical systems. One area for enhancement in DO-356 could involve expanding its guidance on emerging technologies, such as artificial intelligence (AI) and machine learning (ML), which are increasingly integrating into aviation systems and introduce unique cybersecurity challenges, such as adversarial attacks on AI models or data poisoning, which require specialized mitigation strategies. Additionally, DO-356 could benefit from more detailed recommendations on securing communication protocols used in connected aircraft, such as satellite links and ground-based networks, to address vulnerabilities in data transmission.
The document is organized into six chapters that are designed to be used sequentially. Starting with the first chapter, which defines key terms and acronyms that the reader is expected to be familiar with, the subsequent chapters are structured to progressively guide the reader in acquiring the necessary knowledge and methodologies to ensure airworthiness security and conduct effective security risk assessments.
It also contains several appendices that provides concrete examples on how to perform a complete security risk assessment and how to guarantee airworthiness security.

2.5 SAE AIR7368 (Cybersecurity for Propulsion Systems)

The SAE Aerospace Information Report (AIR) 7368 serves as a foundational framework for establishing a common approach to cybersecurity airworthiness certification for propulsion system manufacturers. Developed through a joint effort between industry experts and regulatory members of the E-36 Cybersecurity Subcommittee, its primary purpose is to provide comprehensive guidance on cybersecurity practices. As outlined in the document, achieving compliance requires that the engine control, propeller control, monitoring systems, and all auxiliary equipment systems and networks associated with the propulsion system be safeguarded against IUEI. Such interactions could potentially compromise the safety and functionality of the propulsion system, making robust protection measures essential. This document applies to security protection of propulsion systems for which IUEI are possible when connected to:

- Aircraft Systems like air data computers, flight and trhust management computers, engine interface units.
- Equipment for field and factory loading of production software.
- Equipment for making adjustments to adjustable parameters in the embedded software.

- Equipment to download from or upload data to the propulsion control system using wired or wireless communication protocols.
- Any other systems with digital or network connectivity capability.

In this document is highlighted the distinction between four aviation domains. The Aircraft Control Domain (ACD) that consists of airborne systems and networks which functions are to support safe operation of the airplane. The Airline Information Services Domain (AISD) that enables operation of the airline like electronic manuals, cabin surveillance, cabin crew information access and so on. Passenger Information and Entertainment Services Domain (PIESD) that is a private unstrusted domain that informs and entertains the passengers. Passenger-Owned Devices Domain (PODD) that is the public domain external to the aircraft domain that includes all the devices that passengers may bring on board.
When performing security risk assessment is important to consider the division between these domains and how they can influence each other.

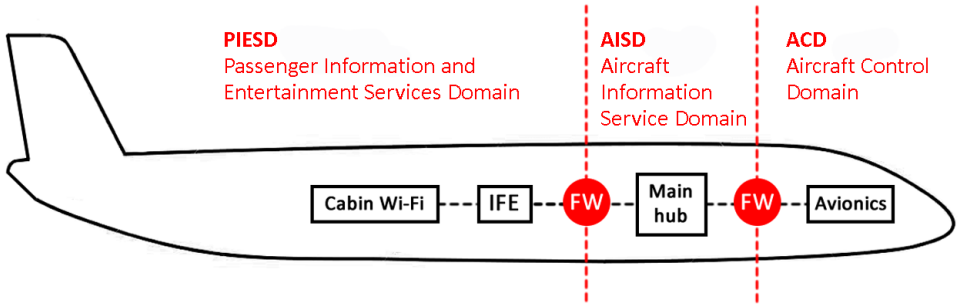


Figure 2.4: Aircraft domains. Source `https://www.pentestpartners.com/security-blog/in-flight-entertainment-system-security/`

Chapter 3

State of the art in avionics product security

Avionic product security encompasses a wide range of components, from the engine's Electronic Control Unit (ECU) and passenger entertainment systems to pilot inputs and mechanical systems. Given that Avio Aero is a leader in engine production, this analysis focuses specifically on the security aspects related to the engine.
The state of the art security measures that can be adapted from other industry sectors focus on safeguarding the confidentiality, integrity and availability (airworthiness security) of systems which increasingly rely on digital systems for control monitoring and diagnostics. For this reason advanced security measures are essential to address potential vulnerabilities.

3.1 Nowadays proposed solutions

The integration of cybersecurity into aircraft engine design is a relatively recent development, driven by the need to protect critical systems from increasingly sophisticated threats. While modern aerospace engines aim to employ advanced security measures, the actual solutions adopted do not always represent the highest level of security. In fact, since Certification Authorities have only recently begun formalizing dedicated regulations (EASA started in 2021), many already certified engines lack security awareness and re-certification requires a significant effort. For example basic mechanisms like cyclic redundancy checks (CRC) are often used to verify data integrity, but they offer limited defense against more complex attacks, such as data manipulation or spoofing. Furthermore the rapid evolution of cyber threats means that even robust systems can become vulnerable without regular updates and improvements.
Although specific implementation details about proprietary engines remain confidential, the general principles and technologies that should be adopted in this sector are known.

3.2 Secure Electronic Control Units (ECUs)

The ECU is the central element of an aircraft engine's digital control system, responsible for managing all the engine functions and this makes it a primary target for cybersecurity measures. Protection applied on the ECU are fundamental for the safety, operational reliability, protection against unauthorized access and information integrity of the system.
Secure Boot represents a pivotal feature to ensure that the ECU only runs software that is verified

and trusted. During the boot process the ECU perform some checks on the software components to confirm the authenticity and integrity, prevent malicious software from being executed.
Cryptographic protection the main technique to safeguard data integrity and confidentiality. Encryption can be used to protect sensitive data while digital signature further ensure that data has not been tampered with, providing a robust mechanism for verifying data authenticity.
Access Control (like role based access control RBAC) is the security measure to prevent unauthorized access to the ECU, ensuring that only authorized personnel can access or modify critical engine parameters.

3.3 Data Communication Security

The communications between the engine, aircraft systems, and external entities such as ground stations are essential for operational efficiency and safety, so it is securing these communications.
Encrypt communications using protocols like TLS is a common solution that ensures that any intercepted data remains untelligible to unatuhorized parties.
Authentication mechanisms are used to verify the identities of communicating parties ensuring that only legitimate systems can exchange data with the engine preventing injection of malicious commands.
Integrity checks using hashing (e.g. SHA Algorithms) or checksum (e.g. CRC) are performed to verify that data has not been altered during transmission.

3.4 Real-Time Monitoring and Intrustion Detection

Solutions like Anomaly Detection or Intrusion Detection Systems (IDS) can monitor the system for signs of malicious activity and to compare it with the expected behavior, enabling rapid response to mitigate threats before they impact the engine performance. Most recent solutions adopt new technologies like AI based Intrusion Detection System to increase the efficiency of these systems.

3.5 Hardware Security Features

Physical security measures complements software-based protections to enhance the engine security.
Trusted Execution Environments (TEEs) provide a secure area within the engine's hardware for executing sensitive operations, such as cryptographic functions, protecting from interference or tampering by unauthorized software.
Tamper-Resistant Components are designed to resist physical tampering, with features such as sealed enclosures and tamper-evident seals, safeguarding the engine from physical attacks.
On-chip Hardware security measures are the most studied solution as they represent the best solutions against common problems like economic constraints, security vs performance tradeoff and compliance with new regulations.

3.6 Software Security

Ensuring the security of engine software is a continuous process that involves multiple layers of protection.
Engine Secure Software Development Lifecycle obtained using secure coding practices, rigorous testing, and compliance with standards making software resilient against known vulnerabilities

and compliant to industry best practices for safety and security.
Regular Updates and Patching for which engine software is regularly updated with security patches to address newly discovered vulnerabilities. These updates are carefully tested to ensure they do not compromise system functionality, maintaining the balance between security and operational reliability.

3.7 Redundancy

By incorporating duplicate systems for critical functions, such as engine control and monitoring, redundancy allows the engine to continue functioning even if one component is compromised or fails. These redundant systems operate independently, providing a backup that can immediately take over in the event of an issue, not only mitigating the impact of potential cyber threats but also enhancing safety by maintaining uninterrupted engine performance during adverse conditions. Full Authority Digital Engine Control (FADEC) due to its full control on the engine operations implement redundancy as a security measure to guarantee the engine availability.

3.8 Is aircraft physical security really necessary?

Although the likelihood of physical attacks on aircraft engines is extremely low, the potential consequences of such an attack could be catastrophic. This underscores the importance of proactively addressing and mitigating possible threats to ensure safety and security.
Even if, for security purposes, there are no publicly known physical attacks on aircraft engines, the following are examples of incidents where intruders gained access to airports and remained for sufficient time to potentially carry out serious attacks on aircraft.

- Avalon Airport, March 2025 [1]: A 17 year old boy managed to bypass airport security and board a plane by disguising himself as a staff member, wearing a security jacket and work belt, while carrying a gun.
- Mumbai Airport, 2019 [2]: A man was detained on Mumbai airport's Runway 27 after being spotted by the pilots of SpiceJet Flight SG634 who alerted airport security. The man managed to climb over the airport wall and enter the aircraft zone.
- Kano Airport, 2024 [3]: A man climbed the airport fence during the night and managed to get on an aircraft without being spotted. The cabin crew found him the next morning when they entered for a new flight.
- London, June 2024 [4]: Two elements of climate activist group Just Stop Oil breached security at a London airport in an attempt to target Taylor Swift's private jet as part of their protest against celebrity carbon emissions. The activists managed to enter the airport and remain there for some time, but ultimately failed to locate the jet.
- Bergamo Airport, July 2025 [5]: A 35-year-old man breached security at Bergamo-Orio al Serio Airport by abandoning his car outside the terminal and running into the aircraft parking area. He accessed the runway by opening a security door and ran toward a jet preparing for takeoff, where he tragically lost his life after being sucked into the engine.

While the incidents described above were not aimed at performing cyberattacks, the ability of intruders to gain unauthorized access to restricted areas and remain there for extended periods demonstrates the potential for serious threats, including cyber or physical attacks on aircraft

systems. Even if these breaches were not malicious in nature, the time and access the intruders had would have been sufficient to execute harmful actions, such as tampering with critical systems or planting devices capable of compromising engine operations. These examples underscore the need for implementing physical security features alongside cybersecurity measures to ensure the safety and resilience of modern aviation systems against a wide range of threats.

Chapter 4

Product Security Assessment

This work's product security assessment (PSA) has been conducted following the guidance provided by the regulations outlined in Chapter 2. In a company like Avio Aero PSAs are initiated to ensure that avionic products (as the one in this work) meet customer product security requirements, adhere to GE product security standards, and comply with applicable regulatory certification requirements.
The PSA described in this chapter has been developed to align with industry-standard guidelines and regulations.

4.1 Use Case

The use case considered in this work focuses on an engine designed for civil aircraft, so it is an highly complex systems that must meet stringent safety, reliability, and performance standards. These engines are responsible for providing thrust to propel the aircraft and maintaining operational stability under various environmental conditions, additionally, they must comply with regulatory requirements such as those set by aviation authorities like the FAA and EASA.
The real case reference is a GE engine designed for civil aviation which integrates advanced digital systems to work, in fact, as modern engines increasingly rely on digital systems, the potential for cyber vulnerabilities grows and without strong security measures, critical systems could be exposed to risks such as unauthorized access, data manipulation, or system disruption. This highlights the need for enhanced cybersecurity protocols to safeguard the engine's digital components and ensure resilience against emerging threats in the aviation industry.

4.2 PSA Model

PSA process follows the guidelines from DO326B and V-model approach and is composed of different phases that include system design and verification activities. Moreover for each phase is expected the production of an artifact that provide evidence of work done in that phase.

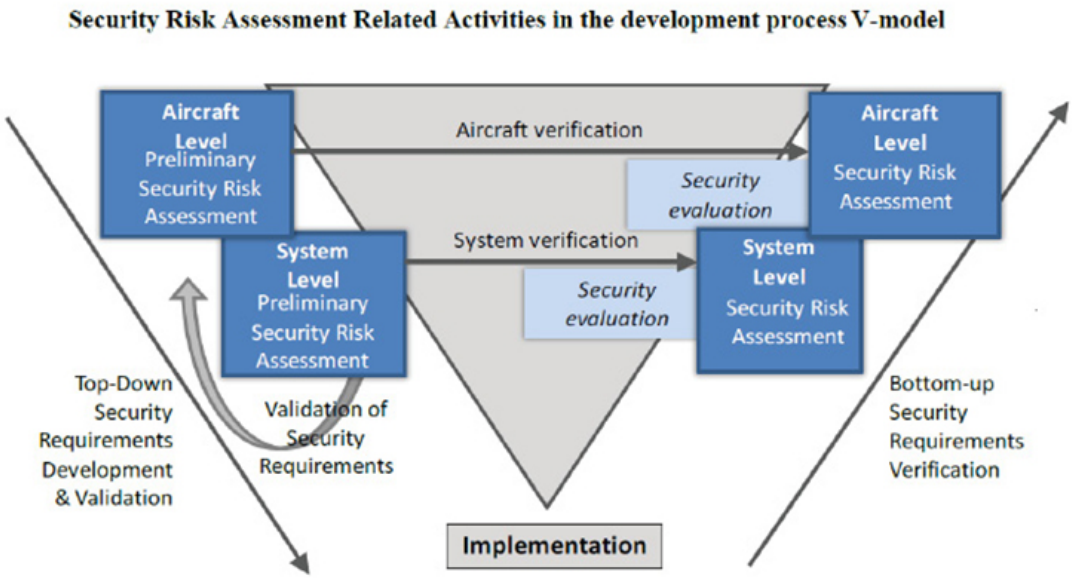


Figure 4.1: DO326 V-model approach

This is a process that aim to identifying, analyzing, and mitigating cybersecurity risks in the system. It is designed to ensure that security considerations are integrated into the system lifecycle from the earliest stages of development thanks to the various phases:

- Planning and scope definition phase: it is the foundation of the Security Assessment process, where the objectives, scope, and boundaries of the cybersecurity effort are defined. This phase begins with identifying the systems, components, and interfaces that require protection (assets). Risk identification is also initiated, focusing on potential vulnerabilities and areas of concern. During this phase it is developed a high-level understanding of the system architecture and its operational environment including identifying external interfaces, data flows, and dependencies that could introduce security risks. This phase sets the stage for a targeted and efficient security assessment.

- Security Assessment phase: it focuses on evaluating the system's current security posture and identifying risks and vulnerabilities conducting a detailed analysis of the system's architecture using techniques such as threat modeling, risk analysis, and vulnerability scanning. The goal of this phase is to prioritize security concerns based on their likelihood and impact, for example risks that could compromise safety-critical systems or sensitive data are given higher priority. The findings from the assessment phase provide the foundation for defining security requirements in the next phase.

- Security Requirements phase: translate the findings from the assessment phase into actionable security measures by defining specific security requirements that address identified risks

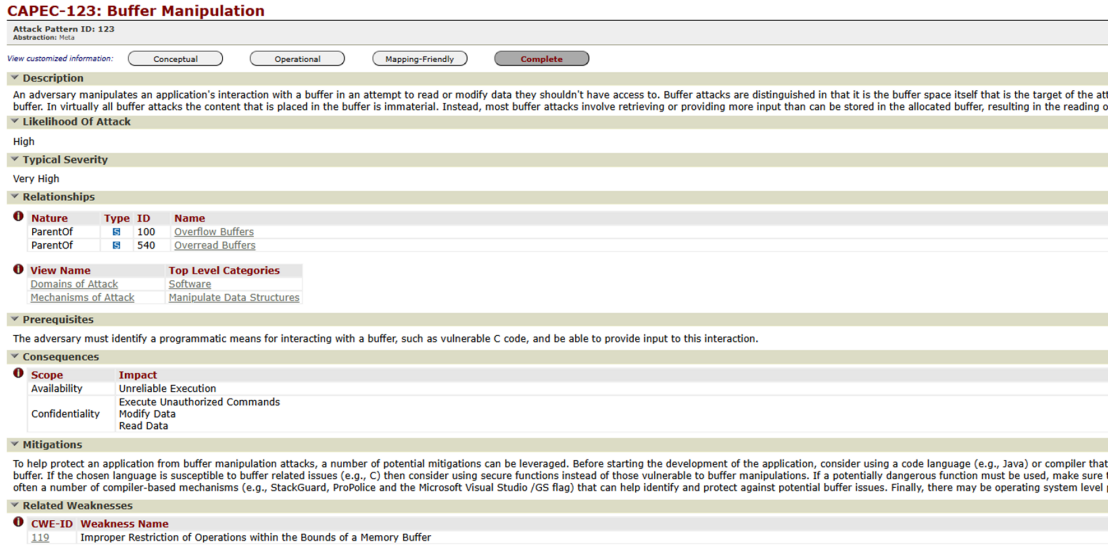
and vulnerabilities. These requirements may include technical measures, such as encryption protocols or access controls, as well as procedural measures, such as incident response plans or training programs. During this phase, the team also develops security specifications and design criteria to ensure that the system meets regulatory standards and industry best practices.

- Security Verification phase: ensure that the implemented security measures meet the defined requirements and effectively mitigate identified risks by developing test plans, executing test procedures, and documenting results. Verification activities may include penetration testing, vulnerability scanning, and functional testing of security features with the goal of validating the system's security posture and ensure compliance with regulatory standards. Any issues identified during testing are addressed through iterative improvements (iterative process).

- Final Reporting phase: it finalizes the Security Assessment process and documents the outcomes by summarizing the security measures implemented, the results of verification, and any residual risks. During this phase, recommendations for ongoing security monitoring and maintenance are developed to ensure that the system remains resilient against emerging threats throughout its lifecycle.

4.3 Threat Repositories: CAPEC & MITRE EMB3D

Threat assessment in this work has been conducted using two widely recognized resources to provide a thorough evaluation of potential cybersecurity risks and vulnerabilities. The first resource, **Common Attack Pattern Enumeration and Classification (CAPEC)**, is a catalog maintained by MITRE that provides a comprehensive repository of common attack patterns, detailing the methods, techniques, and strategies adversaries use to exploit system vulnerabilities. Each attack pattern is described in terms of its characteristics, prerequisites, and potential impacts enabling security professionals to identify weaknesses in system architecture and operational processes, in fact, CAPEC is particularly valuable for mapping attack patterns to specific system features, allowing for the development of targeted mitigation strategies that address the most relevant threats.



CAPEC-123: Buffer Manipulation

Attack Pattern ID: 123
Abstraction: Meta

View customized information: Conceptual | Operational | Mapping-Friendly | Complete

Description

An adversary manipulates an application's interaction with a buffer in an attempt to read or modify data they shouldn't have access to. Buffer attacks are distinguished in that it is the buffer space itself that is the target of the att
buffer. In virtually all buffer attacks the content that is placed in the buffer is immaterial. Instead, most buffer attacks involve retrieving or providing more input than can be stored in the allocated buffer, resulting in the reading o

Likelihood Of Attack

High

Typical Severity

Very High

Relationships

Nature	Type	ID	Name
ParentOf	B	100	Overflow Buffers
ParentOf	B	540	Overread Buffers

View Name	Top Level Categories
Domains of Attack	Software
Mechanisms of Attack	Manipulate Data Structures

Prerequisites

The adversary must identify a programmatic means for interacting with a buffer, such as vulnerable C code, and be able to provide input to this interaction.

Consequences

Scope	Impact
Availability	Unreliable Execution
Confidentiality	Execute Unauthorized Commands Modify Data Read Data

Mitigations

To help protect an application from buffer manipulation attacks, a number of potential mitigations can be leveraged. Before starting the development of the application, consider using a code language (e.g., Java) or compiler that
buffer. If the chosen language is susceptible to buffer related issues (e.g., C) then consider using secure functions instead of those vulnerable to buffer manipulations. If a potentially dangerous function must be used, make sure t
often a number of compiler-based mechanisms (e.g., StackGuard, ProPolice and the Microsoft Visual Studio /GS flag) that can help identify and protect against potential buffer issues. Finally, there may be operating system level

Related Weaknesses

CWE-ID	Weakness Name
119	Improper Restriction of Operations within the Bounds of a Memory Buffer

Figure 4.2: CAPEC Vulnerability Example

The second resource, **MITRE EMB3D**, is a knowledge base of cyber threats created by MITRE to enhance the security of embedded devices focusing on the unique challenges associated specifically with them, which often have constrained resources and critical functions. The framework maps known cyber threats to specific features of embedded devices, such as communication interfaces, memory management, or control mechanisms and for each identified threat, it provides corresponding mitigation strategies, offering actionable guidance to reduce vulnerabilities and improve system resilience. This framework has been used only as support for threat enumeration using CAPEC.

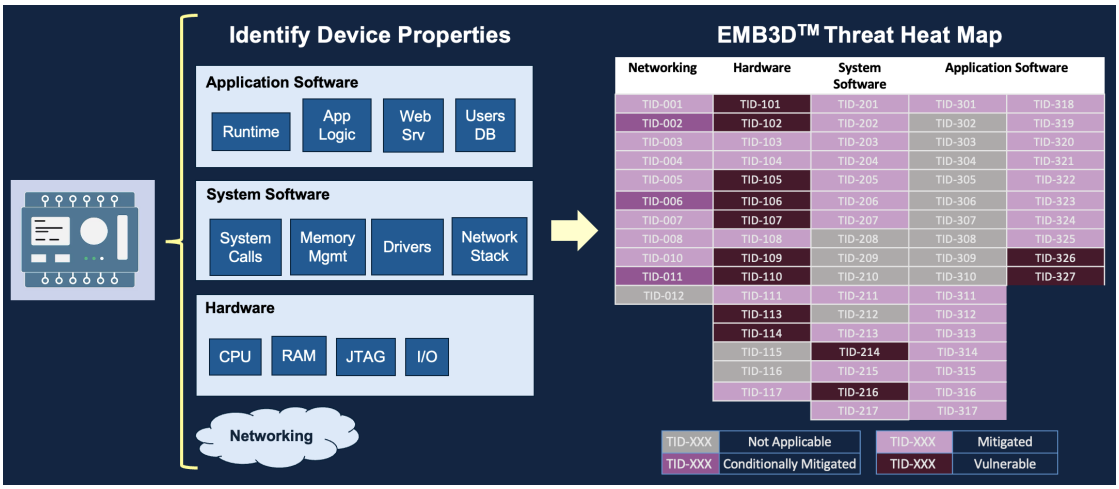


Figure 4.3: EMB3D threat domains. Source `https://emb3d.mitre.org/`

Together, CAPEC and EMB3D provide a robust foundation for threat assessment, combining a broad understanding of attack patterns with a specialized focus on embedded systems. This

dual approach enables the identification of risks in the system and supports the development of effective security measures tailored to the system's unique needs. By utilizing these resources, the threat assessment process can achieve a high level of precision and reliability.

4.4 Scope definition & threat identification

Before conducting threat assessment it is required to define the **Security Environment and System Security Scope** to be analyzed. Since the work is done on the Engine ECS, the domain to be analyzed is the **Aircraft Control Domain (ACD)**, that is the network responsible for the internal communication of the aircraft related to flight and operational control of the aircraft. The **Security Environment** is defined by all the connection that interact with the ECS. These connections are called *Security Vectors* and their presence must be analyzed to spot potential vulnerabilities.
In our study the connection between the ECS and the ADC are represented by:

- ARINC 429: standardized data bus protocol used for communication of avionic information.
- PCAN with the host PC that loads the application image.
- Analog connections: direct analog signal interfaces for specific data or control.

The roles and entities interacting with the aircraft engine:

- Flight Crew: People who fly the plane and have unescorted access to the flight deck with the potential risk to inflict catastrophic loss upon the aircraft.
- Aircraft/Engine Maintainer: People retained by the airframer and subsequent operators who are responsible for the required maintenance operations required by the aricraft with the potential risk to inflict catastrophic loss upon the aircraft and provide unauthorized aircraft access to external parties.
- General Population: People with no responsibility to the engine that can potentially perform attacks if the system is not secured.

After the environment analysis, it is possible to identify possible threats that could harm our system between the ones reported in the threat repositories.

Threat assessment results, reported below, is a list of threat conditions that could potentially harm the system.

CAPEC-ID	Threat Scenario	Threat Category
21	Take advantage of input validation and authentication to attack session IDs and resource IDs to gain access to EEPC.	Bypass authentication mechanism
22	Leverage implicit trust EEPC server places on a client, or what the server believes to be the client.	API/Interface abuse
25	Forced Deadlock.	Race conditions or Deadlocks
26	Manipulate EEPC to run multiple processes concurrently to create a race condition.	Race conditions or Deadlocks

28	Fuzzing, manipulate input validation by feeding random input to ECS.	Manipulate input/parameter/buffer or Inject Traffic
74	Modify state information.	Manipulate state/environment/configuration
94	Adversary in the middle, an attacker can probe the line to place in the middle of a communication and steal information.	Intercept or Eavesdrop
112	Brute forcing, even if adopting adequate solutions, the probability is never zero.	Bypass authentication mechanism
113	An adversary manipulates API to impact the security of the system executing functionality not intended by the API.	API/Interface abuse
114	Exploit flaws in authentication mechanism to gain access to sensitive information and functionalities.	Bypass authentication mechanism
115	Authentication bypass.	Bypass authentication mechanism
116	Excavation, probe the system by using valid interactions but in a wrong way or with wrong arguments to produce errors that leak information.	Manipulate input/parameter/buffer
117	Interception, monitor messages that go from or to the ECS.	Intercept or Eavesdrop
122	Privilege abuse, exploit features of ECS reserved for privileged users.	Privilege Escalation
123	Buffer manipulation, create buffer overflows or similar.	Manipulate input/parameter/buffer
124	Manipulation of resources shared between ECS and other systems.	API/Interface abuse
125	Message flooding, DoS.	Resource attacks or DoS
129	Pointer manipulation.	Manipulate input/parameter/buffer
130	Excessive allocation, force the allocation of excessive EEPC resources via crafted messages.	Resource attacks or DoS
131	Deplete system resources.	Resource attacks or DoS
137	Parameter injection, need proper input validation.	Manipulate input/parameter/bufferor Inject Traffic
148	Content spoofing, adversary modifies content to make it contain something other than what the original content producer intended while keeping the apparent source of the content unchanged.	API/Interface abuse or Spoofing
151	Identity spoofing, adversary may craft messages that appear to come from a different principle or use stolen/spoofed authentication credentials.	Bypass authentication mechanism or Spoofing
153	Input data manipulation, exploiting weakness in input validation.	Manipulate input/parameter/buffer
154	Spoof the location of available resources to leverage an alternate resource.	API/Interface abuse or Spoofing

161	Infrastructure manipulation, manipulate the routing of messages to extract information.	Manipulate state/environment/configuration
165	Modify file contents to cause incorrect processing.	Manipulate state/environment/configuration
169	Footprinting, the attacker uses tools to gather as much information as possible about services and mechanisms of the system.	Footprinting/Fingerprinting
173	Action spoofing, adversary is able to disguise one action for another and trick a user into initiating one type of action when they intend to initiate a different action.	API/Interface abuse or Spoofing
175	Code inclusion, adversary exploits a weakness on the target to force arbitrary code to be retrieved locally or from a remote location and executed.	Execute Arbitrary Code
176	Configuration/environment manipulation, attacker manipulates files or settings external to a target application which affect the behavior of that application.	Manipulate state/environment/configuration
184	Software integrity attack, attacker initiates a series of events to cause a user, program, server, or device to perform actions which undermine the integrity of software code.	Execute Arbitrary Code
188	Reverse engineer the ECS to gain system and security information.	Reverse engineering or Elicitation
212	Functionality misuse, adversary leverages legitimate capabilities to achieve negative impacts.	Manipulate state/environment/configuration
224	Fingerprinting, adversary compares output from a target system to known indicators that uniquely identify specific details about the target.	Footprinting/Fingerprinting
227	Adversary attempts to deny legitimate user access to a resource by continually engaging a specific resource in an attempt to keep the resource tied up as long as possible.	Resource attacks or DoS
233	Privilege escalation by exploiting system weaknesses.	Privilege Escalation
240	Resource injection, change resource identifiers to enable unintended modification of EEPC resources.	Manipulate input/parameter/buffer or Inject Traffic
242	Code injection exploiting improper input validation.	Execute Arbitrary Code / Inject Traffic
248	Command injection, adversary looking to execute a command of their choosing, injects new items into an existing command thus modifying interpretation away from what was intended.	Execute Arbitrary Code / Inject Traffic
272	Protocol manipulation, manipulate the communication protocols through known security vulnerabilities.	Intercept or Eavesdrop
390	bypass physical security.	Physical Unauthorized Access
410	Information elicitation, engage individual from the company to gain information about the system.	Reverse engineering or Elicitation, 10

438	Modification during manufacture, supply chain attack.	Compromise during manufacturing
439	An attacker undermines the integrity of a product, software, or technology at some stage of the distribution channel.	Compromise during manufacturing
440	HW integrity attack, adversary exploits a weakness in the system maintenance process	Compromise during manufacturing
441	Insert malicious logic into the system.	Execute Arbitrary Code or Compromise during manufacturing
507	Physical Theft.	Physical Unauthorized Access
548	Contaminate resource, contaminate ECS information system causing it to handle unauthorized information resulting in unavailability while the problem is resolved.	Resource attacks or DoS
549	Local execution of code, adversary installs and executes malicious code on the target system in an effort to achieve a negative technical impact.	Execute Arbitrary Code
560	Adversary guesses or obtains legitimate credentials to achieve authentication and perform authorized actions under the guise of an authenticated user or service.	Bypass authentication mechanism
594	Traffic injection, adversary injects traffic to degrade or disrupt connection.	Intercept or Eavesdrop
607	Obstruction.	Resource attacks or DoS
624	HW Fault Injection.	Fault injection or Physical signal manipulation

Table 4.1: Mapping of identified threat conditions to CAPEC entries

All the threat conditions have been grouped into five different categories based on the domain of the possible attacks with all of them pointing to a "Top-Event" which is the compromise of the ECU Confidentiality, Integrity or Availability as shown in the figure below.

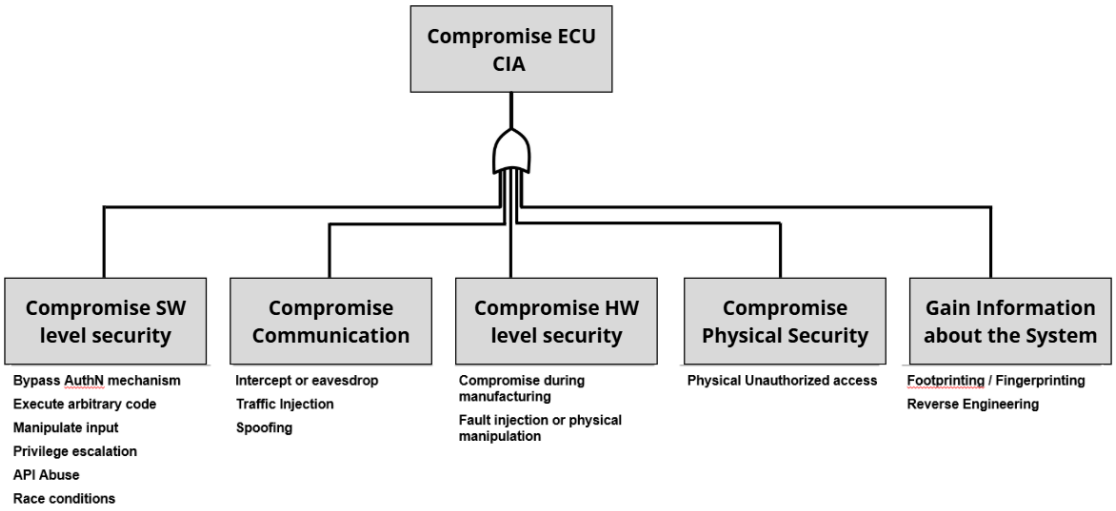


Figure 4.4: Threat condition categories

4.5 High Level Requirements Definition

Each threat condition shapes the creation of High-Level Requirements, each of which defines the necessary system characteristics and safeguards to mitigate the corresponding threat.
After having analyzed and grouped the threat conditions, the resulting High Level Requirements are the following:

Requirement ID	Description	CAPEC Threat ID
R1	The product shall be developed in a secure environment to protect intellectual property and prevent unauthorized access to design data.	404, 410, 416
R2	The product shall validate data communications in accordance with the interface control document (ICD).	116, 130, 227, 490, 137, 175, 242, 248, 624, 126, 148, 151, 184, 216, 28, 21, 114, 123, 153, 161, 272, 548, 607, 441
R3	Unused I/O interfaces, including reserved and manufacturing interfaces, shall be disabled.	116, 117, 169, 224, 125, 130, 131, 227, 490, 137, 175, 240, 242, 248, 624, 126, 148, 151, 154, 173, 184, 440, 25, 26, 29, 74, 462, 113, 212, 216, 554, 28, 39, 112, 155, 21, 114, 115, 22, 69, 122, 233, 234, 30, 123, 124, 128, 129, 153, 161, 165, 171, 176, 268, 271, 272, 548, 607, 97, 542, 438, 439, 441

R4	Clearly defined access controls shall be required for untrusted humans and automated access requests.	126, 173, 74, 28, 112, 155, 21, 114, 115, 122, 233, 234, 30, 129, 390, 507
R5	Write protection shall be provided for executables and images to ensure integrity.	137, 175, 240, 242, 248, 184, 155, 129, 165, 171, 542, 441
R6	Data segment execution protection shall be implemented to prevent unauthorized code execution.	137, 175, 240, 242, 248, 184, 74, 28, 124, 129, 153, 165, 171, 542, 441
R7	Shared system resources shall be protected to prevent unauthorized or unintended information transfer.	126, 148, 151, 154, 173, 74, 161, 410
R8	The integrity of items executed during operation shall be confirmed, and modification shall be prevented.	175, 240, 28, 124, 153, 165, 171, 268, 548, 542
R9	Improper configuration during startup or operation shall be recorded, and normal operations shall be limited or prevented.	175, 126, 113, 212, 155, 176, 268, 272, 548, 542, 441
R10	Data loads and their sources crossing trust boundaries shall be validated and authenticated prior to activation.	116, 125, 130, 227, 490, 137, 175, 240, 242, 126, 148, 151, 154, 173, 28, 28, 112, 114, 124, 128, 153, 171, 548, 542, 441
R11	Test/debug interfaces, ports, protocols, and facilities shall be protected against unauthorized usage.	116, 117, 169, 224, 125, 130, 131, 227, 490, 137, 175, 240, 242, 248, 126, 148, 151, 154, 173, 184, 25, 26, 74, 462, 113, 216, 554, 28, 28, 112, 155, 21, 114, 115, 22, 69, 122, 233, 234, 30, 123, 124, 128, 129, 153, 161, 165, 171, 176, 268, 271, 272, 548, 607, 97, 542, 441
R12	Third-party components shall be analyzed for vulnerabilities and malware.	624, 440, 212, 554, 115, 124, 542, 441
R13	Third-party components included in the product shall be tested to ensure they perform only intended functionalities.	624, 440, 212, 554, 115, 124, 542, 441
R14	Resource usage shall be controlled appropriately for the product.	130, 131, 227, 490, 240, 154, 74, 21
R15	A documented approach shall be implemented to detect, mitigate, and recover from flooding DoS attacks.	125, 490, 607
R16	Users or processes acting on behalf of users shall be uniquely identified and authenticated.	22
R17	The product shall include a vulnerability applicability matrix to identify known vulnerabilities.	All
R18	The vulnerability applicability matrix shall be updated within the last three months.	All

R19	Each product version shall be tested for applicable vulnerabilities prior to release.	116
R20	Software within the product shall comply with secure coding practices.	137, 242, 248, 126, 184, 25, 26, 74, 28, 113, 554, 28, 21, 114, 115, 69, 233, 123, 124, 129, 153, 171, 542, 441
R21	All network interfaces shall be scanned and passively monitored for unexpected services or events.	169, 125, 131, 227, 490, 184, 462, 216, 22, 161, 272, 607
R22	Services responding on network interfaces shall be verified to be necessary per requirements.	169, 125, 131, 227, 490, 184, 462, 216, 22, 161, 272, 607
R23	Interface responses shall provide necessary identification information for the overall product version.	116, 224, 28, 97
R24	Identification information from autonomous responses shall be verified.	116, 224, 28, 97
R25	The operational load image shall be validated prior to execution.	542
R26	The integrity of the load image shall be maintained during transfer from the originating source.	542
R27	Validation mechanisms shall minimize the likelihood of operational load image modification.	542
R28	The operational load image shall include metadata containing version, integrity, and security information.	542
R29	Integrity measures shall be sufficiently complex to uniquely validate the operational load image.	542
R30	Invalid operational load images shall be rejected and not stored in non-volatile memory.	542
R31	The product shall determine the operational load image is invalid if the integrity check fails.	542
R32	Error handling, logging, and recovery mechanisms shall be implemented and clearly defined.	175, 624, 440, 25, 212, 28, 123, 607, 542, 441
R33	All data from outside the security perimeter shall be validated prior to transmission and use.	125, 130, 490, 137, 175, 242, 248, 126, 28, 21, 123, 128, 153, 441
R34	Input that fails validation shall be immediately deleted.	137, 175, 242, 248, 126, 28, 21, 153, 441
R35	Privilege levels shall remain fixed once established.	74, 69, 233, 234, 30
R36	Credential storage and handling shall prevent theft or leakage, and critical operations shall require re-authentication.	112, 114, 115, 560
R37	Users shall be authenticated to enable remote access to files, resources, and log data.	22
R38	The supply chain shall be secured through authentication, validation, and tamper-evident mechanisms.	438, 439, 440

R39	Secure communication protocols shall be implemented to prevent MITM attacks.	94, 216, 272
R40	Resource inputs shall be protected from tampering to prevent unintended future behavior.	548

Table 4.2: System High Level Requirements.

The results obtained could be used in a real case scenario to create all the artifacts of the Planning & Scope Phase of the PSA process.

4.6 Starting Project Assessment Phase

> *Note: From this phase onward, the work will focus on what was possible to address during the internship experience. Specifically, it will include an initial brief analysis of the starting project (NXP "Unified Bootloader" with PCAN-UDS communication), examining the security measures already in place and identifying those that need to be implemented to enhance the system's security. Extensive information on the starting project can be found in its documentation at [6].*

The assessment phase has been conducted on the starting project (NXP Unified Bootloader [6]) considering the security measures implemented to protect the application image loading process. The initial project doesn't include any advanced security measure to protect the loading process, the only protection is the computation of a CRC32 over the image to be checked at every boot to protect the integrity of it. But the CRC32 is not an advanced security feature and doesn't provide an adequate level of protection for this operation. The operation flow of the unified bootloader is the following:

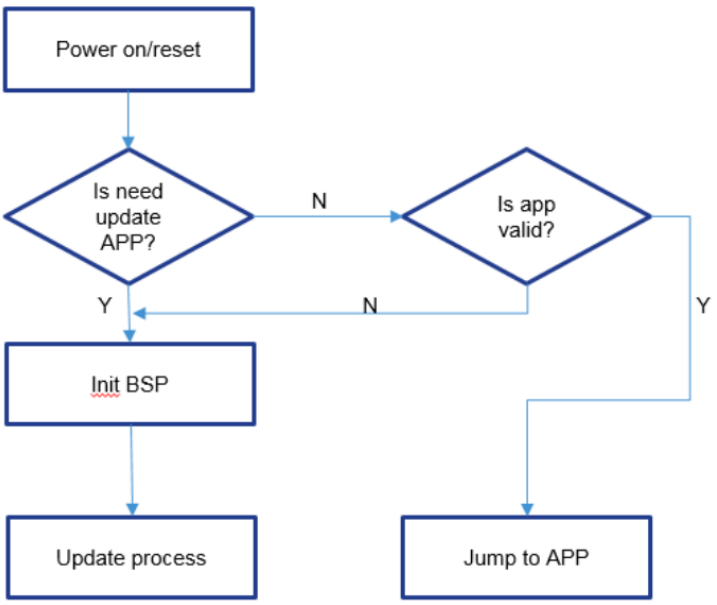


Figure 4.5: Unified Bootloader operation flow

There are two events that control the occurrence of the booting procedure:

1. Booting request from the Host PC via CAN interface.
2. APP Validity Flag set to zero (CRC32 check failure).

If the Validity Flag is set to one and the Host PC doesn't request booting procedure to be executed, then the application is loaded and run.

Given the operation flow of the unified bootloader, it is easy to notice that the process presents several vulnerabilities. The one on which the work focuses the most is the fact that as long as the CRC32 is correctly computed, anyone can load any application image on the board memory and execute it.
The aim of this work is to secure the application image loading process by exploiting the advanced security capabilities offered by a board from the NXP S32K family.

Considering this restricted case, a shorter thread condition identification could be the following one:

Threat Condition	**CAPEC Description**	**CAPEC ID**
Lack of authentication of the application image source	Authentication Bypass, exploiting absence of identity validation to gain unauthorized access	115
No cryptographic integrity check of the application image	Data Integrity Attack, modifying data without cryptographic validation	55
No code signing or digital signature verification	Supply Chain Compromise, injecting malicious software during code update	137
Unrestricted access to bootloader via CAN	Message Injection into CAN Bus, sending unauthorized commands via an exposed interface	240, 131
No secure boot or immutable root of trust	Exploitation of Incorrectly Configured Security Mechanism, exploiting lack of secure boot or memory protection	17
No rollback protection or version control	Exploitation through Downgrade Attack, forcing an older vulnerable version to bypass protections	132
Validity flag manipulation vulnerability	Manipulating Flags or Status Indicators, altering critical boot flags to control system behavior	141
No access control policy enforcement	Privilege Abuse / Insufficient Authorization Checks, issuing commands without privilege separation	18
No runtime memory/code region protection	Buffer Overflow / Memory Corruption, exploiting writable or executable memory without runtime checks	100

Threat Condition	**CAPEC Description**	**CAPEC ID**
No event logging or tamper detection	Inhibition of Audit, preventing detection of malicious activity	574

Table 4.3: Mapping of identified threat conditions to CAPEC entries (specific bootloader use case).

In the next chapters are explained all the solutions adopted to face these threats and secure the bootloading process.

Chapter 5

Theory Background

In this chapter, the theory and working principles of the solution implemented during the internship practical activities are analyzed.

5.1 Hash Functions

A **cryptographic hash function** is a deterministic algorithm that takes an input (or message) of arbitrary length and produces a fixed-length output, commonly referred to as the **digest** or **hash value**. A secure hash function must satisfy the following fundamental requirements:

- **Pre-image resistance:** Given a hash value h, it should be computationally infeasible to find any input x such that $\text{Hash}(x) = h$.
- **Second pre-image resistance:** Given an input x_1, it should be infeasible to find another input $x_2 \neq x_1$ such that $\text{Hash}(x_2) = \text{Hash}(x_1)$.
- **Collision resistance:** It should be infeasible to find any two distinct inputs $x_1 \neq x_2$ such that $\text{Hash}(x_1) = \text{Hash}(x_2)$.

These properties ensure that hash functions are suitable for verifying the integrity of information, enabling secure message authentication, and supporting the generation of digital signatures. A critical aspect of secure hash functions is the **avalanche effect**, whereby a slight change in input leads to a significantly different output, making the hash unpredictable and sensitive to tampering.

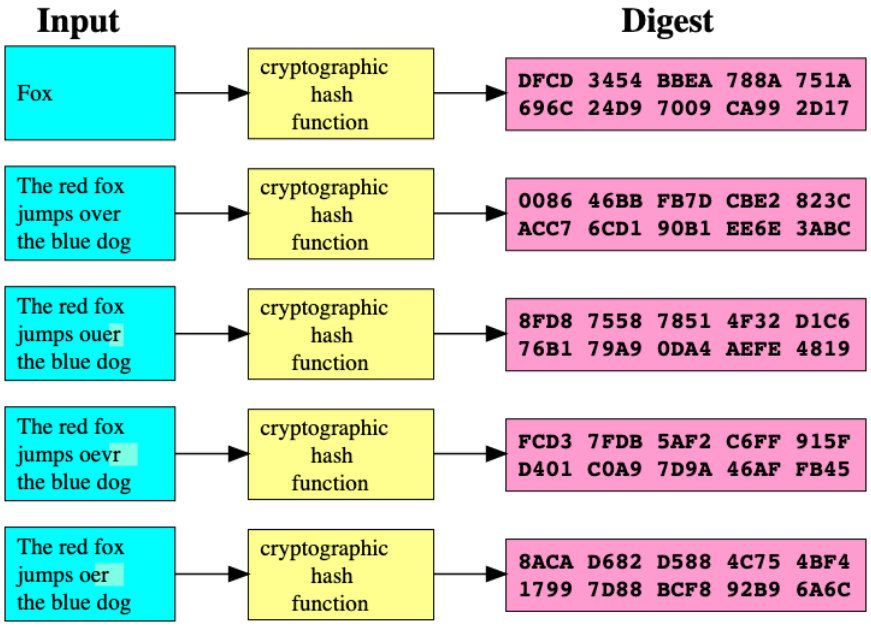


Figure 5.1: Hash function avalanche effect. Source `https://en.wikipedia.org/wiki/Cryptographic_hash_function`

Widely adopted algorithms such as SHA-256 (part of the SHA-2 family) are standardized by NIST and offer a high level of resistance against known cyber attacks, making them suitable for embedded security applications. In this thesis, hash functions are used in conjunction with digital signature algorithms, such as ECDSA, to ensure data authenticity and integrity during secure boot and firmware validation processes.

5.2 Digital Signatures

A digital signature is a cryptographic method that plays a key role in securing digital data and communications, providing a reliable way to verify the authenticity of the sender, ensure that the data has not been altered, and prevent the signer from denying their involvement (properties known respectively as authenticity, integrity, and non-repudiation).
Digital signatures are based on asymmetric cryptography, which uses a pair of mathematically linked keys, a private key and a public key, where the private key is kept secret by the signer, while the public key is openly shared and available to anyone who needs to verify a signature.
The signing process begins with generating a cryptographic hash of the original data by applying a secure hash function (e.g SHA-256), in this way the hash acts as a unique fingerprint of the data, meaning that any slight change in the original content would produce a completely different hash. The hash is then encrypted with the signer's private key, creating the digital signature and the signature is attached to or sent along with the original data.
To verify the signature, the recipient uses the sender's public key to decrypt the digital signature and recover the original hash and at the same time, it computes a new hash from the received data using the same hash function. If both hashes match, the signature is valid, confirming that:

1. The data has remained unchanged during transmission (integrity).

2. The signature was produced by the holder of the private key corresponding to the public key used for verification (authenticity).

3. The signer cannot deny having signed the data, as only the private key holder could have generated the signature (non-repudiation).

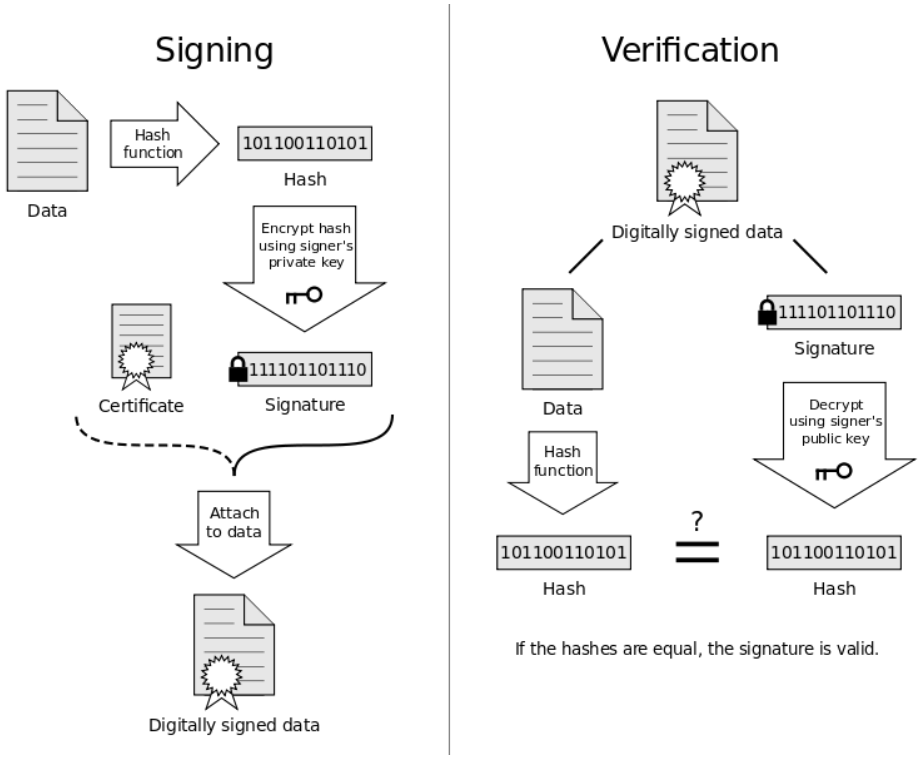


Figure 5.2: Digital Signature flow. Source `https://blog.mailfence.com/how-do-digital-signatures-work/`

Asymmetric cryptography relies on the difficulty of solving complex mathematical problems, such as factoring very large numbers or computing discrete logarithms over elliptic curves, to guarantee the security of the signature process. This inherent computational hardness ensures that deriving the private key from its corresponding public key is practically infeasible, thus providing a strong foundation for trust in digital communications.
Digital signatures, which are a fundamental component of Public Key Infrastructure (PKI), serve the critical function of signing digital certificates, thereby enabling secure and authenticated communication across vast and often untrusted networks like the Internet. By combining the unique properties of cryptographic hash functions, which produce fixed-length fingerprints of arbitrary data, with the asymmetric encryption capabilities of public and private key pairs, digital signatures establish a powerful and reliable mechanism that underpins the security of digital interactions in a broad spectrum of applications, ranging from secure email exchanges and software distribution to electronic contracts and beyond.
Digital certificates play a crucial role within Public Key Infrastructure by binding a public key to the verified identity of an entity, such as an individual, organization, or device. Without this

trusted association, digital signatures alone cannot fully guarantee non-repudiation, since there would be no reliable way to confirm who owns a given public key. Acting as electronic credentials, certificates enable recipients to trust that the public key they use for verification truly belongs to the claimed signer, thereby preventing impersonation and man-in-the-middle attacks. The process of issuing a digital certificate starts when the entity generates a key pair and submits the public key along with identity information to a trusted Certificate Authority (CA) through a Certificate Signing Request (CSR). The CA verifies the requester's identity and, upon approval, digitally signs the certificate with its own private key, effectively endorsing the link between the public key and the entity's identity.
Managing digital certificates involves their issuance, distribution, renewal, and revocation. Certificates have defined lifetimes and must be renewed to remain valid, while compromised or invalid certificates are revoked using mechanisms like Certificate Revocation Lists (CRLs) or the Online Certificate Status Protocol (OCSP). This continuous lifecycle ensures that digital certificates maintain the trustworthiness required for secure and authenticated digital communications, making them indispensable for establishing non-repudiation and overall system security.

5.2.1 Elliptic Curve Digital Signature Algorithm (ECDSA)

Elliptic Curve Cryptography (ECC) is a modern approach to public-key cryptography that uses the mathematical properties of elliptic curves defined over finite fields. Unlike traditional methods such as RSA, which rely on the difficulty of factoring large numbers, ECC bases its security on a problem called the Elliptic Curve Discrete Logarithm Problem (ECDLP), that is much harder to solve for the same key size, with respect to RSA, allowing ECC to provide strong security using much smaller keys. As a result, ECC offers faster computations, requires less memory, and uses less power, making it especially suitable for devices with limited resources, such as embedded systems, IoT devices, and mobile hardware.
Building on ECC, the Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely used algorithm for creating and verifying digital signatures. It adapts the classic Digital Signature Algorithm (DSA) to work with elliptic curves, offering the same security guarantees but with much smaller key sizes. This reduction in key size leads to better performance and lower storage needs, which is particularly important for embedded applications like the one addressed in this thesis, where efficiency and limited computational power are critical.
ECDSA operates through three main steps:

1. **Key Generation:** The private key in ECDSA is a randomly chosen number within a specific range. The public key is then calculated by multiplying this private key by a fixed point on the elliptic curve, known as the generator point GG. An **elliptic curve** is a set of points (x, y) satisfying a specific cubic equation of the form:

   $$y^2 = x^3 + ax + by^2 = x^3 + ax + b$$

   defined over a finite field, where aa and bb are constants that satisfy the condition $4a^3 + 27b^2 \neq 0$ to ensure that the curve has no singularities. These curves and their parameters, such as the well-known NIST P-256, are standardized to ensure both security and compatibility.

2. **Signature Generation:**
   To sign a message, the signer first creates a hash of the message using a secure hash function like SHA-256. Then, a random number k is selected, and the point

   $$R = k \cdot G$$

is computed. The x-coordinate of R modulo the curve's order gives the first part of the signature, called r. The second part, s, is calculated using the private key, the hash, and r in the formula:

$$s = k^{-1} \cdot (H(m) + r \cdot d) \mod n$$

where d is the private key, $H(m)$ is the message hash, and n is the order of the curve. The signature is the pair (r, s).

3. **Signature Verification:**
   When someone receives a signed message, they compute the hash of the message and then calculate two numbers:

$$u_1 = H(m) \cdot s^{-1} \mod n$$

$$u_2 = r \cdot s^{-1} \mod n$$

   Using the public key Q, they then calculate:

$$R' = u_1 \cdot G + u_2 \cdot Q$$

   If the x-coordinate of R' modulo n equals r, the signature is valid, confirming that the message was indeed signed by the private key holder and has not been altered.

Advantages of ECDSA

ECDSA offers several important benefits:

- **Smaller Key Sizes:** ECDSA can provide the same security as RSA or DSA but with much smaller keys, for example a 256-bit ECDSA key matches the security of a 3072-bit RSA key making it ideal for systems with limited storage and processing power, such as embedded devices or IoT hardware.
- **Efficiency:** Because of the smaller keys and optimized math operations, ECDSA requires less computation for generating keys, signing, and verifying signatures leading to faster processing times and lower power consumption, which is especially important in battery-powered or real-time systems. Smaller keys and signatures also reduce the bandwidth needed when transmitting data.
- **Strong Security:** The difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP) represents ECDSA's security. This problem is considered extremely hard to solve with current technology, providing strong resistance against many known cryptographic attacks.

Challenges of ECDSA

However, using ECDSA also comes with some challenges:

- **Random Number Generation:** The security of ECDSA depends heavily on the randomness of the number k used during signature creation. If k is reused or predictable, it can expose the private key, compromising the whole system.
- **Implementation Complexity:** ECC and ECDSA require careful and secure implementation that exploit solutions like robust coding and hardware protections, to avoid vulnerabilities such as side-channel attacks that exploit timing or power consumption to reveal secret keys.

- **Standards and Compatibility:** To ensure security and interoperability, it is important to use well established elliptic curve parameters, such as those standardized by NIST. Using non-standard or weak curves could introduce security risks.

5.3 Secure Boot

Secure Boot is a security mechanism that ensures only authenticated and trusted software is allowed to execute during the boot process of a system. It is a fundamental component of a secure embedded architecture, particularly in systems where software integrity and authenticity are critical, such as in automotive, aerospace, and IoT devices. The core principle of Secure Boot relies on cryptographic validation of the firmware image before execution that is typically achieved through the use of digital signatures, that the bootloader or a hardware-based root of trust verifies against a trusted public key stored securely in read-only memory or fuses.

The Secure Boot process begins with an immutable first stage of code, often referred to as the *Root of Trust*, which is inherently trusted and responsible for verifying the next stage of the boot chain. Each stage of the boot process verifies the integrity and authenticity of the following stage, forming a *chain of trust* that extends up to the operating system or application firmware. If any component in this chain fails validation, the boot process is aborted or redirected to a safe recovery mode.

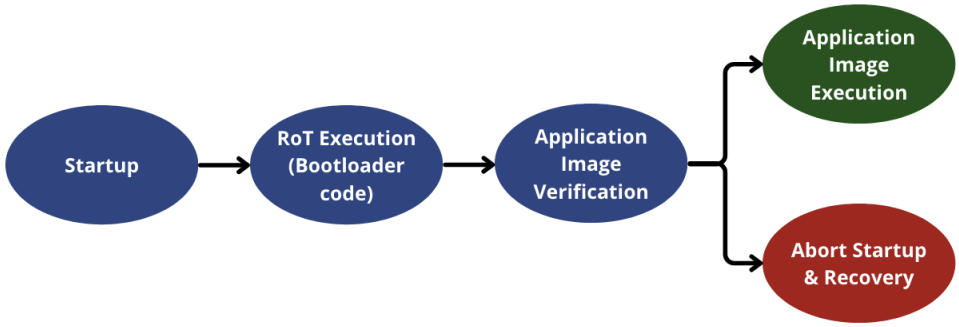


Figure 5.3: Secure boot operation flow

By enforcing that only signed and verified code can be executed, Secure Boot mitigates threats such as unauthorized firmware updates, malware injection during startup, and rootkits. Furthermore, when combined with version control and anti-rollback protections, it can also prevent downgrade attacks where attackers attempt to load older, vulnerable firmware versions.

In this work, the firmware authentication process is based on the verification of an ECDSA digital signature and the procedure involves two main phases: during the signing phase, the firmware supplier signs the application firmware using its private key, subsequently, when the firmware is transmitted to the embedded device, the signature is verified by the board itself, which securely stores the corresponding public key in protected memory. This mechanism ensures the authenticity and integrity of the firmware before it is executed on the device.

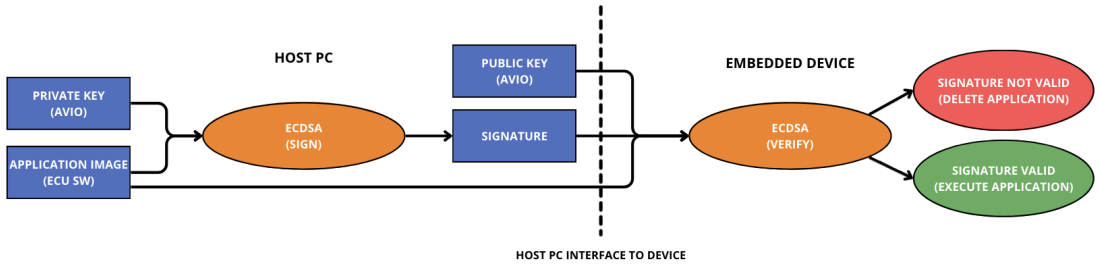


Figure 5.4: Secure boot signature verification

If the signature verification is successful, the application is flagged as valid and subsequently executed by the system, otherwise if the verification fails, the loading process is aborted, and a valid signed application is required to proceed with execution.

5.4 Controller Area Network (CAN)

The Controller Area Network (CAN), standardized as ISO 11898 [7], is a robust serial communication protocol originally developed by Bosch for real-time communication among Electronic Control Units (ECUs) in automotive systems. Its design supports high reliability, fault tolerance, and efficient message arbitration without requiring a central host computer, making it a widely adopted solution in embedded and automotive networks.
CAN follows a multi-master, broadcast-based communication model where each node can transmit messages on the bus and all nodes receive them. Each message contains an identifier that also serves as its priority: lower numerical values represent higher priority, and the arbitration process ensures that the highest-priority message is transmitted without collisions.
The standard CAN frame supports a **maximum payload size of 8 bytes**, which is sufficient for basic control signals but inadequate for transmitting larger data structures such as firmware images or diagnostic logs. To address this, higher-layer protocols such as CAN TP (ISO 15765-2 [8]) are used to segment and reassemble large messages over CAN.
Despite its reliability and efficiency, **CAN lacks built-in security features** such as encryption, authentication, or message integrity and this represents a risk since all nodes on the bus are able to read and potentially inject messages, making CAN susceptible to attacks like spoofing, replay, or DoS. These limitations make it essential to design secure architectures, especially when CAN is used in safety and security critical systems such as bootloaders or over-the-air updates.

CAN Framing

The framing structure of a CAN message plays a crucial role in ensuring reliable and deterministic communication across the bus. A standard CAN frame, defined in ISO 11898-1, consists of several distinct fields: the *Start of Frame (SOF)*, an *Arbitration Field* (composed of an 11-bit Identifier in standard format, or 29-bit in extended format), the *Control Field* (indicating the data length), the *Data Field* (carrying up to 8 bytes of payload), the *CRC Field* (for error detection), an *ACK Field*, and the *End of Frame (EOF)*.
The arbitration mechanism uses the Identifier field to determine message priority, where lower binary values represent higher priority. During transmission, if multiple nodes initiate communication simultaneously, the arbitration logic ensures that only the highest-priority frame proceeds

without collision, while others back off and retry later making this behavior essential for real-time applications where timing constraints are critical.

The *Cyclic Redundancy Check (CRC)* field provides basic error detection by allowing receivers to validate the integrity of each frame, however, no message authentication or encryption is provided at the frame level, which leaves the CAN bus vulnerable to attacks such as message injection or spoofing. In the context of security, the open nature of the frame structure allows an attacker with access to the physical bus to craft malicious frames that appear legitimate to other nodes, highlighting the importance of incorporating higher-layer cryptographic protections in secure system design.

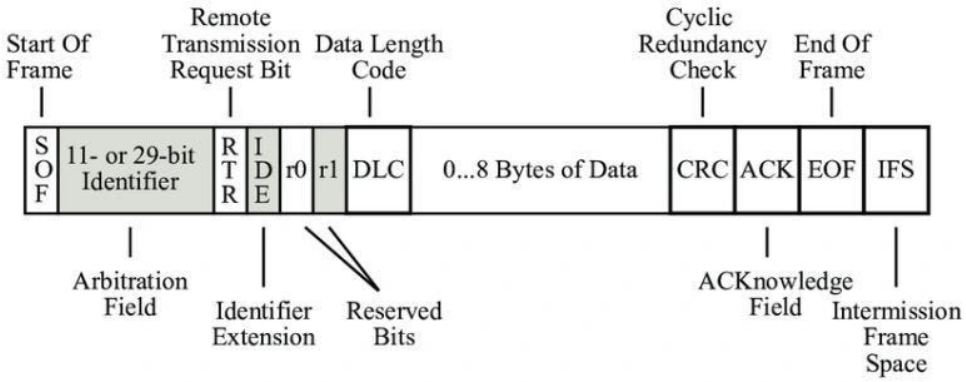


Figure 5.5: CAN frame structure. Source: `https://medium.com/@sjindhirapooja/can-standard-data-frame-format-846b8f9fc749`

5.5 CAN TP

The CAN Transport Protocol (ISO 15765-2 [8]), commonly referred to as *CAN TP*, is a communication protocol designed to facilitate the transmission of data packets larger than 8 bytes over the CAN bus, since standard CAN frames are limited to a maximum payload of 8 bytes. CAN TP provides a mechanism to segment and reassemble longer messages, enabling the reliable transfer of larger data structures such as diagnostic messages, firmware images, or configuration data between ECUs.

The protocol defines four types of frames: *Single Frame*, used for payloads up to 7 bytes, *First Frame*, which initiates the transmission of a multi-frame message, *Consecutive Frame*, used for continuing the segmented data and *Flow Control Frame*, which is used by the receiver to control the flow of incoming data and avoid buffer overflows. CAN TP includes mechanisms for flow control, sequence numbering, and timeout handling to ensure data integrity and correct reassembly on the receiver side.

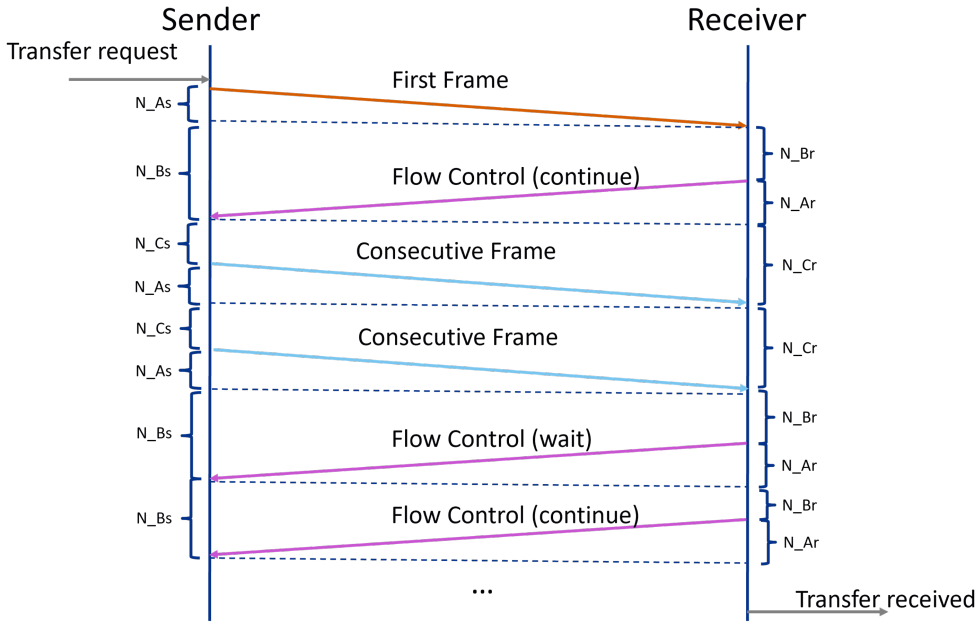


Figure 5.6: Example CAN-TP message over a CAN bus. Source: `https://onlinedocs.microchip.com/oxy/GUID-9C356E20-C5BD-430F-8C0B-CCA1B85ECC7C-en-US-3/GUID-F040354D-0842-4EFC-99F2-F1B8A649D106.html`

CAN TP plays a critical role in automotive communication stacks, especially in diagnostics over CAN (UDS on CAN), software updates, and bootloader communication. However, due to the lack of built-in security features such as authentication, encryption, or integrity verification, CAN TP may be susceptible to various threats like message injection, buffer overflows, or DoS attacks if not protected by additional security layers. In the context of this work, CAN TP is particularly relevant as it is used as the transport layer for receiving application images during the bootloader process, making it a potential attack vector if not properly secured.

5.6 Unified Diagnostic Services (UDS)

The Unified Diagnostic Services (UDS), standardized under ISO 14229 [9], is a diagnostic communication protocol widely adopted in the automotive domain for enabling communication between external diagnostic tools and ECUs. UDS operates over various transport layers including CAN (as UDS on CAN defined in ISO 15765-3 [10]), and builds upon protocols like CAN TP to support the reliable transfer of diagnostic messages.
UDS defines a rich set of diagnostic services organized into functional groups, each identified by a unique Service Identifier (SID). These services include, but are not limited to:

- **Diagnostic Session Control (0x10):** Initiates different diagnostic sessions such as default, extended, or programming sessions, each granting different levels of access to ECU functionalities.
- **ECU Reset (0x11):** Requests a reset of the ECU, with different reset types (soft, hard, etc.).

- **Security Access (0x27):** Manages authentication mechanisms through seed-key exchanges to protect critical operations.

- **Communication Control (0x28):** Enables or disables specific communication types (e.g., transmit/receive) to manage network traffic.

- **Routine Control (0x31):** Triggers execution of routines like memory checks, bootloader entry, or cryptographic operations.

- **Request Download (0x34) and Transfer Data (0x36):** Used to perform firmware updates or application flashing by downloading large data to the ECU.

- **Write/Read Data by Identifier (0x2E/0x22):** Enables reading or writing internal variables or configuration parameters identified by specific IDs.

A typical UDS communication involves a request-response model where the client (tester) sends a request containing a Service Identifier and optional parameters and the server (ECU) responds with either a positive or negative response code (NRC) depending on the success or failure of the requested service.

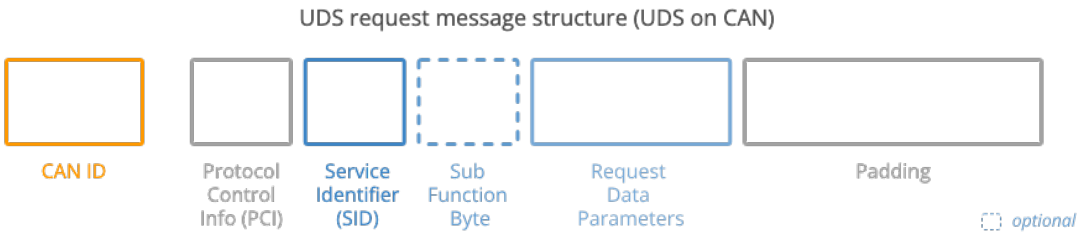


Figure 5.7: UDS on CAN request message. Source: `https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services`

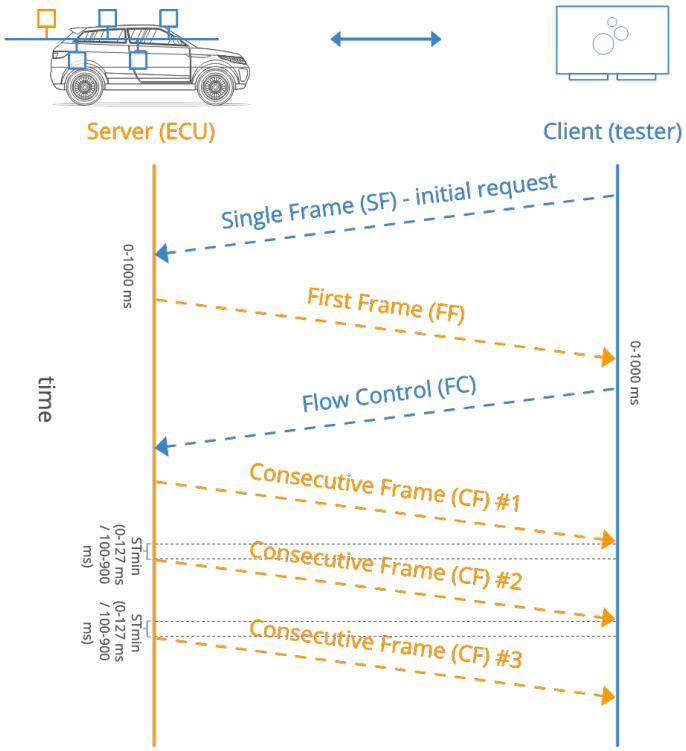


Figure 5.8: ISO TP multi-frame communication. Source: `https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services`

UDS is an extremely adaptable diagnostic protocol, capable of being deployed over a wide variety of underlying communication technologies, including CAN, FlexRay, Ethernet (DoIP), K-Line, and LIN, thus making it suitable for diverse automotive architectures and evolving vehicle network infrastructures.

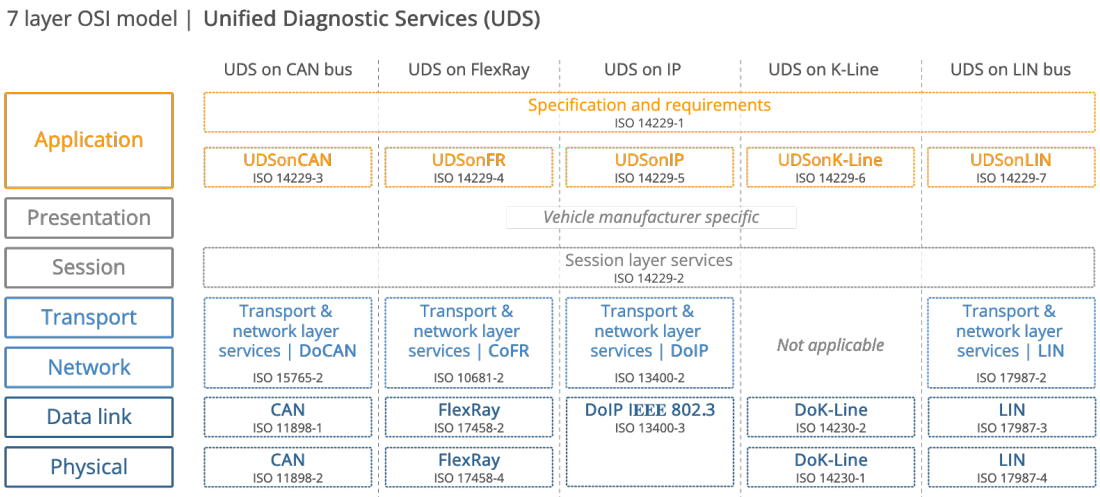


Figure 5.9: OSI model layers of UDS. Source: `https://www.csselectronics.com/pages/uds-protocol-tutorial-unified-diagnostic-services`

In this work, UDS over CAN (ISO 15765-3 [10]) is employed in conjunction with CAN TP to facilitate the secure reception and validation of firmware updates, the RoutineControl service is particularly relevant, as it is used to manage the secure flashing process and to trigger digital signature verification routines within the bootloader. Ensuring that UDS services like this are executed only within authenticated sessions (e.g. through successful SecurityAccess) is crucial to protecting the system from unauthorized firmware uploads or execution of privileged commands. While UDS offers powerful diagnostic capabilities and flexibility, it also presents potential security risks if not properly safeguarded. Without robust access control and cryptographic protections, attackers may exploit services like ECUReset or WriteDataByIdentifier to disrupt system operation or compromise integrity. Therefore, the secure implementation of UDS, especially in the context of firmware updates and bootloader design, is a fundamental aspect of embedded system cybersecurity.

Chapter 6

Laboratory Setup

In this chapter, it is described the laboratory setup on which the practical activities were carried out during the internship. The purpose of this section is to provide a detailed overview of the hardware and software components used, the configuration of the embedded system, and the development environment adopted. Particular attention is given to the elements involved in the secure boot process, the communication interfaces and the procedures used for firmware signing and verification. This context is essential to understand the implementation and validation of the security mechanisms described in the following sections.

6.1 Laboratory Setup Components

The whole setup is composed of several elements:

- **Host PC**: Company PC running Windows 11 OS, used for firmware development, compilation, flashing, and debugging.
- **Development Software**: The primary development environment is the *S32 Design Studio for Power Architecture*, an IDE provided by NXP, which integrates compiler toolchains, debugger configurations, and project management features.
- **Target Board**: The embedded platform used for testing is part of the *NXP S32K* family, more precisely part of the S32K3 family of automotive MCUs based on the Arm Cortex-M7 core. It includes an Hardware Security Engine (HSE) that integrates directly on chip several security features like cryptographic operations, key management and secure memory storage.
- **Debugger Interface**: A *Lauterbach PowerDebug PRO* is used as a high performance debugging and trace interface, connected to the board via the JTAG port. It allows precise control over the firmware execution and facilitates low-level debugging and trace analysis.
- **Communication Interface**: A *PEAK-System PCAN-USB* interface is employed to connect the Host PC to the CAN network on the target board. Through this interface, the host PC communicates with the target board to perform the bootloading routine following the UDS over CAN-TP protocol.
- **Power Supply**: The evaluation board is powered through a laboratory *DC power supply*, capable of providing a stable voltage source (12V) with adjustable current limits.

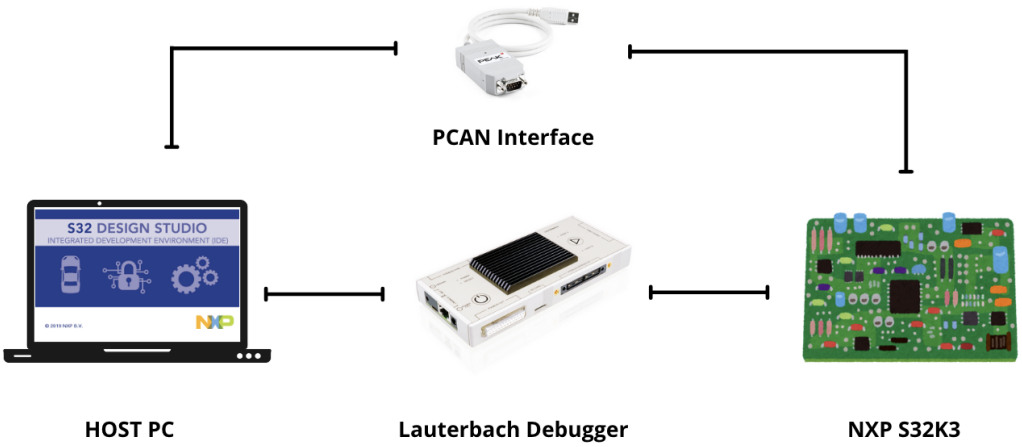


Figure 6.1: Laboratory setup scheme

6.1.1 Development software - Libraries

The environment required for this work also involves the installation of specific libraries, which can be directly installed through the dedicated window within the S32 Design Studio IDE called *Extensions and Updates*. The environment comprehend the following elements installed on the version 3.6.0 of the S32 Design Studio IDE:

- GDB Client for Arm Embedded Processors 15.1 Build 1703.
- GNU ARM PEMicro Interface Debugging Support v5.9.2.
- NXP GCC for Arm Embedded Processors v10.2 build 1728.
- NXP GCC for Arm Embedded Processors v11.4 build 1763.
- NXP GCC for Arm Embedded Processors v9.2 build 1649.
- S32 Design Studio Debugging Core v3.6.0.
- S32 Design Studio Platform Tools package v3.6.0.
- S32G development package v3.6.0.
- S32K1xx development package v3.6.0.
- S32K3 RTD AUTOSAR 4.4 v2.0.0.
- S32K3xx development package.

6.1.2 Target Board - NXP Hardware Security Engine (HSE_B) in S32K3

The **Hardware Security Engine (HSE)** integrated in the NXP S32K3 microcontroller is a dedicated on-chip security co-processor designed to offload and accelerate cryptographic operations,

enforce secure boot mechanisms, manage keys in a protected environment, and ensure compliance with stringent automotive security requirements (e.g., ISO 21434, AUTOSAR). The specific implementation employed in this work is the **HSE_B** firmware, an official NXP proprietary firmware image delivered encrypted and directly installed on the NXP board that verifies its authenticity and installs it reconfiguring the flash memory.

HSE_B Firmware Installation

The installation of HSE_B can happen in two ways that define how the HSE firmware image is flashed into the device's non-volatile memory:

- **FULL_MEM Mode:** In this mode, the entire reserved HSE firmware memory region is used for a single image that is written once and occupies the full partition dedicated to the HSE. This approach is simpler and more space-efficient, but it does not support firmware rollback or atomic updates, therefore, it is more suitable for production deployments where the firmware is stable and secure boot integrity checks are firmly established. Firmware updates using FULL_MEM require halting the application and reprogramming the entire HSE region (no zero-downtime updates).

- **AB_SWAP Mode:** In this configuration, the memory area reserved for the HSE is logically split into two partitions, partition A and partition B and only one of the two partitions is active at a time, while the other remains available for staging a new firmware update. This method enables safe and atomic updates since a new HSE firmware version can be downloaded and flashed into the inactive partition, and once validated, the active pointer is swapped to point to the new image. If any issue occurs during update or boot, the system can fall back to the previous working image, ensuring robustness and preventing bricking due to corrupted or incomplete updates. Moreover this is a solution that permits zero-downtime updates.

The choice for this work has been FULL_MEM since it represents the only reversible way of installing the HSE_B on the board.

HSE_B Messaging Unit

The HSE architecture is functionally isolated from the application core (host domain) since communication between the main application running on the Cortex-M7 core and the HSE firmware occurs through a shared **Messaging Unit (MU)**, which consists of a set of registers used by the host to trigger service requests and receive service responses while from the HSE it is used to receive service requests, provide service responses along with HSE status information relevant to the host.
The Messaging Unit (MU) operates through a dual-interface architecture, consisting of two sides: *MUA* and *MUB*. In this configuration, the Hardware Security Engine (HSE) exclusively manages the *MUA* side, while the host processor is granted control over the *MUB* side. Communication between the host and the HSE is established by writing data into a 32-bit readable and writable transmit register ($\texttt{TR}_\texttt{i}$) on one side, which can then be read from the corresponding 32-bit read-only receive register ($\texttt{RR}_\texttt{i}$) on the opposite side. Additionally, certain control registers on one interface, such as the 32-bit read-only Flag Control Register (`FCR`), are directly linked to 32-bit read-only status registers on the other side, like the Flag Status Register (`FSR`).

Service Descriptors

To request a service from the Hardware Security Engine (HSE), the host must prepare and transmit a *service descriptor*, which is a data structure residing in memory that encapsulates all necessary parameters for a specific HSE operation. The descriptor includes fields such as the service ID (defining the type of request, e.g. key generation, signature verification), service-specific parameters (e.g. pointers to key handle or data buffers), and optional flags. Once the descriptor is populated, the host writes its memory address into a transmit register ($\texttt{TR}_\texttt{i}$) associated with a free service channel of a Messaging Unit (MU) instance. This action triggers the HSE to begin processing the requested service. The HSE internally reads the descriptor from the specified memory location, performs the requested operation, and then writes the result into the corresponding receive register ($\texttt{RR}_\texttt{i}$). The host can poll or be notified via interrupt when the service completes, and it must subsequently read the response and clear the channel before reusing it. The correct alignment and location of service descriptors in memory are critical, they must be located in accessible RAM regions and conform to the expected structure layout as defined in the HSE Reference Manual.

Service Channel

A *service channel* is a temporary communication path that links a service request to a specific pair of transmit/receive registers ($\texttt{TR}_\texttt{i}/\texttt{RR}_\texttt{i}$) within a Messaging Unit (MU) instance. Each channel remains available until the host writes the address of a service descriptor to the corresponding $\texttt{TR}_\texttt{i}$ register, thereby initiating the request. The channel remains occupied while the HSE processes the service and until the host reads the result from the $\texttt{RR}_\texttt{i}$ register. On the S32K3, with two MU instances and four transmit registers each, up to eight service channels can be used concurrently. A service channel is considered free when the transmit register is empty (bit i in the `TSR` is 1), the receive register has been read (bit i in the `RSR` is 0), and the channel is not currently being processed (bit i in the `FSR` is 0). Service requests are triggered by writing the service descriptor address into a free channel, note that channel 0 is reserved exclusively for administration services, using it for other service types results in an error. Proper management of service channels is essential to avoid writing to channels that are currently busy, which would otherwise lead to incorrect behavior or system faults.

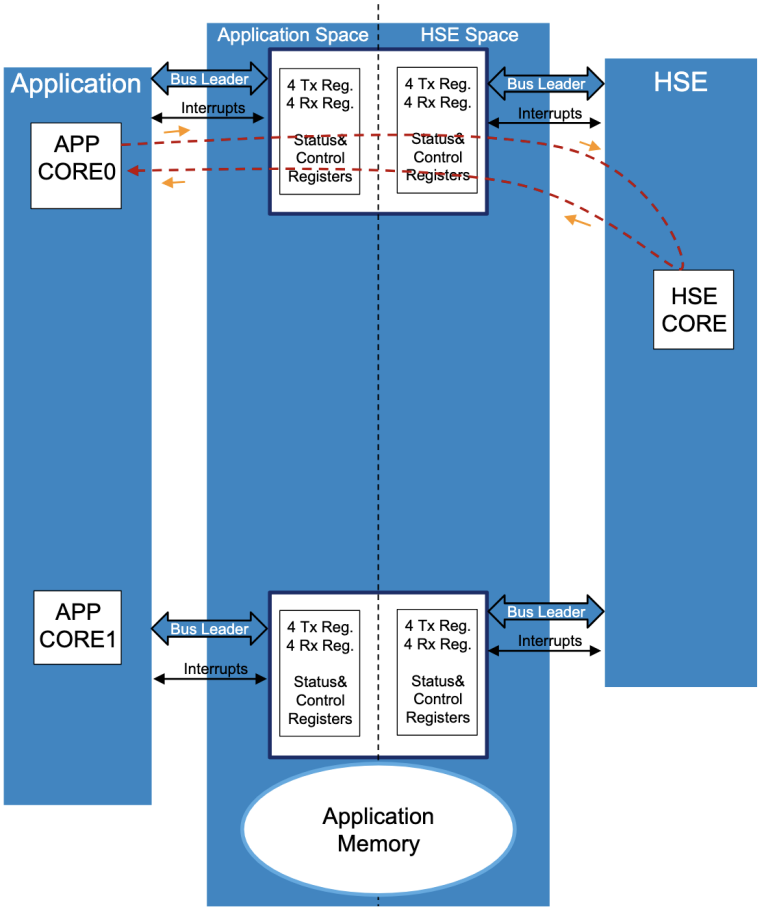


Figure 6.2: HSE_B Messaging Unit

Security Features

The HSE_B supports a wide range of security services, including:

- **Secure Boot and Application Authentication**: Verifies the integrity and authenticity of the application firmware before execution using cryptographic checks over the image.
- **Symmetric and Asymmetric Cryptography**: Supports algorithms like AES (ECB, CBC, GCM), HMAC, CMAC, and asymmetric operations such as RSA and **ECDSA** over standardized NIST curves (e.g. P-256).
- **Random Number Generation**: Includes a TRNG (True Random Number Generator) compliant with NIST SP 800-90B for secure key generation and nonce creation.
- **Key Derivation and Lifecycle Management**: Supports symmetric and asymmetric key derivation, as well as the ability to securely load, delete, or rotate keys based on lifecycle policies.
- **Secure Debug Control and Lifecycle Transitions**: Enables restriction of debug access (password-based or through asymmetric challenge-response and the possibility to manage

the device lifecycle from development phase which permits to operate always with high privilege to production and distribution phases where all the protections are enabled.

Cryptographic Keys

The keys available to the host via cryptographic services are categorized into specific groups based on their types, organized within catalogs that the host configures statically. Each key resides in a dedicated key slot containing its value and associated attributes which are configured dynamically by the host through key management services. These keys can be provisioned using multiple methods, including direct generation or derivation from existing keys. The catalogs for keys stored in non-volatile memory (NVM) and RAM are defined within the SYS-IMG, whereas the catalog for keys stored in ROM is defined within the HSE.

Key Group and Key Type

A key group consists of a set of cryptographic keys of the same type where each group is assigned an index within the key catalog in which it is defined. This index reflects the order in which the groups are declared, the first group is assigned index 0, the second index 1, and so forth.
The different key types possible are:

Key Type	**Description**	**Key Catalog**
HSE_KEY_TYPE_AES	AES key	NVM and RAM
HSE_KEY_TYPE_SHE	AES key used with SHE-specific services	NVM and RAM
HSE_KEY_TYPE_HMAC	HMAC key	NVM and RAM
HSE_KEY_TYPE_RSA_PAIR	RSA key pair (public and private)	NVM only
HSE_KEY_TYPE_RSA_PUB	RSA public key	NVM and RAM
HSE_KEY_TYPE_RSA_PUB_EXT	RSA public key, stored in application NVM	NVM and RAM
HSE_KEY_TYPE_ECC_PAIR	ECC key pair (public and private)	NVM and RAM
HSE_KEY_TYPE_ECC_PUB	ECC public key	NVM and RAM
HSE_KEY_TYPE_ECC_PUB_EXT	ECC public key, stored in application NVM	NVM and RAM
HSE_KEY_TYPE_DH_PAIR	DH key pair (public & private)	NVM and RAM
HSE_KEY_TYPE_DH_PUB	DH public key	NVM and RAM
HSE_KEY_TYPE_SHARED_SECRET	Shared secret – can be used to derive a secret key	RAM only

Key Slots and Key Values

A key slot is a memory location used to store a single key along with its value(s) and associated attributes. Each slot is identified by an index within the key group it belongs to and this index corresponds to the order in which the slots are defined within the group: the first slot has index 0, the second has index 1, and so forth.
Key Values are represented by unsigned integers of various sizes that depend on the key size. For example an ECC Keypair is composed by:

- ECC Private Key: A private random generated scalar W where the key size is given by the bit length of W.
- ECC Public Key: A public point Q on the elliptic curve that results from the multiplication of the private scalar W and a given point G called generator (as seen in 5.2.1).

Key Attributes

A stored key can have several key attributes available to the host to be configured.

Key Attribute	**Description**	**Type**
Bit size	Specifies the key length in bits.	16-bit integer
Update counter	Counter to prevent roll-back; new values must increment. For NVM keys: 0 to 0xFFFFFFFE. For RAM keys: fixed at 0xFFFFFFFF. For SHE keys: 28-bit counter (RAM keys set to zero).	32-bit integer
MU instance map	Flags indicating which Messaging Unit (MU) instances can activate key services. Defined at key group level.	Bit field
SMR verification map	Flags specifying which Secure Memory Regions (SMR) must be verified before the key can be used.	Bit field
Key Type	Key category identifier as defined in the key access restriction flags.	8-bit integer
ECC Curve ID / RSA Exponent Size / AES Block Mode	Indicates one of: ECC curve ID, RSA public exponent size (in bytes), or AES block mode mask, depending on key type.	8-bit integer
Access restriction flags	Flags controlling how and when the key can be accessed. See the access restriction section.	Bit field
Usage flags	Flags defining allowed usage operations for the key. See the usage flags section.	Bit field

One fundamental key attribute is the **Key Usage Flags**, because it strictly define the purpose of the key. The different key usage flags are defined by macros and are:

Key Flag	**Effect When Enabled**
HSE_KF_USAGE_ENCRYPT	Allows the key to be used for encryption purposes.
HSE_KF_USAGE_DECRYPT	Permits the key to participate in decryption operations.
HSE_KF_USAGE_SIGN	For RSA/ECC: enables signature generation (private key only). For AES/HMAC: permits MAC creation.

HSE_KF_USAGE_VERIFY	For RSA/ECC: enables signature verification (public key only). For AES/HMAC: allows MAC verification.
HSE_KF_USAGE_EXCHANGE	Authorizes use in key exchange protocols such as DH or ECDH.
HSE_KF_USAGE_DERIVE	Grants permission to derive new keys from this one. Note: not valid for RSA, ECC, or DH key types.
HSE_KF_USAGE_KEY_PROVISION	Restricts usage to key provisioning operations (import/export of keys). When this flag is set, the key is only valid for decrypting imported keys or encrypting keys for export. If not set, the key can be used for regular cryptographic operations on memory data (e.g., encryption, verification). Attempting to use the key for provisioning when the flag is cleared results in an error.
HSE_KF_USAGE_AUTHORIZATION	The key is valid for authenticating requests to elevate privileges to Super User (SU). Requires HSE_KF_USAGE_VERIFY to be set, and HSE_KF_USAGE_SIGN must not be set.
HSE_KF_USAGE_SMR_DECRYPT	Allows the key to decrypt Secure Memory Regions (SMR). If enabled at installation, the firmware disables the standard decryption flag (HSE_KF_USAGE_DECRYPT).

Key Catalog

The HSE manages cryptographic keys using three distinct key catalogs: **ROM**, **NVM**, and **RAM** and each of them is uniquely identified and organizes its keys into groups. The ROM key catalog is static and pre-configured by NXP during production, with keys stored in secure NVM (like the one to decrypt and install the HSE firmware). In contrast, the NVM and RAM catalogs are configurable by the host. The NVM catalog stores key attributes in the system image and key values either in the system image or application NVM. The RAM catalog holds both key values and attributes directly in secure RAM. The following table summarizes the properties of each catalog.

Key Catalog ID	**Value**	**Configurable**	**Description**
HSE_KEY_CATALOG_ID_ROM	0	No	ROM key catalog; keys are stored in secure NVM and provisioned by NXP before shipment.
HSE_KEY_CATALOG_ID_NVM	1	Yes	NVM key catalog; key values are stored in the system image or in application NVM (e.g., for select public RSA and ECC key certificates). Key attributes are stored in the system image.
HSE_KEY_CATALOG_ID_RAM	2	Yes	RAM key catalog; both key attributes and values are stored in secure RAM.

The NVM and RAM key catalogs are configured statically by the host through a table, in

which each entry defines the attributes of a key group. A **key group** is characterized by the following five properties:

- **MU instance map:** Indicates which Messaging Unit (MU) instances are allowed to access the key group (MU0 or MU1).
- **Owner:** Identifies the owner of the key group (see the table on key group owners 6.1.2).
- **Key type:** Specifies the type of key (see the table 6.1).
- **Number of key slots:** Defines how many individual keys the group can contain.
- **Maximum key size:** Sets the upper limit for the key size in bits, depending on the key type.

The allowed maximum key size for a given key type is constrained by design and summarized in the table below.

Key Type	**Allowed Maximum Key Sizes (in bits)**
HSE_KEY_TYPE_AES	128, 192, or 256
HSE_KEY_TYPE_SHE	128
HSE_KEY_TYPE_HMAC	128 up to 1152
HSE_KEY_TYPE_ECC_PAIR	192 up to 640
HSE_KEY_TYPE_ECC_PUB	192 up to 640
HSE_KEY_TYPE_ECC_PUB_EXT	192 up to 640
HSE_KEY_TYPE_RSA_PAIR	1024 up to 4096
HSE_KEY_TYPE_RSA_PUB	1024 up to 4096
HSE_KEY_TYPE_RSA_PUB_EXT	1024 up to 4096
HSE_KEY_TYPE_DH_PAIR	1024 up to 4096
HSE_KEY_TYPE_DH_PUB	1024 up to 4096
HSE_KEY_TYPE_SHARED_SECRET	128 up to 2048

Each key group is assigned an owner, which defines the access control policies for managing the keys within that group determining whether a host is allowed to perform operations such as provisioning or modifying keys, depending on its rights and identity (HID). The host's privileges are evaluated based on whether it holds *Super User (SU)* or *User* rights, and whether it matches the expected Host ID (CUST or OEM). The following table summarizes the ownership options and their effects.

Key Group Owner	**Applies to**	**Description**
HSE_KEY_OWNER_CUST	NVM key catalog	Keys can be fully managed if the host has SU rights and the HID is CUST (system integrator). With User rights, the host may provision keys only if it knows a key owned by CUST.
HSE_KEY_OWNER_OEM	NVM key catalog	Full management is allowed with SU rights and HID set to OEM. With User rights, provisioning is allowed only with knowledge of an OEM-owned key.

HSE_KEY_OWNER_ANY	NVM[1] and RAM key catalogs	With SU rights, keys can be managed freely. With User rights, key management operations are restricted.

Key Handle

A key handle is a 32-bit identifier (*hseKeyHandle_t*) that uniquely identifies a key within a key catalog and that is used in all services requiring key access. The format of the key handle is structured by bit fields that define the catalog, group, and slot where the key is located.

Bit Number	31 - 24	23 - 16	15 - 8	7 - 0
Description	0	Key Catalog ID	Key Group Index	Key slot Index

Figure 6.3: Key Handle Structure

Where:

- 31-24: Reserved and Must be set to 0.
- Key Catalog ID: Identifies the key catalog (see 6.1.2).
- Key Group Index: Indicates the position of the key group in the catalog configuration (first group has index 0).
- Key Slot Index: Indicates the index of the key within the specified group (starts from 0).

Key Management

Key Management services can be used by the host to:

- Initialize and update key values.
- Export key values.
- Generate and derive key values.
- Establish secret keys in a secure way.

First of all, the services described in this section are available to the host only after the key catalog have been formatted using the dedicated service to be invoked.
All the keys[2] can be imported through the import service (*hseImportKeySrv_t*).
A key can be imported both in plain or encrypted (using a dedicated encryption key). In this work the only import needed is the the public ECDSA key needed to verify the signature on the

[1] With some limitations

[2] With some exeptions, consult the HSE_B reference manual (RM00286) for more specific information

application image so encryption is not needed.
The key import service can be used only by a user who has been granted with super user rights. The key management services include the possibility to **securely erase a key** using the dedicated service (*hseEraseKeysSrv_t*). Keys in the RAM key catalog can be erased unconditionally while keys in the NVM can be erased only if the host is granted with Super User Rights.

6.1.3 Debugger Interface - Lauterbach debugger & TRACE32

The Lauterbach PowerDebug PRO is a high-performance JTAG-based debug and trace probe that provides comprehensive hardware level control over the target microcontroller, allowing detailed management of the execution flow through the insertion of breakpoints, watchpoints, and single-step execution while also enabling real-time inspection of variables, memory contents, CPU registers, and the call stack. Its advanced trace capabilities capture both instruction and data traces, which facilitate thorough analysis of program execution paths and timing, allowing for effective performance profiling and identification of bugs.

Figure 6.4: Lauterbach PowerDebug System

This debugger is operated through the Lauterbach TRACE32 software suite, which integrates a powerful graphical user interface alongside a flexible command line environment, offering features such as complex breakpoint conditions, multi-core debugging, and comprehensive visualization tools that display variable states, stack calls, peripheral registers, and memory mappings, thus streamlining the process of low-level firmware debugging and system validation with a high degree of precision and efficiency.

6.1.4 Communication Interface - PEAK-PCAN-USB & PCAN-View

The PEAK-System PCAN-USB interface serves as the communication bridge between the host PC and the CAN network on the target board, providing a reliable connection that supports CAN protocol standards essential for automotive and embedded system development. This interface enables the host PC to transmit and receive CAN messages, which are crucial for executing the bootloading routine in compliance with the UDS (Unified Diagnostic Services) protocol over the CAN Transport Protocol (CAN-TP). The communication process is monitored and analyzed using the PCAN-View software, which offers a comprehensive graphical interface for real-time visualization and logging of CAN messages, including support for filtering, message interpretation, and error frame detection. PCAN-View facilitates the inspection of message identifiers, data

payloads, timestamps, and bus load statistics, providing a very useful tool for debugging and verifying communication sequences during the bootloading process as well as for general CAN bus diagnostics and testing.

Chapter 7

Program Development

This chapter provides a detailed description of the implemented work, outlining the guiding principles and explaining the operational functionality of the embedded board. During this phase of the internship it has been implemented, starting from the NXP Unified Bootloader project cited in 4.6, a secure application image loading procedure that authenticates the application image using an ECDSA digital signature.

7.1 Project Structure

The project folder includes all the elements involved in the process of generating asymmetric keys, signing the application image, loading the application image on the target board and verifying the signature over the application image. The different programs included are reported below:

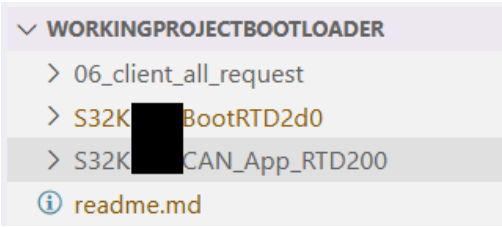


Figure 7.1: Project folder structure

Where:

- 06_client_all_request is the host PC side folder where are contained a series of python scripts to be used to invoke the signature procedure over the image and the image loading trough CAN routines.
- S32K***_BootRTD2d0 contains the bootlaoder program executed on the target board to manage the signature keys and the signature verification process.
- S32K***_CAN_App_RTD200 is the application image to be signed, loaded and executed on the target board after signature verification.

- readme.md is a markdown file containing all the instructions useful to autonomously execute the programs on both host PC side and target board side and replicate the experiments done during this work.

7.2 Execution flow

7.2.1 Key generation

First, the application image has to be signed by the authorized software provider (hypothetically Avio Aero), which generates an asymmetric ECDSA keypair. During the internship, the keypair was created using OpenSSL with the standard NIST P-256 curve. The private key used to sign the application image is stored in the file `eccPrivateKey.pem`.

7.2.2 Signature generation and Flashing Procedure Script

The signing process for the application image is handled by a Python script, which also takes care of loading the image onto the target board via CAN UDS routines (`\workingProjectBootloader \06\_client\_all\_request\Flashing\_Procedure.py`). The script is run from the command line. As explained in the `readme.md` file, it takes four arguments (i.e., `.\Flashing_procedure.py argv1 argv2 argv3 argv4`), which are:

1. `argv1`: address where the flash drivers will be placed in the target board's flash memory.
2. `argv2`: name of the binary file containing the flash drivers.
3. `argv3`: address where the application image will be placed in the target board's flash memory.
4. `argv4`: name of the binary file containing the application.

The first thing the script does is generate the signature with OpenSSL:

```
openssl dgst -sha256 -sign eccPrivateKey.pem -out signature.bin App_filename
```

Here, `App_filename` is the fourth argument when running the script.
The generated signature is in ASN.1 format and is then split into the two values r and s (32 bytes each) that make up the ECDSA signature. If everything goes well, the signature is printed in the terminal along with a success message. If there's an error, the process is automatically retried until a valid signature is produced.
After that, the script follows the same execution flow as the original NXP Unified Bootloader project:

1. Initialize the UDS communication session.
2. Map and define source and response messages.
3. Switch to Extended Session, which allows certain diagnostic accesses but no flashing rights.
4. Switch to Programming Session, which sets up the environment for reprogramming the target board's flash.

5. Security Access 1[1] unlocks moderately restricted diagnostic functions but still no flash programming.
6. Security Access 2 unlocks full reprogramming rights, including flash erase, firmware download, and memory programming.
7. Download the flash drivers (second script argument) and run control checks on them.
8. Download the application image (fourth script argument).
9. Run signature checking for the application image.
10. Download the application image signature and verify it.

The signature verification decides whether the application image is valid. If it is, it will run at the next power-on, if not, it's erased from flash and a new image needs to be loaded. The application's validity is determined by the APP Validity Flag described in 4.6. Finally, when the script finishes, a physical `ECUReset` message is sent to the target board to restart it and launch the bootloader.

7.2.3 Signature Checking Procedure

The signature checking process is invoked as a `UDS_RoutineControl 0x31` request, identified in the code as `UDS_IsSignatureRoutineControl`, with the routine identifier `0x31 0x01 0x03 0x03`.

```
//Is signature routine control?
else if(TRUE == UDS_IsSignatureRoutineControl(m_pstPDUMsg))
{
    //save signature value
    Flash_SavedReceivedSignature( &m_pstPDUMsg->aDataBuf[4u] );

    /*request client timeout time*/
    UDS_SetNegativeErroCode(i_pstUDSServiceInfo->serNum, RCRRP, m_pstPDUMsg);

    m_pstPDUMsg->pfUDSTxMsgServiceCallBack = &UDS_DoSignature;
}
```

Figure 7.2: Signature Routine Control Switch Case Detail

When this routine is received, the function `UDS_IsCheckRoutineControlRight` is first called to verify that the parameters passed match the expected format for the requested routine. If the parameters are valid, the received signature is stored in a dedicated RAM buffer via `Flash_SavedReceivedSignature`. To avoid blocking the communication flow with a long cryptographic check, the UDS service handler immediately responds with a *Response Pending* (NRC 0x78) to keep the tester session alive, and then assigns the `pfUDSTxMsgServiceCallBack` function pointer to `UDS_DoSignature`, which will set the active job (`SIGNATURE_CHECKING`) to be executed

[1]Security Access 1 and 2 are both implemented using the UDS SecurityAccess service (0x27) with a seed/key challenge–response. Each level has its own seed and corresponding key, so obtaining Level 1 access does not grant Level 2 privileges.

in deferred mode and invokes the `UDS_DoResponseSignature` function that meanwhile prepares a final positive or negative response for the signature verification service.

Deferred execution in the main loop The bootloader `main()` function continuously calls `BOOTLOADER_MAIN_Demo()`, which drives the UDS state machines. During each iteration, the UDS manager checks whether `pfUDSTxMsgServiceCallBack` is non-`NULL`.

```
int main(void)
{
    /* Write your code here */
    BOOTLOADER_MAIN_Init(&BSP_init, &AbortTransferMsg);
    for(;;)
    {
        BOOTLOADER_MAIN_Demo();
        SendMsg();
        if(exit_code != 0)
        {
            break;
        }
    }
    return exit_code;
}
```

Figure 7.3: Bootloader main function

If it is, the `Flash_OperateMainFunction()` function is called and inside it, using a switch, the function to execute is selected (in this case `Flash_SignatureCheck`). This function calls the `isSignatureValid` function that runs the ECDSA signature verification against the stored buffer using the *HSE_B* services by crafting the service request and sending it.
Since this whole process happens during the UDS transmission, to not violate the timing constraints defined by the UDS protocol, the `HSE_Send` function, has been modified and divided in a three-state machine to manage iterative calls of the callback function while waiting for the operation to be completed:

- Case 0: the object `pHseSrvDesc` is filled with all the information needed to send a service request to the HSE as defined in 6.1.2. After the request is ready, it is sent to the `HSE using HSE_Send_2()`, which is a simplified, non-blocking version of `HSE_Send()`, that originally manages both synchronous and asynchronous modes, including waiting for the response, handling callbacks, and tracking channel state. In the modified version, all this waiting logic is removed since the request is sent with `HSE_MU_SEND_NON_BLOCKING()` and the function immediately returns `HSE_SRV_RSP_UNFINISHED`. This change was introduced because the call is made during the CAN-UDS session since any blocking inside the function could delay the response or reception of UDS frames, potentially causing the session to time out.

- Case 1: in this case there is an active waiting for the HSE response where if the response is `HSE_SRV_RSP_UNFINISHED`, then the state machine does nothing and waits. Otherwise, when a different response is received, the result is saved and it goes to the next state.

- Case 2: this is the exit state which marks the operation as finished.

```
boolean SignatureValidity(boolean * finished, const uint8_t* sig,
                          uint32_t appStartAddress, uint32_t appLength)
{
    hseSrvDescriptor_t* pHseSrvDesc = &gHseSrvDesc[0U][1U];
    hseSignSrv_t* pSignSrv;

    memcpy(a, sig, 32);
    memcpy(b, sig + 32, 32);

    switch (state)
    {
    case 0: // crafting service request and send it to HSE
        memset(pHseSrvDesc, 0, sizeof(hseSrvDescriptor_t));
        pHseSrvDesc->srvId = HSE_SRV_ID_SIGN;
        pSignSrv = &(pHseSrvDesc->hseSrv.signReq);
        pSignSrv->accessMode          = ACCESS_MODE;
        pSignSrv->signScheme          = signScheme;
        pSignSrv->authDir             = VERIFY;
        pSignSrv->keyHandle           = EcckeyPubHandle;
        pSignSrv->inputLength         = appLength;
        pSignSrv->pInput              = StartAddress;
        pSignSrv->bInputIsHashed      = ...;
        pSignSrv->sgtOption           = ...;
        pSignSrv->sig[0]       = (ADDR)a;
        pSignSrv->sigl[0] = (ADDR)&aLen;
        pSignSrv->sigl[1] = (ADDR)&bLen;
        pSignSrv->sig[1]       = (ADDR)b;

        HseResponse = HSE_Send_2(0U, 1U, xxx, pHseSrvDesc);
        state++;
        *o_pbIsOperateFinsh = FALSE;
        break;

    case 1: // wait HSE to respond
        HseResponse = HSE_MU_ReceiveResponseNotBlocking(0U, 1U);
        *o_pbIsOperateFinsh = FALSE;
        if (HseResponse == HSE_SRV_RSP_UNFINISHED) {
            // wait for the response
        } else {
            if (HSE_SRV_RSP_OK == HseResponse) {
                result = TRUE;
            }
            *o_pbIsOperateFinsh = TRUE;
            state++; // when the response is received, go to the next state
        }
        break;

    case 2:
        *o_pbIsOperateFinsh = TRUE; // operation finished
        break;

    default:
        break;
    }

    return result;
}
```

UDS Response crafting Following the UDS protocol a positive response to a service is formed by adding `0x40` to the original service identifier (SID). For the `RoutineControl` service (`0x31`), the corresponding positive response SID is therefore `0x71`. The rest of the response maintains the same subfunction and routine identifier of the request (`0x01 0x03 0x03`), followed by a *routine specific status byte* that indicates the outcome of the operation (`0x00` for successful signature verification, `0x01` for failure).
This format ensures that the tester can unambiguously correlate the response to the original request and interpret the result of the executed routine.

When the process is finished, the callback is cleared, and the TP layer transmits the final response in its next transmission cycle.
Advantages are:

- Preventing block of the communication stack during long cryptographic operations.
- Keep UDS session alive using NRC 0x78 until the final result is ready.

7.2.4 Target Board Bootloader

When the target board powers on, the bootloader is the first piece of software to run. Its primary role is to determine whether control should be passed to the existing application or whether it should remain in bootloader mode to receive and program a new image. This decision is based on the *APP Validity Flag*, a status indicator that reflects whether a valid application is present in flash memory.
The entry point of the bootloader program is the `main` function in `\workingProjectBootloader\S32K***\_BootRTD2d0\BSWL\Startup\Core0\src\main.c`. This function immediately calls `BOOTLOADER_MAIN_Init` which performs initial hardware and communication setup before invoking `Boot_JumpToAppOrNot`, a function perform the decision: if the APP Validity Flag is set to TRUE and no flashing procedure has been requested by the host, execution jumps to the application image using the `Boot_JumpToApp` function. At this point, the reset handler address in the vector table is updated to point to the application's entry point in flash, effectively transferring control to user code.

```
void Boot_JumpToAppOrNot(void)
{
    uint32 resetHandlerAddr = 0u;

    if((TRUE == Boot_IsAPPValid()) && (TRUE != Boot_IsRequestEnterBootloader()))
    {
        Boot_RemapApplication();

        resetHandlerAddr = Flash_GetResetHandlerAddr();

        Boot_JumpToApp(resetHandlerAddr);
    }
}
```

Figure 7.4: Jump To App or not function

This structure allows the bootloader to act as a gatekeeper for the device firmware since it can either launch a known good application immediately, or stay active to accept firmware updates

via the chosen communication protocol (in this case, CAN with UDS services). This ensures that corrupted or incomplete application images are never executed, preserving system stability.
Since the signing and signature verification processes are computationally demanding for embedded systems like the one in this work, at each reset the only check that remains active is the CRC verification already implemented in the NXP Unified Bootloader. While this provides a basic level of integrity verification, it does not offer strong protection against intentional tampering. An improvement that was considered, but not implemented due to the time constraints of the internship, would be to leverage the secure memory regions (SMR) available through the HSE_B to store cryptographically strong integrity tags (e.g. CMAC or similar). This would allow the system to perform, at every startup, a more secure and reliable integrity check of the application image before executing it, combining the efficiency of the existing CRC mechanism with the robustness of modern authentication codes.

7.2.5 Application Image Content

Since the main focus of this work was on the bootloader, the program that is loaded on the target board after a successful flashing procedure is intentionally minimal. In this case, it is a simple blinking LED application, chosen because it provides a clear and immediate visual indication that the application has started running, without introducing additional complexity or unrelated functionality.

In a real avionic context, the application image to be deployed on an ECU would typically be far more complex, incorporating safety-critical control logic, communication stacks, sensor fusion algorithms, and diagnostic routines. The size of such software can vary significantly depending on the platform and certification requirements, but for modern avionic ECUs it is not unusual for a compiled and linked application image to range from a few hundred kilobytes to several megabytes.

Chapter 8

Results

The implementation of the proposed security mechanisms on the NXP S32K*** platform has led to a substantial improvement in the protection of the application image loading process for avionic ECUs. Moving from a CRC32-only verification to an architecture based on *ECDSA* digital signature validation, executed through the Hardware Security Engine (HSE_B), has significantly increased the system's resilience against the attack vectors identified during the Product Security Assessment (PSA) made in 4.4.

8.1 Achieved Security Improvements

The original bootloader relied solely on a CRC32 integrity check which even if effective for detecting accidental data corruption, offers no cryptographic strength, making it insufficient against deliberate tampering. In the secured implementation developed in this work, **authenticity** and **integrity** are enforced through **ECDSA signature verification**. This mechanism ensures that only firmware images produced and signed by an authorized software provider can be executed, preventing unauthorized code injection, malicious modifications, and supply chain tampering.

Another key improvement is the integration of the signature verification process into the timing critical CAN UDS communication flow. In the baseline system, adding cryptographic operations directly within the bootloader violated UDS timing constraints, leading to session timeouts. To address this, the signature verification routine was implemented using a non-blocking adaptation of the `HSE_Send` function giving an approach that defers long cryptographic operations while keeping the diagnostic session alive through response pending messages, preserving protocol compliance without sacrificing security and making it suitable for CAN UDS application which are very common in the avionic and automotive industry.

Execution control is now more robust since the APP Validity Flag no longer depends solely on the CRC check, but also on the outcome of the cryptographic verification. Unsigned or invalid applications are rejected before execution, and the system remains in bootloader mode until a valid, signed image is provided. These measures align with multiple High-Level Requirements defined in the PSA and contribute to compliance with aerospace security standards.

8.2 Performance Considerations

To compare the performances before and after the introduction of new security features, execution time analysis has been conducted.

The three targets of this analysis are:

- Base Bootloader, which is the NXP Unified Bootloader described in section 4.6 that has no security measure implemented on it. It can be considered as a plain bootloader.
- Secure Bootloader, including the process of signature generation over the application image performed by the Host-PC before starting the CAN communication with the target board.
- Secure Bootloader after the signature generation considering only the exchange between Host-PC and target board and the signature verification. This is the more realistic scenario considering that when updating the firmware, the target board receives the application image already signed.

Moreover these execution time are referred to the loading of a $\sim 120KB$ application image. Bigger images will require more time to be exchanged with CAN-UDS.

Table 8.1: Comparison of Bootloader Execution Times

Base Bootloader [s]	**With Security: Including Signature** [s]	**With Security: After Signature** [s]
53.3772	54.5618	53.7791
52.6974	54.1341	53.7256
52.5216	54.8869	54.2985
52.9448	56.5208	54.7715
52.8405	55.1842	54.5487
52.8693	55.5819	54.9310
52.2216	67.3238	54.9638
53.5779	57.6514	55.1414
52.4730	57.5739	56.6954
52.7652	57.7062	57.1501
52.6918	55.2805	54.8632
53.0480	57.3037	54.7867
52.8433	57.4537	55.7898
52.2536	56.9467	55.1474
52.6152	56.9200	54.9808
52.7689	56.8228	55.1504
52.5723	57.9515	56.0676
53.3604	59.1366	58.2269
52.6688	54.3598	52.6546
52.5117	52.9571	52.7021
50.0911	52.7102	51.8491
52.4202	53.0074	52.6966
52.9448	52.8735	52.5931
52.5215	52.5574	52.2049
52.5653	53.9963	52.4576

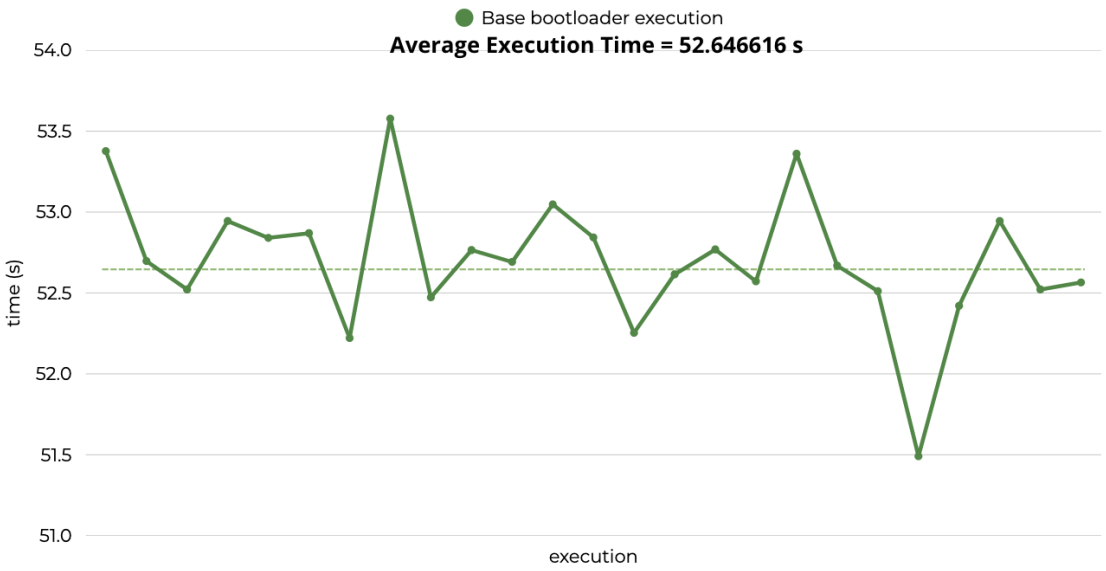


Figure 8.1: Basic Bootloader Execution Time

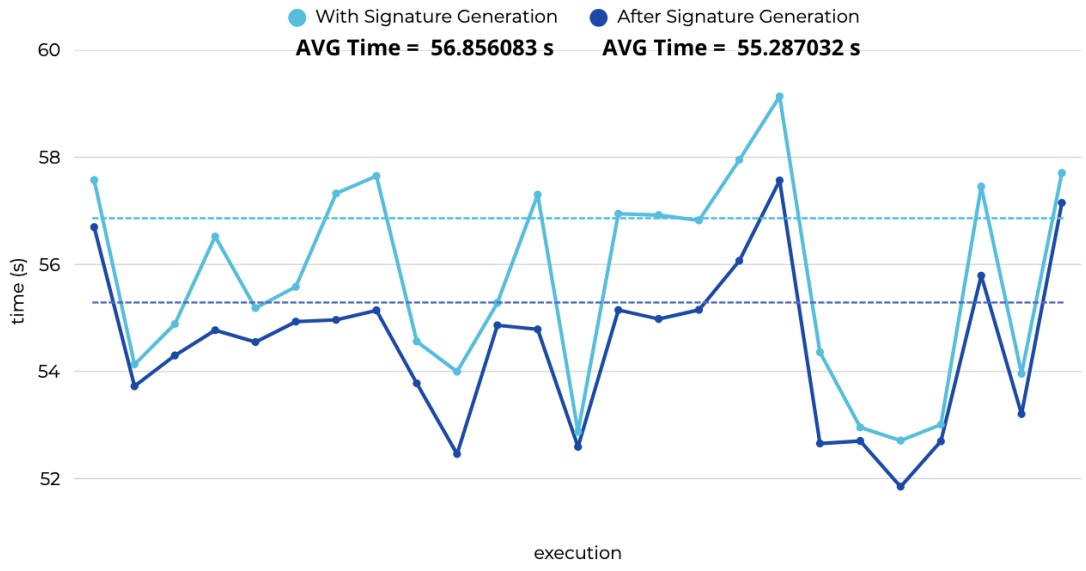


Figure 8.2: Bootloader Execution Time with Security Features

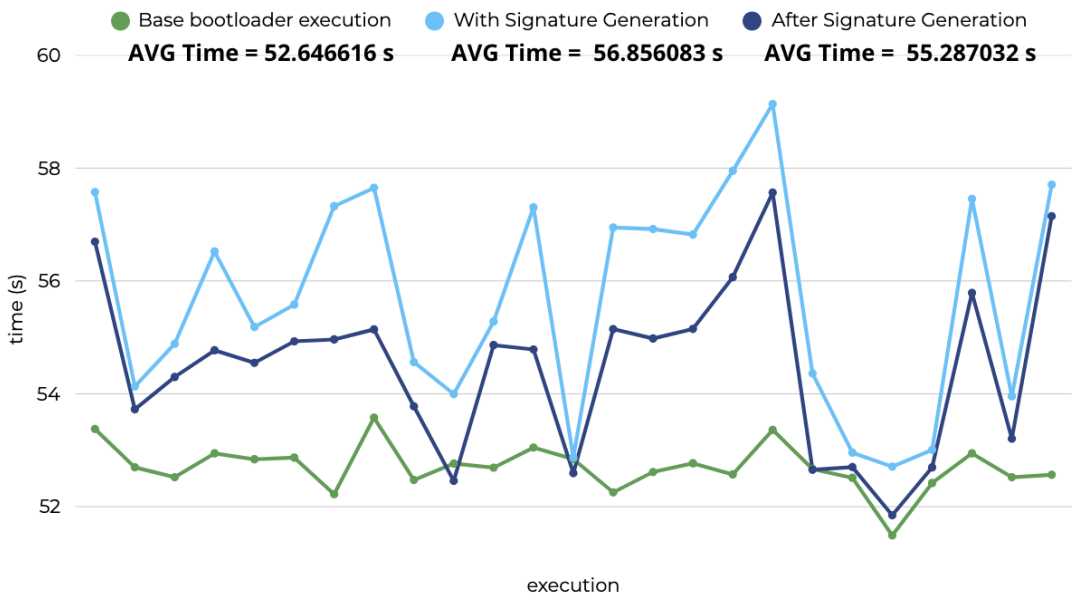


Figure 8.3: Bootloader Execution Time Comparison

The analysis of the experimental results highlights how the execution time of the base bootloader remains rather stable across the different runs, with an average value of about 52.647 s, while the activation of the security mechanisms, which in this case involve the verification of a digital signature through the ECDSA algorithm, leads to a measurable increase in duration that brings the average execution time to roughly 55.287 s (considered after signature generation since the image should be given already signed), corresponding to an increment of around 5% when compared to the non-secure configuration.
This increase, although evident in the data and clearly visible in the comparison between the two series of measurements, should not be considered excessive, since the use of hardware acceleration provided by the HSE allows the cryptographic operation to be performed in a relatively efficient manner, and the overhead that emerges appears to be contained if one considers the guarantees that such a verification introduces in terms of firmware integrity and authenticity. It is also relevant to observe that in the absence of dedicated hardware support the same verification would likely result in a much more significant penalty, as reported in the literature where purely software implementations of ECDSA tend to slow down execution to a far greater extent, therefore the value obtained here can be interpreted as a balanced compromise between the need to maintain an efficient flashing procedure and the necessity of integrating a robust security mechanism. In this sense, the additional few seconds that the secure bootloader requires do not represent a limitation to its applicability but rather a demonstration that security can be integrated into the operational workflow without undermining performance to an unacceptable level, making the observed increment not only tolerable but also fully justified in the light of the benefits it provides, especially if one considers that the cryptographic checks are executed entirely "on the ground" during maintenance or update phases, so the strict real-time constraints that apply to avionics applications during flight (which are defined "safety critical") are not impacted in the operational context considered in this work. Moreover, the presence of a dedicated Hardware Security Engine minimizes the computational load on the main core, and the adoption of a non-blocking execution strategy ensures uninterrupted communication over the CAN bus, reinforcing the idea that

the secure solution represents the most efficient trade-off between robustness and practicality, where the modest overhead is amply compensated by the stronger guarantees of integrity and authenticity that it provides.

8.3 Validation Outcomes

Validation was carried out in a controlled laboratory environment replicating a realistic firmware flashing scenario where three representative cases were tested:

- **Positive Case:** A correctly signed application image was successfully flashed and executed, with the bootloader marking it as valid and launching it at the next restart.
- **Tampering Attempt:** Modifying the firmware content after signing resulted in signature verification failure, with the bootloader preventing execution and awaiting a valid image.
- **Protocol Stability:** UDS sessions remained stable throughout the verification process, demonstrating the effectiveness of the non-blocking approach in preserving communication reliability.

8.4 Impact on the Attack Surface

The secured bootloader closes several vulnerabilities present in the original implementation. In the baseline system, any image passing the CRC check could be executed, allowing an attacker to load malicious firmware without detection. The new system enforces cryptographic authentication, making it practically infeasible to execute unauthorized code without access to the private signing key. Furthermore, by coupling verification with UDS SecurityAccess procedures, flashing rights are now bound to both protocol-level and cryptographic checks.
As a result, the attack surface is significantly reduced since tampering during firmware transmission is detected immediately, supply chain attacks are mitigated through mandatory signing and replay of outdated images can be prevented by extending the current design with version metadata checks.
Below there is a table which analyzes the threat conditions (previously identified in 4.4) whose effectiveness is reduced by the applied solution:

Threat condition	**CAPEC ID**	**Mitigation effect**	**Residual risk**
Authentication bypass of firmware source	115, 114, 112, 560	Only firmware signed with authorized private key is accepted via ECDSA/HSE_B.	Requires secure key handling.
Software integrity attack	184, 55	Post-signing modifications fail verification and are rejected.	Integrity verified at flashing; no continuous boot-time recheck yet.
Code injection during update	175, 242, 248, 240	Signing binds content; injected binaries fail verification.	UDS channel unauthenticated; DoS still possible.
Firmware package spoofing	148, 151, 154, 173	Signature overrides metadata; spoofed IDs ineffective.	Spoofed frames may appear valid at transport level.

Adversary-in-the-middle	94, 272	Any in-transit change triggers signature failure.	No confidentiality; DoS remains possible.
Privilege abuse of flashing capability	212, 122	Flashing requires SecurityAccess *and* valid signature.	Relies on robustness of seed/key exchange.
Supply-chain tampering	438 to 441	Altered firmware in distribution fails signature validation.	Cannot detect malicious code in legitimately signed images.
Message flooding to cause invalid state	125, 227	Non-blocking verification prevents UDS timeouts leading to false acceptance.	Service availability can still be targeted.
Manipulation of APP Validity Flag	165, 141	Execution depends on signature, not flag alone.	Hardening of flag storage still advisable.
Reverse engineering to craft binaries	188	Without private key, cannot craft passing image.	Protect signing infrastructure from key theft.

8.5 Limitations and Future Work

While the implemented solution meets the scope defined for this internship, further enhancements could strengthen security even more:

- **Startup Cryptographic Integrity Check:** Leveraging the HSE_B Secure Memory Regions (SMR) for cryptographic verification at every boot would provide persistent integrity assurance, complementing the flashing-time signature verification, ensuring that the application remains unchanged and authentic throughout the ECU's lifecycle, even after extended downtime or in the presence of physical access attempts.
- **Version Rollback Protection:** Enforcing metadata-based version control to prevent downgrade attacks, ensuring that only newer or equal versions of the firmware are accepted. This mechanism can be combined with monotonic counters stored in non-volatile secure memory to maintain the current version state even across power cycles.
- **Full Secure Boot Chain:** Extending signature verification to all boot stages, from the immutable first stage (ROM-based) to the final application, thus establishing a continuous hardware root of trust guaranteeing that no intermediate component can be altered to bypass subsequent verifications.
- **Anti-Tampering Protection:** Integrating tamper detection mechanisms such as voltage, clock, and temperature anomaly sensors, or intrusion switches that trigger immediate key invalidation or zeroization upon physical tampering attempts. The NXP S32K3 HSE_B includes anti-tampering features that can be configured through the `hseAttrPhysicalTamperConfig_t` attribute but Physical tamper detection can be enabled during initialization as a one time configurable attribute.
- **AI-based Detection:** Employing machine learning algorithms to monitor runtime parameters, firmware update patterns, and communication anomalies in order to detect deviations from normal behavior that may indicate an ongoing intrusion or exploitation attempt. Such systems could provide early warning and enable pre-emptive countermeasures, even for novel attack patterns.
- **Encrypted Firmware Updates:** In addition to signing, encrypting the firmware image would preserve confidentiality, preventing reverse engineering of proprietary algorithms or

sensitive data contained in the binary, even if it is intercepted during transfer. The HSE_B supports the management of multiple keys and the execution of cryptographic operations, enabling encryption and decryption using private keys and modern cryptographic algorithms as part of the update process.

- **Secure Debug Management:** Implementing strict control over debugging and programming interfaces (e.g., JTAG/SWD lock, authenticated debug sessions) to prevent unauthorized readout or modification of memory contents, especially in deployed units. The NXP S32K3 HSE_B provides a secure mechanism for this through a dedicated Application Debug Key Password (*ADKP*). This one-time programmable password is intended for deployment scenarios since once set, it cannot be altered or removed, which is why it was not applied in the context of this work.

Chapter 9

Conclusions

This thesis was conducted during an internship at *Avio Aero*, where the industrial setting allowed the research to be grounded in real hardware and operational constraints.
The work demonstrates that it is possible to integrate robust, standard-compliant security mechanisms into an avionic bootloader without violating performance or communication requirements and preserving safety. More importantly, it shows how dedicated hardware security features can be effectively leveraged to raise the baseline protection level in a safety-critical environment, while maintaining compatibility with existing workflows and industry standards.
Looking forward, the proposed approach offers a clear foundation for further improvements. Potential developments could include for example startup cryptographic verification, version rollback protection, a complete secure boot chain, anti-tampering mechanisms, encrypted firmware distribution, and AI-assisted intrusion detection. These advancements would further strengthen the resilience of the system against an evolving threat landscape and align it with future aerospace cybersecurity requirements.
In closing, this work provides not just a functional implementation, but also a methodology that can be applied to similar embedded platforms facing comparable security challenges. It stands as a step toward more resilient avionics systems, combining practical engineering constraints with a forward-looking security vision.

List of Figures

Bibliography

[1] «Teenager overpowered by plane passengers after allegedly boarding Jetstar flight with 'large gun' at Avalon airport». In: (). URL: https://www.theguardian.com/business/2025/mar/06/avalon-airport-melbourne-man-detained-alleged-firearm-gun-ntwnfb (cit. on p. 17).

[2] «Mumbai: SpiceJet pilots spot man on runway, alert security». In: (). URL: https://www.indiatoday.in/crime/story/mumbai-airport-man-on-runway-1590502-2019-08-22 (cit. on p. 17).

[3] «The Kano airport security breach, an example of a red flag». In: (). URL: https://tribuneonlineng.com/the-kano-airport-security-breach-an-example-of-a-red-flag/#:~:text=Shola%20Adekola,bound%20flight%20by%20two%20hours. (cit. on p. 17).

[4] «Just Stop Oil targets Taylor Swift's jet – and fails to locate it». In: () (cit. on p. 17).

[5] «Aeroporto di Bergamo, entra in pista e muore risucchiato dal motore di un aereo». In: (). URL: https://www.corriere.it/economia/trasporti/aerei/25_luglio_08/aeroporto-di-bergamo-entra-in-pista-e-muore-risucchiato-dal-motore-di-un-aereo-voli-sospesi-92e76ea5-102d-455a-880b-6db2428f6xlk.shtml (cit. on p. 17).

[6] «NXP, Unified Bootloader Documentation». In: (). URL: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/S32K/7138/1/UBLUG.pdf&ved=2ahUKEwjVy_3X8JaQAxWIgf0HHc7CJpMQFnoECCcQAQ&usg=AOvVaw0WokzaunB5Why48e2__8zR (cit. on p. 30).

[7] «ISO 11898-1:2015». In: (). URL: https://www.iso.org/standard/63648.html (cit. on p. 39).

[8] «ISO 15765-2:2016». In: (). URL: https://www.iso.org/standard/66574.html (cit. on pp. 39, 40).

[9] «ISO 14229-1:2020». In: (). URL: https://www.iso.org/standard/72439.html (cit. on p. 41).

[10] «ISO 15765-3». In: (). URL: https://www.iso.org/standard/33618.html (cit. on pp. 41, 44).

Acknowledgements

...