POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A novel open-source HSM Firmware compatible with AUTOSAR specifications for Secure Hardware Extensions

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Franco OBERTI

Candidate

Leonardo PALMUCCI

July 2023

Summary

Automotive control systems face security as one of their biggest challenges in the next few years. The race towards Vehicle-to-Everything (V2X) technology, while improving vehicle capabilities and enabling driverless cars, poses a significant security risk. As cars become increasingly interconnected, it is easier for malicious users to attack them by exploiting their enhanced vehicle communication capabilities. Moreover, these attacks can occur at various levels, ranging from the network to the physical layer.

In particular, one of the potential targets of a cyberattack consists of critical information to decrypt confidential data; such an attack could allow a malicious user to access information that should be kept secret, potentially enabling the attacker to gain control of the vehicle itself. For this purpose, key management systems such as Hardware Security Modules (HSM) address this issue by defining a secure area where these secrets are safe. However, most HSMs are deployed on specific boards with highly specific hardware requirements and features, making any software developed for such systems not portable; moreover, given the specificity of the target system, such software is neither available to reuse.

The present work describes a novel open-source HSM Firmware compatible with AUTOSAR specifications for Secure Hardware Extensions. By examining these requirements and the prerequisites to achieve an acceptable degree of security when using an HSM, this work explores in detail the design and development of firmware to deploy on a specific category of boards that can support HSM by hardware design.

Besides, the Thesis encompasses the issue of the intrinsic dependency of such a kind of firmware from the underlying hardware while considering portability as a crucial project requirement. Without this additional effort, trying to make the HSM firmware open-source would prove useless because it would suffer from low applicability. Particular attention has been focused on the interaction between the HSM, its underlying hardware and the external domain with which it communicates by implementing a suitable driver that acts as the sole interlocutor with access to this module by both software and hardware.

For validation, functional tests are first conducted in emulation and then by

porting the project on a target board, thus proving the portability of the firmware. Starting from the results of this Thesis, it would be possible for future developers to enhance the driver's capabilities and increase the set of supported boards by porting the current project to them, too.

Table of Contents

List of Figures

Acronyms

AES
Advanced Encryption Standard

API
Application Programming Interface

AUTOSAR
AUTOmotive Open System ARchitecture

C-V2X
Cellular Vehicle-to-Everything

CAN
Controller Area Network

COTS
Commercial Off The Shelf

CBC
Cipher Block Chaining

DSRC
Dedicated Short-Range Communication

ECB
Electronic Code Book

ECU
Electronic Control Unit

HAL

Hardware Abstraction Layer

HSM

Hardware Security Module

ISR

Interrupt Service Routine

MAC

Message Authentication Code

MPU

Memory Protection Unit

OSAL

Operating System Abstraction Layer

RTOS

Real-Time Operating System

SHE

Secure Hardware Extension

SoC

System-on-Chip

TEE

Trusted Execution Environment

TPM

Trusted Platform Module

V2I

Vehicle-to-Infrastructure

V2N

Vehicle-to-Network

V2P

Vehicle-to-Pedestrian

V2V

Vehicle-to-Vehicle

V2X

Vehicle-to-Everything

Introduction

If a single word were chosen to describe the industry's ongoing trend during the last 50 years, that would be *Digitalization*. From the 1960s, when Computer Science transformed from an academic topic to practical knowledge that could enable disruptive technologies, nearly every industrial sector has moved its steps towards the digital domain.

Enhanced efficiency, cost reduction and improved productivity are some of the advantages provided by this technology; the possibility to automate tasks and streamline processes while also reducing costs made the industry capable of concentrating its resources towards more profitable activities. In the meanwhile, employees could avoid spending their working hours on redundant or dangerous tasks and focus on how their organization.

Especially from the 2000s, the rise of the Internet of Things (IoT) has brought the concept of connectivity to sectors that were traditionally distant from the topic, enabling features like inter-product communication, data retrieval and analysis and remote control that were impossible up to a decade before. Eventually, all these characteristics have an ultimately desirable effect: improve the user experience.

The automotive sector does not represent an exception to this pattern. Since the development of the first Electronic Control Units in the 1970s, the automotive industry has continuously embraced digitalization. Recent advancements in technologies such as Autonomous Driving and Vehicle-to-Vehicle (V2V) communication, which allows vehicles to exchange information and collaborate on the road, further affirm that this trend will continue to accelerate in the coming years. The integration of V2V communication enables enhanced safety, improved traffic management, and more efficient transportation systems. Market analyses project a significant growth in the market size of autonomous vehicles, with expectations of tripling between 2021 and 2030 [1].

This data demonstrates how the development of the automotive industry is strictly intertwined with these features, to the point where their absence leads to the inevitable obsolescence of the product regardless of all its other qualities.

Despite the positive effects of these new technologies, they also imply new

challenges and risks that must be handled. This is especially true for a safety-critical industrial segment such as the automotive domain, where the most naive error can lead to catastrophic consequences. Given the strong effect that V2V technologies have on the evolution of the automotive sector, it is interesting to explore which are the main drivers of this evolution and what are the challenges the industry is currently facing, with a particular focus on Security, which is the main topic of the current Thesis.

After reconstructing how the automotive sector has moved towards the digital domain by considering the Vehicle-to-Everything (V2X) communication system, this Thesis aims to highlight the security criticality that this technology implies. A special focus is given to key management systems, i.e. modules whose purpose is to protect secrets to be used to maintain a reliable level of confidentiality and integrity during the ongoing critical communication inside the ECU, thus providing a solid secure foundation to the system on which they are implemented.

The current Thesis briefly explores the main solutions that have been implemented over time to fulfil the strong security requirements imposed by these modules, such as Trusted Platform Modules, Trusted Execution Environments and Hardware Security Modules. AUTOSAR, one of the reference partnerships in the automotive sector, has paid close attention to today's vehicle security issues and has defined several standards specifically for this industry [2][3][4][5], ultimately creating a software framework for vehicle security.

Starting from this framework, it is possible to define security-critical components such as key management systems that are both compliant to AUTOSAR standards and provide an inherently high level of security. For this reason, this Thesis focuses on the design and development of Hardware Security Module that follows specifications provided by AUTOSAR and is structured as follows:

The topics of this Thesis are structured as follows:

Chapter 1 provides an overview of the automotive sector and its path towards V2X, together with the challenges that this technology implies in terms of Security and how to solve them, up to describing why Secure Key Management Systems are needed.

Chapter 2 explores the open points of the previous chapter and describes how a secure key management system could be implemented by reporting the current state of the art; in the end, a special focus on how AUTOSAR addresses the issue is reported.

Chapter 3 describes the current work consisting in the HSM Firmware project, whose purpose is to design and develop a portable and configurable HSM to deploy on a certain category of ECUs; in particular, this chapter delves into the details of the preliminary requirement analysis for the project

Chapter 4 is the core chapter of the work; it describes and illustrates the design of the HSM Firmware project, its principles and mechanics.

Chapter 5 covers how the project has been developed and tested, eventually reporting the final results and suggestions for future improvements of the firmware.

Chapter 1

Background

Nowadays, automobiles are one of the products that represent the current times: cities, towns and even the countryside are brimming with cars; every family possesses at least one of them and there is a wide variety of choices in terms of size and formats, which proves there is no target that can be considered out of reach for the automotive industry.

The dominance of the automobile as the principal mean of private transport is now taken for granted, but this has not always been the case. In fact, during its early development stages, automobiles needed to compete and overcome horse-trained carriages; eventually, this required designing vehicles providing better performances at an affordable cost.

For this reason, early manufacturers focused their attention on improving the performance of their vehicles while making production sustainable and scalable, eventually leading to the rise of this industry segment.

At that time, safety was not a concern, at least until accident rates became alarming. For instance, in the US, there were about 18 deaths per million Vehicle Miles Travelled (VMT) in 1925, much higher than the about 4 deaths per million VMT experienced 70 years later [6].

Together with the absence of safety regulations for the newborn automobile, other reasons behind this high fatality rate were linked to the absence of safety considerations in the design and development stage and the inexperience of the drivers [6]. This led government agencies to regulate vehicle circulation and manage licenses while enterprises, for their part, addressed the issue by designing safety subsystems like seat belts and concentrating on the mechanical stability of the vehicle, also thanks to the experience gained in the previous years.

The rise of electronics in the 1960s, when they became affordable for mass production, represented another step forward for car design and development in terms of safety; in fact, the possibility to embed automated control systems inside the first Electronic Control Units (ECUs) paved the way to new potential features

and improvements. Anti-Blocking Systems (ABS), Adaptive Cruise Control (ACC) and Electronic Stability Control (ESC) are some of the main achievements in the automotive field that connect electronic control systems to the underlying mechanics. From the first experimental models decades ago, these subsystems have progressively become standard components of modern vehicles, such that they are currently embedded in any model and are strongly regulated.

In general, these improvements have a twofold purpose: first, they enhance driver safety; in fact, fatality rates in car accidents have drastically decreased over time [6]; as a consequence, the reputation and stable presence in the market for the manufacturer is guaranteed. Safety is not only a high priority for the driver but also for the manufacturer. A single failure, even the most naive, may result in driver injury, with devastating consequences. Secondly, the end user can benefit from better control of the vehicle, which eventually leads to a better User Experience. Parking Assistance Systems are a clear example: the driver can leverage them so that they do not need a passenger in those contexts where parking is not trivial. Instead, they can rely on a *trusted* supervisor to park safely. It is important to notice that the driver must be confident of the vehicle and its assistance systems on these occasions, and this trust is easy to fall with the slightest of errors.

This is especially true when the driver should trust their car to the point that it runs without human input, with the only exception of the target destination to reach: Autonomous Driving is the culminant point of this evolution, where vehicles transform from passive locomotion systems into reactive and intelligent products that can interact with the surrounding environment in complete autonomy.

To enable this step, everything meaningful for the driver, from pedestrians to traffic lights, from approaching vehicles to traffic conditions, must live in the same ecosystem.

1.1 Towards Vehicle-to-Everything

The idea of a common ecosystem where automobiles can interact with other smart systems and assist each other by sharing information is not new and is known as Vehicle-to-Everything (V2X). This umbrella term is a composition of several different interactions:

Vehicle-to-Device (V2D) Here, devices are intended as applications or systems that can be used inside the car, e.g. Apple's CarPlay; in this category, we can include features that enhance the User Experience, e.g. supporting Bluetooth communication to connect a smartphone to the vehicle.

Vehicle-to-Grid (V2G) The possibility of exchanging information with the electrical grid to reduce its stress by transforming the vehicle into a power provider when it is not in use for transportation [7].

Vehicle-to-Network (V2N) By enabling access to cellular networks, vehicles become capable of interacting with any other actor in the same ecosystem; the following three definitions are subcategories of V2N:

Vehicle-to-Infrastructure (V2I) The interconnection of elements of the road infrastructure, e.g. traffic lights, with its users. This link unlocks a smart usage of the road network such as Dynamic Traffic Control Systems, which modify the semaphore timing depending on traffic conditions at the intersection, thus improving vehicle circulation and reducing fuel consumption.

Vehicle-to-Pedestrian (V2P) Pedestrians may be equipped with devices that can help the car recognize them at intersections and prevent accidents, while pedestrians may actively use their devices to monitor traffic conditions and optimize their route.

Vehicle-to-Vehicle (V2V) Last but not least, vehicles can exchange information and share their position to keep a safe distance and detect potential collisions; this interaction does not need a surrounding infrastructure as it only requires that the vehicles can communicate by themselves and may be useful in regions where road infrastructures are poor.

Another used term is Cooperative Intelligent Transport Systems (C-ITS), even though it refers to the principle of making all the actors of the road network, e.g. pedestrians, road infrastructures and vehicles, communicate with each other [8], while V2X focuses on the communication capability of the car with the environment. The goal is to define an ecosystem where all participants can contribute by improving the driving experience of the others, both in terms of safety and in terms of performance.

For instance, a suggestive feature that V2X wants to achieve is smart traffic management: if the road infrastructure were able to monitor the local traffic conditions by extracting this information from the vehicles involved, it could suggest which roads are less crowded and encourage the driver to take them; another option would be to adjust the timing of traffic lights so as to reduce queues on busy roads.

Features like this need a complex and coordinated network among all vehicles as shown in fig. 1.1; building and maintaining this network is one of the main goals of V2X.

V2X is an ongoing process, even though it is possible to find the first traces of work on communication projects between vehicles to increase safety, reduce accidents and help the driver to the 1970s with projects such as US's Electronic Route Guidance System (ERGS) and Japan's CACS [9].

ERGS represented the first attempt at a Vehicle-to-Infrastructure (V2I) communication system, a step towards an ante litteram intelligent transportation system.

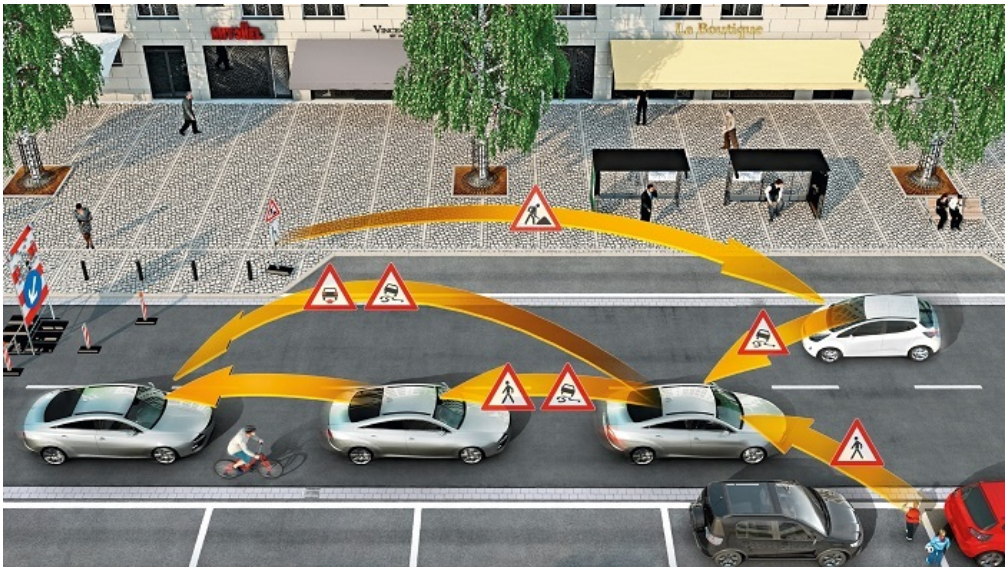

Figure 1.1: A visual representation of V2X (Source: *everythingrf.com*)

This peculiar technology relied on a destination-oriented system where the user could receive routing information to reach a desired destination by selecting it inside the car; this request would be forwarded through a network embedded in the road infrastructure up to the target point, where a local node could process the request and return instructions to reach the destination [10]. It is noted that this technology, ahead of its time, required a solid infrastructure to support it, as by definition of a V2I system.

It is with the definition of wireless networks and, in particular, of IEEE 802.11 that V2X starts to form: in particular, the IEEE 802.11p amendment, approved in 2010, allowed to include Wireless Access in Vehicular Environments (WAVE) to Wi-fi connections [11]. During that period, wireless connections became increasingly affordable and widespread: it was time for the road infrastructure to benefit from such development and evolve. Wireless communication technologies like Dedicated Short-Range Communications (DSRC) represent the starting point of modern V2X.

DSRC is a wireless communication technology designed for short-range and high-speed communication between vehicles and infrastructure, used in Intelligent Transport Systems (ITS) to enable Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communication [12].

DSRC enables direct communication between vehicles and Roadside Units (RSU) such as traffic lights, toll booths, parking management and surveillance systems, weather monitoring stations and so on [12]. In this way, vehicles can exchange safety-related information in real-time, including real-time vehicle speed, position, acceleration, and braking data.

This technology provides several key features. First and foremost, it enhances road safety by enabling cooperative collision avoidance systems. Vehicles can communicate their positions and intentions, allowing for early warning and collision mitigation. Besides, it enables cooperative traffic management by providing real-time information, so that drivers can optimize their routes and reduce travel time.

However, DSRC has challenges to face. First, since DSRC does not rely on preexisting networks, it requires a massive installation of compatible RSUs in the road infrastructure to work properly, which implies investments, costs and time. For this reason, other technologies that can leverage the current telecommunications infrastructure have emerged in the following years.

Cellular V2X (C-V2X), developed within the 3rd Generation Partnership Project (3GPP) [13] in three releases around 2016, aims to use cellular networks rather than Wi-fi. Besides, this communication system plans to extensively support 5G, thus enabling Vehicle-to-Network, which DSRC does not support. Finally, C-V2X expects to be used for direct communication so that it can work regardless of the quality of the surrounding infrastructure, which can prove useful when the latter is not available, such as during natural disasters.

This communication system defines two interfaces: PC5 for direct interaction between vehicles, useful when low-latency communication and fast coordination is the priority, and UU to connect drivers to the cellular network via their User Equipment (UE) so that they can use base stations to interact with farther nodes of the network [14].

These two technologies are the main pioneers in the development of the vehicle-environment interaction, although it is not clear which of the two is preferable at present: although there exist studies that emphasize the better coverage, latency, reliability and scalability capabilities of C-V2X [13], assuming that DSRC is not suitable for the development of this technology, others refute the same findings and attribute greater maturity to DSRC [15] [16]. Both C-V2X and DSRC have been assigned to the same frequencies around 5.9 GHz, designated for the Intelligent Transport System (ITS) [17]; this further fuels the debate as to which of the two systems should be better supported [13].

Regardless of the direction that V2X will follow from a technological point of view, smart vehicles and infrastructures are the current trend and are gaining interest and a growing slice of the market [1]. However, there are noticeable challenges to overcome to make this technology stand the test of time. The rise of AI and Edge Computing, for instance, requires stronger computational capabilities on the vehicle that should not always rely on underlying Cloud infrastructures. Other features, such as Crowd Sensing, i.e. the capability to exploit a large number of users for retrieving more complete data, need to preserve the users' privacy and be effective at the same time [9].

Besides, any solution to these challenges must satisfy the same unavoidable requirements. Safety is a core priority: providing no harm to the driver is the primary measure of the success of the technology; in case a life-threatening scenario ever occurs due to a vehicle failure, drivers will trust the car and the manufacturer much less, with great costs and reputational damage.

This becomes increasingly true in a future where intelligent vehicles are the common case: for the car to be profitable, the human user must rely on the fact that it is *intelligent enough* to be trusted.

However, safety is not the only concern; in fact, other two kinds of services of interest in the race towards V2X can be identified: non-safety services and infotainment [9]. While the latter services provide features such as Internet Access and video streaming, aiming to improve user comfort, non-safety services try to optimize traffic management and maximize the efficiency of the road network [9].

Another difference lies in their performance requirements; even though non-safety services are not the most critical, they still require sufficiently low latency to achieve optimal results. For instance, traffic management systems should always provide the latest updates about traffic congestion. Instead, infotainment does not need to be reliable: the user can tolerate short intervals of low-quality service or even unavailability.

Going back to safety, there is another non-functional requirement that is highly correlated and equally important: Security. A system that applies any existing safety measure cannot be claimed as a safe system: if an ill-intentioned user can penetrate the system and affect its behaviour in a relevant way, it can become dangerous and thus unreliable.

1.2 Needing Security

Ever since the inception of Vehicle-to-Everything (V2X), security has emerged as one of the foremost challenges in the automotive domain. While granting vehicles access to a global network unlocks numerous innovative features, it also subjects them to external vulnerabilities. With the growing proliferation of smart cars on the roads, this exposure is amplified, providing attackers with a ripe opportunity to capitalize on the increasingly profitable automotive industry. The combination of low-security protection and the abundance of valuable data available makes the automotive domain a lucrative target for malicious actors.

1.2.1 The pillars of Security

Security is a multifaceted problem: various properties make a system secure, implying several directions along which an attacker could move. Besides, attackers do not need to know every possible weak point of a system to attack it successfully;

one vulnerability, if not handled correctly, is more than enough. These reasons highlight that Security is an intrinsically difficult problem to solve; thus, it is needed to define a taxonomy of the features that a secure system must support and define attack models.

Despite the complexity of the topic, over time the fundamental properties that a secure system must possess have been defined, such as the CIA Triad (Confidentiality, Integrity, Availability) [18] which is referred to as the basic set of Security properties to grant to recognize whether a system is secure.

In addition to these fundamental properties, however, it is necessary for a system to support others. For example, as much as a system is capable of encrypting any communication that takes place within it and thus has a very good level of confidentiality, the lack of an authentication system would allow anyone to interact with the system, which would be risky. For this reason, both the definitions of the CIA Triad properties and other properties that are now indispensable in a secure system, including cars [14], are given below.

Authentication The capability of a system to recognize and identify who is requesting a given resource or communicating over a network. It can be split into two parts [14]: *User Authentication*, ensuring that the user that needs a resource is legitimate, and *Message Authentication*, identifying who has sent a certain message in the network. *Non-Repudiation* is strongly related to Message Authentication and consists of the capability to recognize who has performed a given action unambiguously; this feature is handy to solve disputes where an author of misbehaviour must be identified.

Authorization The capability of a system to recognize whether a user is allowed to use a given resource; this feature is required to define privilege levels and filter resource access; it requires Authentication as a prerequisite.

Availability The capability of a system to be up and running whenever required; depends on the type of system involved. This feature is crucial for vehicles from the moment the engine is turned on; in general, this is a fundamental feature for any real-time system where missing any deadline could lead to failure.

Confidentiality The capability of a system to preserve the secrecy of sensitive data such as private keys to prevent access from non-authorized users and information leakage; usually, it is achieved by adopting cryptography [14] both to hide precious content sent over an insecure channel or to protect data from physical access.

Privacy Related to Confidentiality, but with a focus on the personal identity of the users and the secrecy of their sensitive data. Anonymization or information

hiding are two of the possible strategies to cover the user identity while allowing them to access the network [14]. In principle, this feature can conflict with Authentication, which requires a user to be easily recognisable; so, the system should be able to identify who is performing some actions while hiding their identity away from prying eyes.

Integrity Given that a user can recognize who sent them a message thanks to Authentication, it is not ensured that what has been received corresponds to what has been sent. Attacks such as Man-in-the-middle (MITM) aim to place attackers between two or more users, to sniff data coming from the user, alter it and send it to the expected receiver. It is noted that such an attack would also violate Non-Repudiation as the original sender would be blamed in case something happens. Typically, message signatures are used to detect whether the content has been corrupted.

Creating networks of vehicles that can communicate and exchange information implies exposing them to an unsecured network as this increases their attack surface. Since this side-effect is unavoidable, countermeasures must be taken to achieve a reasonable level of Security, which is particularly high for automobiles because they are safety-critical systems.

Despite all precautions, vehicles are still vulnerable: an experiment conducted in 2016 by two experienced hackers bypassed the safeguards of a Jeep Cherokee by using a laptop directly plugged into the Jeep's CAN network via a port under its dashboard [19]. By carefully crafting CAN frames over the bus, the two hackers could control several subsystems while the driver could not counteract. Causing unintended acceleration, slamming on the brakes and taking control of the steering wheel as the two researchers managed to do is extremely dangerous and can prove fatal.

The perpetrators of this attack already did something similar the previous year [20]; the scale of the news was such that Chrysler was forced to recall 1.4 million vehicles to apply the necessary Security patches, at considerable cost and embarrassment to the company.

Moreover, once a vulnerability has been discovered and the strategy to penetrate the system is publicly available, an increasing set of attackers can exploit it in turn and profit from it. This proves how the entire automotive industry is compelled to deal with Cyber Security since there is no room for errors when the driver's safety is at stake.

There are various strategies adopted at the level of state, international and industry organizations to address these critical issues. For instance, public databases like the Common Vulnerabilities and Exposures (CVE) system provide reference methods for publicly known information-Security vulnerabilities and exposures, even though CVE is not specifically defined for this industry sector [21]. Sharing

this kind of information improves manufacturers' awareness of possible defects in their products so that they can apply appropriate countermeasures.

There are also organizations whose purpose is to provide information on safety-related defects and also on vehicle recalls or investigations, such as the National Highway Traffic Safety Administration (NHTSA) in the US [22].

1.2.2 Security measures on C-V2X and DSRC

V2X-enabling technologies like C-V2X and DSRC have defined Security measures, too [14]. In particular, DSRC adopts cryptographic standards for establishing trust and preserving confidentiality between communicating parties. In this way, attacks based on eavesdropping can be prevented. Cryptography can also support authentication if the two parties involved possess a shared secret, in the case of symmetric cryptography, or if there is a certificate management system that can unambiguously map certificates to users, in the case of asymmetric cryptography.

Specifically for DSRC, there is a certificate system where senders are restricted to specific geographical regions, so that the number of nodes that can communicate with a given node is limited, thus improving Security. In fact, it is unlikely that a node would send a message to a remote receiver; by limiting the certificate range, it is easier to filter out malicious nodes.

Interestingly, DSRC defines safety-related messages in plain as encryption and decryption are length operations, whose latency could violate the real-time required latency for these messages [14]. Instead, messages containing sensitive data are ciphered.

To prevent information leakage that could violate users' privacy, DSRC implements anonymization to preserve the identity of a certain node; only the Certificate Authority (CA) is authorized to retrieve sensitive data from a user and is strictly monitored and strengthened to be more resistant to attacks [14]; penetrating a CA successfully would annihilate any anonymization technique.

Since the lower layers of C-V2X are based on LTE, the former leverages the same Security measures as the latter, whereas the upper layers use the same measures as defined for DSRC [14]. In particular, there is no specific recommendation about how to handle sensitive data in C-V2X. However, authentication and authorization are handled using V2X Control Functions, i.e. control systems that are responsible for connecting both the sidelink interface (PC5) and the interface with the cellular network (UU) by verifying authentication and authorization [14].

Interestingly, the protocol also defines one-to-many encrypted communication, where the key is shared among all the participants and is derived from a Group Security Key. Regarding privacy, C-V2X does not strictly demand specific Security measures, which are left to the regional operators. 5G-V2X, the version of

C-V2X supporting 5G, implements further Security functionalities, such as cryptographic keys on multiple layers e.g. session keys, unicast link keys and long-term credentials [14].

1.2.3 Security standards and their implications

Although V2X-enabling technologies such as C-V2X and DSRC apply strategies to improve their Security, this is not enough. This is because the network is not the only means available to conduct an attack: in fact, the one conducted in 2015 and reported in [19] acted on the physical layer as it involved a CAN Bus.

It follows from this that to achieve an acceptable level of reliability, it is necessary to think about the entire system and how to secure it at a structural level; this concept is known as "Security by design". The automotive industry is aware of this need and has therefore invested in the development of Security standards that are adoptable by manufacturers and cover all phases of vehicle development [23].

In 2016, the Society of Automotive Engineers (SAE) proposed a preliminary guideline to establish high-level principles for improving Security in vehicles: SAE J3061 [24]. This guideline does not delve into details but rather focuses on general Cyber Security principles and how to port them to vehicles, which have unique challenges to address.

In particular, the guideline aims to define a new Security framework that should act as a baseline for any future vehicle development. It focuses on various areas of the manufacturing process, from production to development, from techniques to human actors involved in the procedures.

At the time, there already existed safety standards such as ISO 26262 [25], recommending techniques that will be used by other standards, such as Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) to be performed during the Risk Management and Analysis phase of the vehicle development. These are two techniques used to determine the possible sources of failure, the likelihood that they may occur and the potential results from a system perspective so as to estimate their criticality; such techniques can be used during Safety Risk analysis to link vulnerabilities to failures and estimate their criticality, too. However, they could also be used during Security Risk Analysis by following a similar approach.

However, such standards did not cover vehicle security. So, the guideline tried to adapt such existing techniques and embed them into a new Security framework.

In fact, SAE J3061 strongly recommends performing an initial assessment of potential threats and an estimation of risks for any systems that may be considered Cyber Security relevant or safety-critical [23].

The guideline also encourages the application of consolidated secure technology; common encryption algorithms, certificate and digital signature techniques, known methodologies and security concepts not only are faster to implement and maintain

than custom solutions, but they also have the merit of being validated for a long time and therefore are reliable.

Finally, employees are an integral part of the process of improving product security: no matter how seemingly inviolable a vehicle may be, it can still be accessed if someone leaves the keys attached to the door. Data breaches and malicious insiders can cause serious damage without having to interact with the product. For this reason, SAE J3061 also insists on awareness and training of workers and stakeholders [24].

Moreover, it recommends defining specialized personnel for continuous monitoring and threat intelligence to detect and respond to Cyber Security incidents effectively, together with the establishment of incident response plans and procedures to address and mitigate the impact of Cyber Security incidents. It is noted that these actions must be carried out at the company level and are not strictly technical.

This partition of the guidebook highlights what the three pillars of enterprise-wide Security are: people, process and technology, as succinctly shown in Fig. 1.2.

Figure 1.2: The three pillars of enterprise-wide Security (Source: *clounomy.com*)

After SAE J3061, ISO and SAE collaborated on the development of a new Cyber Security standard for the engineering of road vehicles [23], which resulted in the

publication of ISO 21434 in 2021 [26]. This standard had similar objectives in comparison with SAE J3061; in fact, the purpose of the standard to be created was to define a structured framework to ensure Cyber Security engineering of in-vehicle systems, thus reducing the likelihood of a successful attack and, consequently, of losses. It also aimed to provide clear means to react to Cyber Security threats across global industry [23].

Although SAE J3061 and ISO 21434 are very similar in terms of objectives, ISO 21434 has a holistic approach that seeks to cover every project phase; moreover, unlike its predecessor, it seeks to provide applicable tools and methodologies rather than providing general guidelines that are more focused on specific security threats. Moreover, ISO 21434 is designed to be integrated with other automotive industry standards, such as ISO 26262, which deals with Functional Safety and ISO 9001 handling Quality Management.

It is noted that since both standards derive from pre-existing Safety and Security frameworks, which were not defined especially for the automotive industry, they still need adjustments to address the unique challenges of the sector. For instance, vehicles can suffer from Man-at-the-End attacks [27], where a malicious user attempts to sabotage the same system it is connected to, as happened in [19] when the researchers were physically attached to the vehicle and were able to hack it remotely.

An analysis conducted in [27] highlights how ISO 21434 considers network-level attacks as more likely than at the physical level. However, it is more challenging to inject malicious code and messages with ECUs that do not support Firmware-on-the-Air (FOTA) updates, for instance. Instead, being able to gain physical access to a vehicle is easier compared to other types of systems, thus the likelihood of a physical-level attack is higher.

Several insights can be drawn from these two standards. In general, there is a strong necessity to improve the Cyber Security focus during the entire development process rather than limit it during code development and testing. Risk Management Analysis tools must be used for detecting Security Risks, too; in particular, they should be linked with Safety risks to evaluate how critical they are. Special attention must also be paid to human resources, a real and integral part of the development process, who must be properly trained.

Last but not least, both standards insist on the concept of "Security by design", a proactive approach to system and software development that emphasizes security considerations during the entire product lifecycle, including design. In fact, assuming that Security must be embedded in the entire development process, design is a critical phase where the system can either strengthen its resistance to attacks or completely lose it. For this reason and with a view to defining a secure system by design, it is essential to define secure architectures with trusted components and subsystems.

1.2.4 Secure Architectures

Nowadays, any system that needs to fulfil basic Security requirements leverages consolidated techniques, with proven effectiveness and lower implementation cost with respect to custom solutions. Moreover, these solutions are independent of the application domain, making them suitable for any product, vehicles included.

In the following, some of the key Security techniques used in V2X communication are reported, together with an explanation of which side of the CIA Triad (Confidentiality, Integrity, Availability) they cover:

Cryptography Assuming V2X communication takes place over an unsecured channel, sharing plain text messages is not an option, especially if it is needed to deal with sensitive data. Encrypting the information to be sent using a secret makes communication confidential without blocking it. If the secret is shared and known a priori, before the connection is established and active, we refer to *Symmetric Cryptography*. On the other hand, *Asymmetric Cryptography* allows for encrypted communication without possessing any secrets. It is based on public-private key pairs such that only the owner of a certain key pair can interpret messages intended for them. Often, asymmetric encryption is used to generate the keys to be used for symmetric encryption.

Digital Signatures and Certification Systems Even though messages are correctly encrypted, attacks such as Man-in-the-Middle can still harm by corrupting message data, thus violating Data Integrity. To detect whether this happens, a sender can use a Digital Signature to validate the content of a message. These signatures consist of hashes computed using cryptographic hash functions using the target message and a key; they are used by recomputing the hash for that message at the receiver node and checking if they correspond, given that the receiver knows the key and the algorithm to use. Typically, the key to be used is a public key of the sender node. Certificates are a special type of keys, part of a common Public Key Infrastructure (PKI) defined by Certificate Authorities (CA) so that it is possible to link a key to a specific user, which can be identified and trusted.

Authentication/Authorization procedures In a network where nodes do not possess the same privileges and must be identifiable, it is necessary to refuse requests coming from unknown or unauthorized nodes. Challenge-based authentication procedures allow to recognize whether a requesting user is actually who claims to be by checking their certification, thus enabling Authentication and Non-Repudiation. Once the requesting user has been identified, it is possible to decide whether their demand is valid, which enables Authorization. Finally, filtering out requests from all unauthorized or unidentified users allows

to refuse malicious users, thus preventing Denial of Service (DoS) attacks and improving the overall Availability of the system.

Random Number Generation Whenever unpredictable data needs to be generated, Random Number Generators can help by providing values whose distribution in the output domain is uniform, so that any value is as likely as the others. They can be classified in either *Pseudo Random Number Generators* in case randomness is injected using deterministic algorithms, thus providing a number that is random only apparently, and *True Random Number Generators* in the case where the number is random because the physical process generating it is random itself, e.g. thermal noise.

Except for the last feature, the others have a common requirement: it must not be possible to deduce keys without cracking what uses them, such as cryptographic algorithms or certification systems. If keys were easy to infer with a low effort from the attacker side, all the systems relying on them would easily fail. It is noted that the strength of the keys, hence their unpredictability, is taken for granted by these systems. Typically, high-quality Random Number Generators are used to guarantee that keys cannot be found with low effort. Moreover, it must be difficult for a malicious user to retrieve a key from where it is stored. Ideally, keys should be used without ever being visible to guarantee maximum anonymity.

So, key generation and handling have a major impact on the effectiveness of the entire Security infrastructure of the vehicle. For this reason, there exist specific modules whose purpose is to create keys, store and distribute them carefully. In fact, not only keys must be unpredictable and almost random, but they also need to keep their secrecy. Secure Key Management systems are the submodules deployed in current vehicle ECUs to achieve this goal.

In the following chapter, the principles of Secure Key Management systems are reported, together with various implementations known in the literature. Starting from ch. 3, an implementation based on AUTOSAR's specifications for Secure Hardware Extensions [4] is provided; this project is the main focus of the present Thesis.

Chapter 2

State of the Art and Technical Background

Considering the complex nature of vehicles and the ever-expanding network of actors within the global road network, it becomes apparent that vehicles cannot be deemed completely secure systems. With the growing number of individuals involved, the risk of malicious users seeking to exploit vulnerabilities also grows.

These nefarious actors employ various techniques, ranging from manipulating network communications by injecting harmful messages to physically tampering with crucial components [27]. As can be noted, the attack surface of vehicles is extensive, encompassing numerous entry points that can be targeted; the complexity of the entire system leaves it inherently vulnerable to potential breaches. As a consequence, any component of the system cannot be trusted by nature, if it does not implement high-quality security features by design.

Unfortunately, there exists sensitive information related to the user, the system or the vehicle as a whole, that must be protected and entrusted to specific components. For this reason, defining a *Secure Zone* where this data can be reliably stored with a very low risk of being stolen is not only a Security feature but an inescapable requirement.

This goal implies two sub-objectives to be achieved, as reported in the following:

- Preventing critical stored data from being stolen; expected attacks do not only consist of accessing this information via software since there exist also side-channel attacks, which can be countered only by using hardened hardware or by encrypting stored data.

- Using this critical data to ensure a sufficient level of security within the system. Trivially, preventing any access to such data at all would guarantee that this

information is never leaked; however, this approach would make this data unavailable to the entities that need it.

Finding the perfect trade-off between the two is not trivial and there exist various solutions to address this issue. Indeed, different strategies exist to define secure zones within embedded systems such as vehicular ECU; these solutions are explored in the current chapter.

2.1 Trusted Execution Environments

Trusted Execution Environments (TEE) are secure computing environments designed to protect sensitive applications and data from unauthorized access and malicious activities. Typically, they establish a trusted zone within the main processor, leveraging a combination of hardware and software security mechanisms, with a particular focus on the latter.

Essentially, TEEs provide secure enclaves that separate the trusted domain of the system, making it possible to store and use critical data even in untrusted systems [28]. These secure enclaves host isolated memory and resources that cannot be freely accessed; instead, specific implementation-defined protocols must be followed by any task that is interested to use them.

All these security procedures are monitored in real time by the underlying Operating System, which is also responsible for the separation between the trusted zone and the external domain and is, ultimately, the core component that enables the TEE.

To interact with the trusted domain, the Operating System provides an API for interacting with trusted applications, enabling secure communication and data exchange, while hiding the internal details. In this way, the Operating System enforces access control policies, authenticates users and applications, and protects against various threats, including malware, code injection, and unauthorized access.

It is worth noting that the Operating System must be trusted or must be paired with a trusted TEE kernel that shall manage the secure execution environment while facilitating secure interactions with the main operating system and applications.

Typically, the Operating System is not sufficient to implement a Trusted Execution Environment on its own, since additional security shall be provided by the underlying hardware to support the trusted software so that it can fulfil its objectives in terms of security. For this reason, most of today's commodity CPUs implement forms of TEEs [28]. As an example, ARM processors implement a so-called ARM TrustZone that defines two separate domains, the Normal World and the Secure Worlde [29]. A secure Operating System shall handle the communication between these two domains while with hardware designed specifically for

security, such as functional blocks to provide security services, e.g., encryption, or an additional processor that only handles operations within the Trusted World.

Notably, using separate processors for the two domains is a recurrent solution and is also used for Hardware Security Modules.

2.2 Trusted Platform Modules

Trusted Platform Modules (TPM) are specifically designed to instil trust in computing platforms, encompassing a broad range of security features and functionalities [30]. As established by the Trusted Computing Group, TPMs adhere to a standardized interface and service framework, ensuring compatibility and interoperability across diverse systems.

These security platforms encompass a comprehensive set of subsystems that offer essential high-level features, fulfilling the requirements of various security-critical industrial domains, including but not limited to the automotive sector. By integrating TPMs into computing architectures, organizations can fortify the security posture of their platforms, establishing a foundation of trust and enabling robust protection against a wide array of threats and vulnerabilities.

While Trusted Execution Environments (TEEs) offer a more expansive range of functions and capabilities compared to Hardware Security Modules (HSMs) and Trusted Platform Modules (TPMs) [28], there are core security services that all three technologies commonly provide. These include fundamental features such as secure key storage, which ensures the confidentiality and integrity of cryptographic keys, and memory protection, which safeguards sensitive data residing in memory against unauthorized access or tampering. In the context of TPMs, in addition to these critical security functions, the TPM specifications define a comprehensive set of security-related functionalities that must be supported in order to achieve compliance with TPM standards.

These operations encompass key functionalities like MAC generation, RSA-based asymmetric encryption, cryptographic hash functions such as SHA-1, and random number generation [30]. Without these operations, the system cannot guarantee the execution of critical tasks within a trusted environment. While TPM specifications outline the minimum requirements for these operations, they also allow for the inclusion of additional components and functionalities. For example, the specifications permit the use of established algorithms like AES for symmetric cryptography, enabling broader support for encryption and decryption. This flexibility accommodates evolving security needs and empowers TPM implementations to meet a wide range of cryptographic requirements, reinforcing the overall security and trustworthiness of computing platforms.

In most desktop computers, Trusted Platform Modules (TPMs) establish their

secure enclave by leveraging separate hardware components distinct from the main processor and peripherals [28]. This approach ensures the establishment of a trusted domain within the system architecture. To achieve this objective, TPMs often rely on Hardware Security Modules (HSMs) [28], which are discussed in detail in the subsequent section, to facilitate the creation of this secure zone at the hardware level. Nevertheless, it is important to note that firmware-based TPM implementations also exist, presenting an alternative approach to realizing the functionalities and security guarantees of TPM technology [31].

2.3 Hardware Security Modules

Hardware Security Modules (HSM) provide an alternative solution to TPMs and TEEs, in the embedded system domain to implement a reliable security layer between critical information and functions and the surrounding untrusted domain.

HSMs rely extensively on specialized hardware to support a wide range of security features, including key handling and encryption, making them highly hardware-dependent compared to other approaches. This dedicated hardware is physically isolated from the surrounding environment and provides a very limited hardware interface to narrow the attack surface of the component [28].

Due to the absence of a defined standard and their dependency on the specific application, HSMs can be implemented using custom hardware tailored for specific purposes or by utilizing COTS boards that incorporate security features, such as a crypto-processor. In the latter case, it is up to the firmware to select which operations to support and to implement all additional desired characteristics, if required.

It is worth noting that, even though it is possible to enrich the capabilities of a Hardware Security Module by supporting them in firmware, the dependency on the underlying hardware tends to make these security components much less flexible than TPMs and TEEs. Moreover, the lack of universally established standards to define them further reduces the portability of a Hardware Security Module, which is usually adapted to the context in which it is expected to use it. However, there exist standards such as AUTOSAR's specifications for SHE [4] that address the problem by defining the main characteristics of a suitable HSM for vehicular ECUs.

Hardware Security Modules are primarily utilized for securely storing critical data, including secret keys, which should always remain within the secure zone and exclusively accessed by the HSM. These keys play a vital role in guaranteeing the confidentiality and integrity of protected data by serving as encryption/decryption keys or validating MACs, making it crucial to avoid any exposure of these keys.

Moreover, these modules have the capability to perform additional security operations, provided that the underlying hardware supports them. Examples of

such operations are Pseudo Number Generation or Asymmetric Cryptography [32], which can be used both to validate the confidentiality and integrity of protected data and for user authentication. Some HSMs may also support Secure Boot [4][32], which aims to verify that the current Operating System image has not been tampered with by computing the digest of the current image and comparing it with its expected value.

Chapter 3

Project Presentation and Requirement Analysis

From the State of the Art analysis reported in the previous chapter, it can be deduced that a dedicated module for security is a fundamental component of modern vehicle ECUs. Not only protecting secrets and running the cryptographic primitives with a certain level of performance is required; also controlling how and when to access them is a security-enabling feature that prevents several attacks involving stealing or deducing secrets. This, in turn, simplifies the security measures to be applied in the rest of the system, as it can rely on a trusted secure zone that is also robust.

The previous chapter has discussed several alternatives to implementing a secure module, especially focusing on secure storage capability. As can be seen, these solutions explore different strategies: some rely on special hardware hardened to resist cyberattacks, for example, by exposing very few interfaces with highly restricted communication or by preventing physical access to the device. Others aim to improve security by acting at the firmware level, defining Secure Zones that must remain invisible to all unsafe tasks. Some solutions require an underlying operating system, while others can work on bare metal.

The current chapter presents a novel approach to defining Secure Key Management Systems by illustrating the HSM Firmware Project, which aims to combine the inherent high security of Hardware Security Modules with the portability of Firmware; such property allows abstracting the underlying hardware as much as possible, with limited customization required for the target board.

Before designing and implementing the HSM Firmware, it is necessary to identify the context in which such a system shall be used, to determine which is the most suitable strategy to follow.

3.1 Non-functional Requirements

Since no system can be considered entirely secure, including automotive ECUs, the first step is to establish secure storage to safeguard sensitive or critical data. Any new Secure Key Management System must include the definition of a Secure Zone, regardless of the approach taken.

Among the solutions explored in ch. 2, Hardware Security Modules have been considered as the starting point of the project. They can act as a reliable measure to implement the Secure Zone of the ECU, as they implement specific security measures on the whole stack, from the software level to the hardware level. Defining specific cores, memory and peripherals to handle vehicle Security makes it a difficult component to violate. This does not imply that the system is invulnerable; security in Hardware is never sufficient on its own. It also needs suitable secure software that leverages all hardware security features properly.

Moreover, the design of the Secure Zone shall abide by established standards within the automotive industry specifically tailored for this type of module. By leveraging the expertise and best practices documented in these consolidated standards, it becomes feasible to develop high-level components with a solid foundation. In this way, the Secure Zone can be built upon a well-tested framework that aligns with industry requirements and expectations. Using standard protocols, algorithms and software strategies that are widely recommended in the cybersecurity field makes the system further reliable and robust, while reducing the development effort and potential security risks. For instance, AUTOSAR and NIST strongly recommend using widespread encryption algorithms such as AES-128 [4][33], which are typically supported by current Hardware Security Modules.

This project aims to follow the directives provided by such authoritative organizations, in an effort to develop Secure Firmware based on an established background. In particular, the current work is based on AUTOSAR's specifications for Secure Hardware Extensions, defined in [4], whose purpose is to define an on-chip extension for microcontrollers; the underlying intention is to embed cryptographic key management in hardware rather than in software to protect them [4].

According to AUTOSAR, a SHE is implemented by defining a specific hardware domain that is separate from the rest of the system; communication with the other cores and subsystems must be severely restricted by allowing only one core to communicate directly with the HSM, as depicted in Fig. 3.1. This is one of the few strong requirements regarding the hardware domain; so, to obtain a Secure Firmware satisfying AUTOSAR's expectations, it is not demanded to design custom hardware. Moreover, there already exist commercially available boards providing isolated hardware areas to support Root-of-Trust components, e.g. by defining cryptographic processors that can only handle security-related operations within the secure zone.

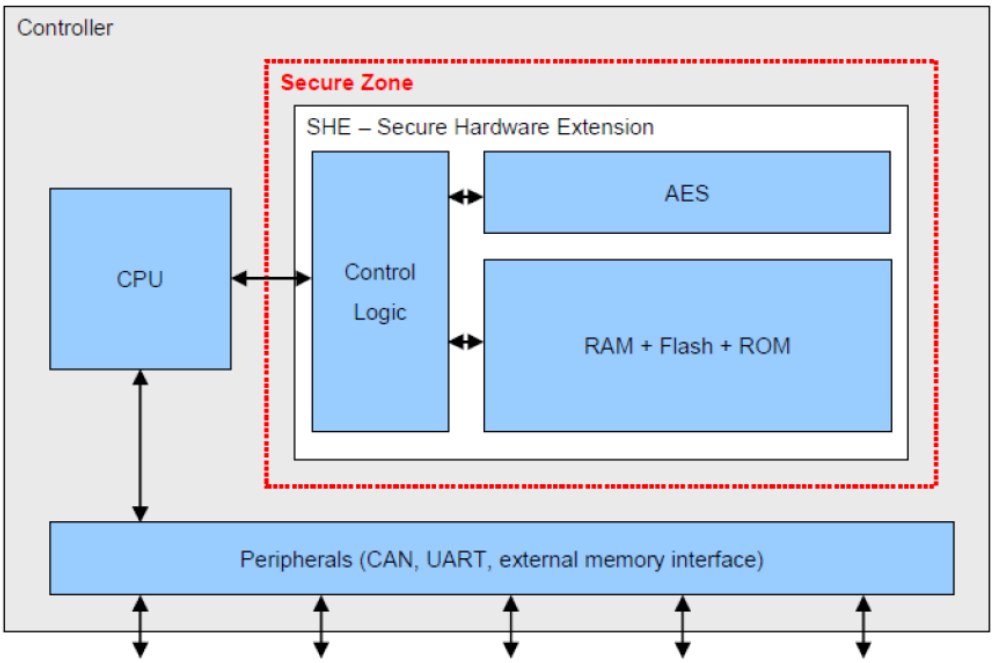

Figure 3.1: HSM physical isolation (Source: [4])

Therefore, this project should not exclusively target a specific board, as this would limit its applicability. Instead, the focus should be on supporting a certain category of boards and systems that are compatible with AUTOSAR's hardware specification for HSMs. This approach would ensure flexibility and adaptability to different architectures.

With the goal of flexibility and compatibility in mind, this project aims to design the Firmware for an HSM that can be deployed on an MCU board with minimal effort to adapt the Firmware to a specific platform. The only specific hardware required for this project is that the chosen core for the HSM must belong to the ARMv7-M family, as most HSMs are based on this family of microcontrollers. Additionally, if the target processor or SoC incorporates hardware accelerators for computationally intensive operations like encryption/decryption, utilizing these accelerators can enhance performance and leverage their capabilities. In summary, portability is an additional feature that should be supported. It is important to note that HSMs are not inherently designed to be portable, as their design heavily relies on the target hardware.

AUTOSAR defines the specifications for secure modules in the Secure Hardware Extensions document, which integrates the specifications for a Security Software Stack that should be supported by vehicles' ECUs. As illustrated in Figure 3.2,

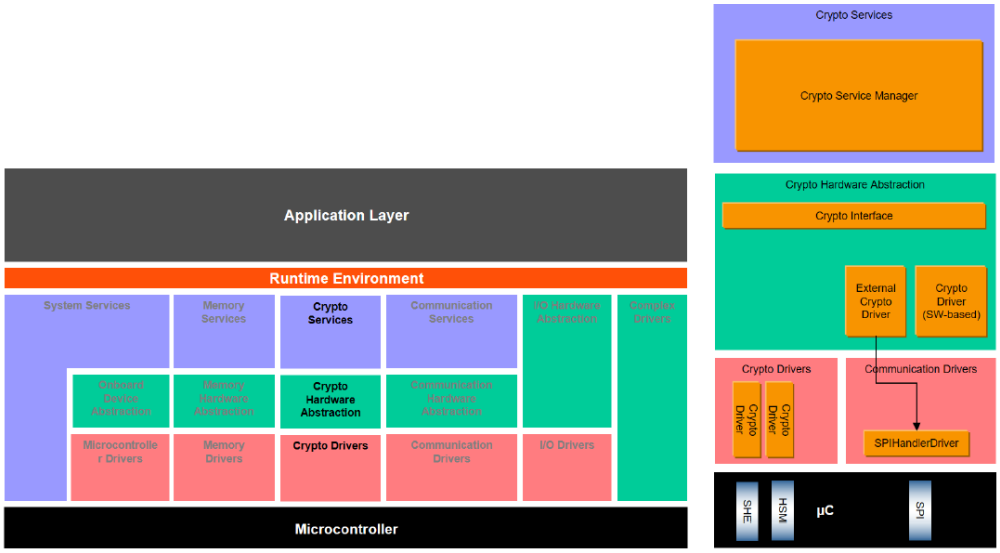

Figure 3.2: AUTOSAR Security Software Stack (Source: [2])

this stack encompasses all the software deployed on the ECU and establishes well-defined interfaces between each layer. In this architecture, application-level software, responsible for running user tasks, is not allowed to directly access security-critical peripherals. Instead, it must interact with a Crypto Service Manager [5]. The application-level software submits requests to the Crypto Service Manager, which analyzes and forwards them to the appropriate lower layers based on their validity. These requests may involve tasks such as message encryption/decryption and the loading and storing of private keys. Since application-level tasks are not equipped to handle these activities, they must rely on services provided by dedicated security modules.

In this context, AUTOSAR defines a spot for Secure Hardware Extensions in the Security Stack, where they work at the lowest hardware abstraction level. Above it, the Crypto Driver acts as an orchestrator that receives requests coming from the upper layers and dispatches them to Crypto Driver Objects, software modules that assist the Crypto Driver; their only purpose consists of submitting what the Crypto Driver is forwarding them to the underlying Security modules. According to this architecture, each Crypto Driver Object deals only with one specific security module, and the link between Crypto Driver Objects and Security modules is defined at design-time [2].

Consequently, it is known a priori who can interact with the HSM; considering that each Crypto Driver Object is a running task that manages its request queue, it can be assumed that the HSM interacts with one and only one task that is known from the beginning. It is noted that Crypto Driver Objects are not user tasks, but

system tasks that can be deemed trusted. This design choice allows us to manage access to the device strictly, filtering any request that does not come from the only certified task.

In short, a whitelisting policy can be applied. Although AUTOSAR does not mention using such a strategy, it would be possible to implement it without violating the specifications while increasing the security of the HSM.

The communication between the HSM and the Crypto Driver Object is facilitated through a dedicated driver. In line with the comprehensive specifications for Secure Hardware Extensions, AUTOSAR also provides an API document in [4] that outlines the recommended methods for utilizing these modules. By adhering to this standardized API, the project maintains a high level of consistency and compatibility with AUTOSAR's framework.

Given that the Security Stack is designed to incorporate SHE-compliant modules and serves as a fundamental reference for software security within AUTOSAR, integrating the current project into this framework would enhance its reliability and software portability. This integration ensures compatibility with software systems that adhere to the same architecture, further bolstering the project's overall robustness and expanding its potential reach.

3.2 Functional Requirements

Having clarified the intended context for the utilization of the HSM Firmware project, the next essential step is to articulate the specific functional capabilities that it shall encompass. By defining these functional requirements, it is possible to move to the design phase while ensuring that the HSM Firmware aligns with the intended objectives.

3.2.1 Key management

After establishing the role of the HSM in providing the Secure Zone of the ECU, the next step is to determine the specific data that needs to be stored and protected. Within the system, various tasks, including both system and user tasks, may have a requirement to share sensitive data. It is crucial that this data remains inaccessible to other tasks, ensuring confidentiality and integrity. To achieve this, encryption is commonly employed to conceal the content of critical messages, allowing only the intended sender and receiver tasks to comprehend it. However, encryption relies on the use of keys, which must be kept confidential; otherwise, the effectiveness of encryption is compromised. Thus, it is imperative for the HSM to securely store these secret keys, ensuring their confidentiality and safeguarding the overall security of the system.

In addition, the HSM must anticipate that the contents of the secret keys may change over time: in fact, it is a well-established practice to change encryption keys to reduce the effectiveness of attacks aimed to steal them. Thus, the peripheral should support protocols to load, remove and update keys. It is also fundamental to define how these keys shall be accessed and used. As AUTOSAR reports in [4], secrets must not leave the HSM unless in encrypted form: thus, it is necessary to provide the Security features of the HSM without exposing the keys.

3.2.2 Key usage and Encryption/Decryption

Once the contents of the Secure Zone have been defined, it is important to discuss the services that the HSM should provide with the keys, specifically how they will be used. As mentioned earlier, one of the primary purposes of secret keys is to encrypt or decrypt messages containing sensitive information that must remain confidential.

It is worth noting that all encryption algorithms developed thus far can be categorized into two main types: symmetric and asymmetric [34]. In symmetric encryption, only one key is shared between the communicating parties, whereas asymmetric encryption utilizes two separate keys per user. An advantage of asymmetric cryptography is that it eliminates the need for the sender and receiver to possess a shared secret or find a way to exchange it. This property of asymmetric cryptography is particularly useful in mitigating Man-in-the-Middle (MITM) attacks, which occur when an attacker intercepts and alters the communication between two parties attempting to share the secret key.

Given the HSM's capability to store a specific number of private keys and AUTOSAR's requirement for these keys to remaining within the HSM domain, it becomes possible for multiple tasks to make use of the same key stored in the HSM. This eliminates the need for tasks to exchange information in plain text, as the secret key is securely stored in the highly protected area of the ECU, safeguarding it from theft by malicious users, and assuming proper management by the HSM. It is important to note, however, that the Security Stack must ensure that the involved tasks are authorized to use these keys. SHE itself does not possess task-specific information, so the responsibility of verifying this condition lies with higher-level modules like the Crypto Service Manager.

Provided that the upper layers establish stringent access controls to the HSM and ensuring that only authorized tasks are granted usage rights, secure communication can be achieved through the utilization of symmetric encryption and the keys securely stored within the HSM. In fact, AUTOSAR mandates the use of SHE for symmetric cryptographic communication [4]. While asymmetric cryptography remains a viable approach for ensuring confidentiality and integrity, it is not a prerequisite in this scenario. Tasks are not equipped with certifications for

identification purposes, and digital signatures are not specified by AUTOSAR's SHE requirements. As per AUTOSAR guidelines, the Advanced Encryption Standard (AES) algorithm, which has been established by NIST as the de facto standard for symmetric encryption algorithms since 2001 [35], is specifically stipulated for SHE-compliant modules [4]. Specifically, the mandated requirement pertains to the variant of the block cypher that employs 128-bit blocks, known as AES-128.

The Advanced Encryption Standard offers multiple modes of operation to handle messages longer than a single block [36]. Among these modes, the Electronic Code Book (ECB) and the Cipher Block Chaining (CBC) modes are commonly used for message encryption, regardless of the input length. While ECB mode is suitable for processing individual blocks, it becomes less secure when applied to longer messages. This is because ECB mode treats each block independently, disregarding the overall structure of the message. In other terms, equal blocks in plain text would provide the same ciphered block. This violates Shannon's diffusion principle, which seeks to eliminate statistical correlations among the input bits [37]; even worse, message confidentiality could be broken without the necessity of stealing the encryption key, by leveraging attacks that target the output statistical correlation.

Hence, it becomes imperative to establish an alternative mode of operation to handle longer messages, and Cipher Block Chaining (CBC) mode is well-suited for this purpose. In CBC mode, when encrypting or decrypting a block, the algorithm takes into account the content of the input message that has been processed thus far. This is achieved by performing an XOR operation between the previous output block and the current input block before applying the encryption function [36]. By utilizing this approach, CBC mode ensures the interdependence and integrity of the encrypted blocks within the message.

While it is feasible to implement all other modes of operation, they are not deemed essential in the current context. For example, Counter Mode (CTR) is particularly suitable for scenarios where block encryption can be parallelized. It operates by relying on a nonce and a counter, with the output block solely dependent on the corresponding input block and the specific counter value chosen prior to the encryption process.

Instead, CFB mode is typically employed when AES needs to be utilized as a stream cipher, allowing the encryption of data bit by bit. On the other hand, OFB mode is primarily used to minimize error propagation. As these specific requirements are not present in the current project, the implementation of CFB and OFB modes is deemed unnecessary.

AES-128 offers more than just message encryption/decryption capabilities; it can be applied in various other scenarios. The algorithm's inherent strength and its resistance to decryption attempts make it suitable in other cases in which it is needed to use functions that provide confusion and diffusion within an input text, thus ensuring integrity. Even in these cases, the objective is to create a function

that is practically impossible to reverse.

One application where AES-128 proves useful is in the computation of Message Authentication Codes (MAC). MACs involve generating a concise message of fixed size, independent of the length of the input and that is appended to the output message. This message is computed using both the input text and a shared secret key known only to the sender and receiver. The MAC generation algorithm combines these inputs to produce a short message that is highly unpredictable without knowing the inputs [33].

Upon receiving a message from the sender, the receiver recalculates the MAC using the shared secret to ensure its correctness. Finally, the receiver then compares the calculated MAC with the expected value; if there is a perfect match, then the Authentication and Integrity of the output message are confirmed and the receiver can trust what has been received. Fig. 3.3 illustrates how MAC generation and validation are performed.

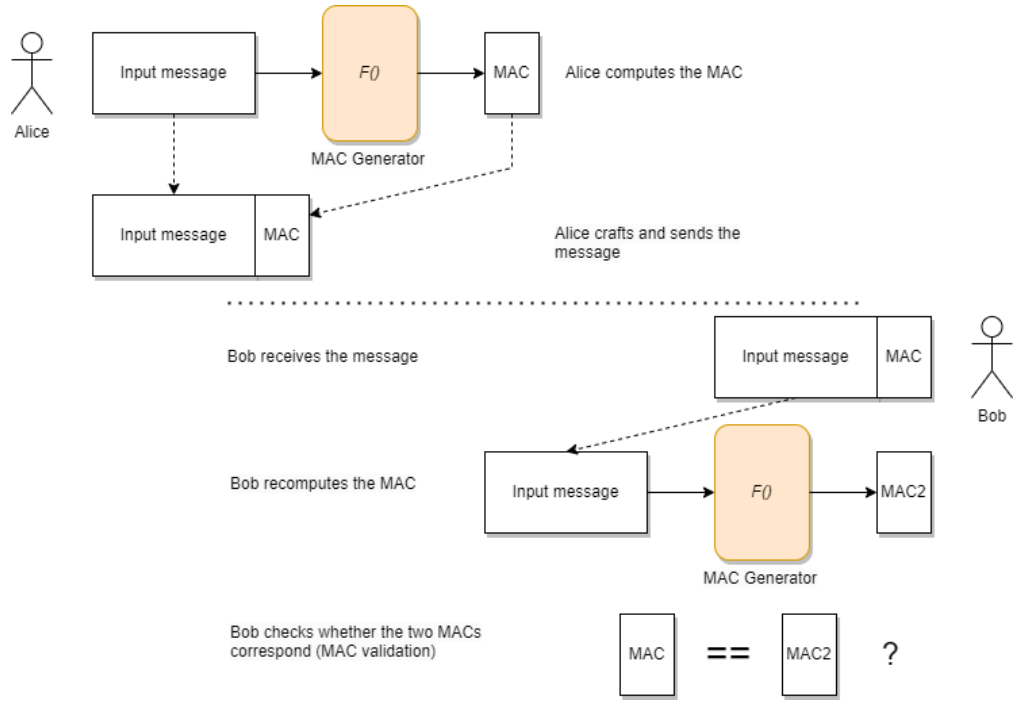

Figure 3.3: MAC generation and validation (Here, the text is not encrypted)

Authentication is ensured through the presence of a shared secret, while the integrity of the message relies on the characteristics of the MAC generation algorithm. The key requirement for any candidate algorithm used to generate MACs is that it must be practically impossible to reverse. One commonly employed approach is to utilize cryptographic hash functions, which produce HMACs when used for

MAC computation. Additionally, symmetric encryption algorithms, such as block cyphers in CBC mode, can be employed to generate MACs, leading to designations like CBC-MAC or CMAC. NIST recommends the use of such algorithms for this purpose [33]. Notably, it is important to recognize that Message Authentication Codes do not inherently provide confidentiality, as the message payload may still be in plain text.

For completeness, it is reported that there exist different strategies to compute the MAC, in case the output message is encrypted. If that is the case, there are three possible choices to compute the MAC, with different costs in terms of latency. All these cases use two different keys for encryption and MAC generation, as it is better to differentiate keys rather than using the same for any security operation:

Authenticate and Encrypt (A&E) In this method, the output message is

$$E(K_1, P) \,||\, MAC(K_2, P)$$

where $E(K, M)$ is the encryption function taking a key K and message M, $MAC(K, M)$ is the MAC generation algorithm that generates a MAC starting from a secret K and a message M. Finally, $||$ represents the concatenation operator. In order to validate the MAC, it is required that the receiver computes the original input message to recalculate the MAC.

Authenticate then Encrypt (AtE) Here, the output message is

$$E(K_1, P \,||\, MAC(K_2, P))$$

Even in this case, it is necessary to perform both decryption and MAC generation before validating the message.

Encrypt then Authenticate (EtA) The final case, where the output is

$$E(K_1, P) \,||\, MAC(K_2, E(K_1, P))$$

In this scenario, the receiver initially calculates the MAC from the encrypted output. The decryption of the incoming message to reveal its concealed content is only carried out if the computed MAC matches the expected value. By conducting MAC validation before decryption, the process is expedited since decryption is a computationally intensive operation.

Regarding the HSM, AES-128 has been defined only to ensure confidentiality by securely encrypting messages thus far. However, a strategy to allow the module to detect incoming message corruption must be implemented, especially for the most security-critical operations such as updating or removing keys; otherwise, it would be possible for an attacker to implement a Denial of Service (DoS) attack

by leveraging a MITM attack that intercepts the value of the new key to update and modifies it before submitting such value to the HSM; as a result, the task that asked to update the key will believe that it knows the contents of the stored key and will be mistaken. Whenever the task must know what has been stored in the HSM in order to obtain a certain service, it will result in failure and the task will not see its demands fulfilled.

For these reasons, AUTOSAR demands to use AES-128 as a MAC generation algorithm [4], by following the specifications provided by NIST in [33]. The details about how AES-128 shall be used for such a purpose are not required during the requirement analysis phase; they will be discussed in the next pages, where the inner workings of the HSM will be described. In particular, the Encrypt-the-Authenticate strategy shall be used to provide fast MAC validation and reduce latency in case of invalid input messages.

3.2.3 Memory Protection

An essential functionality that the HSM should support is Memory Protection. Memory Protection Units (MPUs) are hardware modules that work in conjunction with Memory Management Units (MMUs) to prevent memory management bugs and counteract memory corruption exploits. MPUs are widely employed in modern systems [38], including processors such as the Cortex ARM-v7 architecture [39], which incorporate these hardware modules directly into their System-on-Chip (SoC) designs.

MPUs achieve their objective by defining memory regions and specifying which tasks are permitted to access them, as depicted in Fig. 3.4. This approach ensures that each task or user has access only to a limited portion of memory, preserving memory integrity. Additionally, MPUs play a crucial role in thwarting attacks like stack overflows, which deliberately exceed the task's allocated memory. MPUs can detect such events and respond accordingly by invoking their Interrupt Service Routines, whose behaviour is implementation-dependent.

Moreover, the use of MPUs during the development and testing phase can assist in identifying bugs at an early stage [38]. By monitoring nominal memory accesses and verifying their expected behaviour, MPUs can help detect and mitigate issues before they escalate, thus minimizing false positives and validating memory usage.

Given the established effectiveness of MPUs in enhancing memory management security and their diffusion, the current project aims to leverage this technology to enhance the overall quality of the work. Specifically, the project focuses on verifying that the tasks executing within the HSM do not exceed their allocated memory, ensuring robust memory protection within the system.

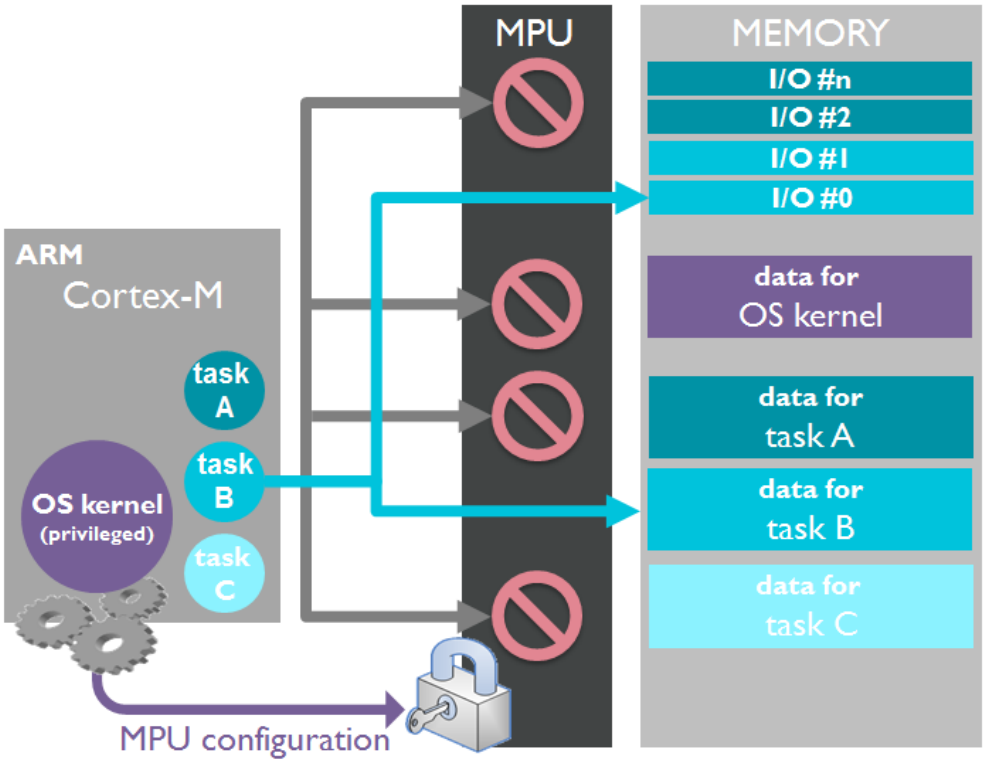

Figure 3.4: Memory Protection Unit in ARM Cortex-M cores (Source: *ARM*)

3.2.4 Asynchronous Communication

Lastly, it is crucial to maintain a high level of responsiveness in the system. Vehicles not only serve as safety-critical systems but also operate as real-time systems with strict deadlines that must be met. Even minor deviations from these deadlines can potentially lead to harm or injury to the driver and passengers onboard.

Hence, it becomes necessary to monitor the latency of all operations within the ECU to ensure they remain within an acceptable range. Certain operations, such as encryption/decryption in CBC mode, can be time-consuming and occupy valuable scheduling time that should be reserved for critical activities. Therefore, it is crucial for the HSM to operate without compromising the responsiveness of the system. AUTOSAR recognizes this challenge and defines specific latency limits for encryption/decryption with AES-128 in both ECB and CBC mode [4].

Additionally, the system is required to function asynchronously [4], meaning that when the Crypto Driver Object sends a request to the HSM, it should not wait for the request to be completed but should be able to return immediately. It is worth noting that the HSM operates on a dedicated core, allowing it to run in parallel with the rest of the system. Finally, the system must also be capable of

detecting when the HSM has completed a request so that the Crypto Driver Object is aware that it can submit a new one to the HSM.

Chapter 4

HSM Firmware Design

4.1 High-level architecture

After establishing both functional and non-functional requirements, the following step of the project involves the exploration of potential solutions that can fulfil these specified requirements. This section is a crucial step in the project's development journey as it aims to evaluate potential solutions that will facilitate a robust and effective implementation. A thorough examination of the project's structural and behavioural aspects during this exploration lays the foundation for informed decision-making and enhances the overall quality of the final design First of all, it is recalled that one of the project's distinguishing characteristics pertains to the distinctive nature of the underlying hardware. Hardware Security Modules (HSMs), including those compliant with the Secure Hardware Extension (SHE) standard, impose specific hardware prerequisites that demand careful consideration. It is worth noting that SHE-compliant modules require deployment on isolated CPU cores with limited interfaces to other board components of the board [4]. Consequently, it is presumed that the entire project will be implemented on a board accommodating a core designated for Root-of-Trust elements.

Nevertheless, it is essential for the module to be accessible from external sources. From the perspective of the end user, represented by a Crypto Driver Object [2], submitting expected requests with minimal complexity and basic knowledge of the HSM becomes imperative. To facilitate this interaction, the presence of a driver is essential, as it provides a software interface.

The driver's responsibilities extend beyond merely receiving incoming requests from the Crypto Driver Object. It also assumes the role of analyzing and verifying the validity of these requests. Furthermore, it handles all internal communication with the HSM, including forwarding input requests to the hardware module itself, retrieving results, and executing any necessary post-processing steps mandated by

the user. The Driver is also the *only* software module that is allowed to access the HSM.

Importantly, the driver cannot be deployed on the target HSM; instead, it must be available for the core where the Crypto Driver runs; as a consequence, the project can be split into two domains, as depicted in 4.1.

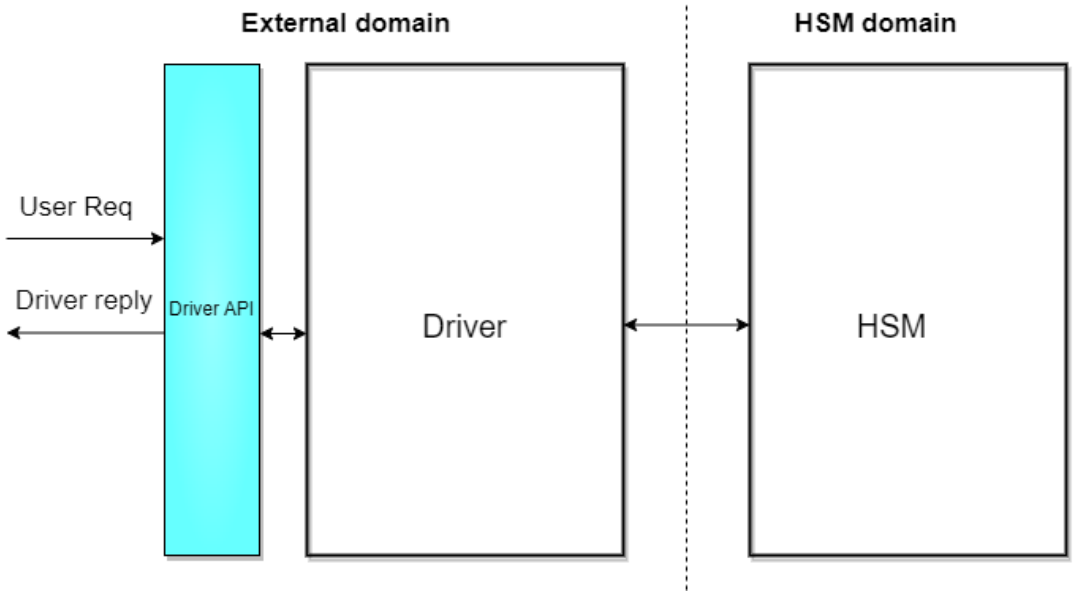

Figure 4.1: Conceptual representation of the Driver and the HSM on separate domains

To ensure adherence to AUTOSAR standards, developers experienced with AUTOSAR can simplify integration by utilizing the AUTOSAR-prescribed API for Secure Hardware Extensions (SHE)[4] simplifies integration for developers that are experienced with AUTOSAR standards. In fact, it is expected that the HSM will be seamlessly integrated into the existing AUTOSAR Security software stack. As a result, it is likely that the developer responsible for creating the Crypto Driver will adhere to the interface specifications defined by AUTOSAR, ensuring smooth access to the HSM In particular, the API covers all operations of interest for the HSM, such as:

- Encryption and decryption, both in ECB and CBC mode;
- MAC generation and validation
- Retrieving the current status of the HSM

HSM Firmware supports all the functions related to these operations and strives to adhere to the API specifications

The API also includes functions for:

- Cancelling ongoing operations
- Secure Boot
- Random Number Generation

which are out of the scope of the project, thus they are not used. Although these functions are currently not utilized, they lay the foundation for future integrations, enhancing the capability of the HSM and providing an AUTOSAR-compliant interface. Regarding the driver, its main purpose is to allow the end user to submit requests to the HSM, which means that the driver itself should not perform any security-critical operation on its own. Moreover, this would violate the founding principle of the HSM, designed to perform such operations in its secure environment. For this reason, the driver acts like a software filter that rejects any ill-formatted request. However, it also filters out any request that does not come from the Crypto Driver Object by means of a whitelisting policy that is aware of the memory area where that task is located.

After validating a request, the driver forwards it to the HSM for processing. However, the driver must know the hardware interface to communicate with the peripheral and such interface can change from one board to another.

To simplify the communication protocol between the driver and the HSM, both utilize a software-defined Hardware Abstraction Layer (HAL), an interface that ignores the hardware details of the physical interface between the HSM domain and the outside.

It is crucial to consider the potential vulnerability of the module to Man-in-the-Middle (MITM) attacks when the external domain and the HSM communicate via a shared channel, such as a CAN bus. In such cases, it would be possible for an external module to intercept and listen to critical communication occurring on the bus. To mitigate this risk, it is recommended that both the driver and the HSM encrypt all their messages exchanged during communication [4].

Alternatively, the highest level of security in the channel can be achieved if it is a point-to-point connection, allowing only the driver and the HSM to directly communicate with each other. Some boards incorporate cores specifically designed for security-critical operations and may define such a type of channel. Therefore, for the purposes of this project, it is assumed that the communication between the driver and the HSM is point-to-point, eliminating the need for message encryption.

The Hardware Abstraction Layer must differ between the two sides of the channel, since the driver must access the HSM registers and memory in write-only

or read-only mode, whereas the HSM must use them in the opposite way. Hence two separate HALs, one for the driver and one for the HSM are needed. It is worth noting that, although the HAL abstracts the underlying hardware, board-specific code must be implemented to support the communication between the driver and the HSM. It's important to note that while the HAL abstracts the underlying hardware, board-specific code is necessary to facilitate communication between the driver and the HSM. As reported later, the HAL is one of the few software interfaces that need customization to enable porting the entire project from one board to another. Another abstraction interface is the Operating System Abstraction Layer (OSAL), which is defined only for the driver to handle the asynchronous communication with the HSM. Both of them will be explored in detail in sec. 4.2 and sec. 4.3.

Thanks to these abstraction layers, requests can eventually reach the HSM, where they shall be treated and processed. Essentially, the HSM behaves as a Finite State Machine that stays in an idle state until a request is detected; when it happens, the HSM analyzes what has been received and determines request metadata such as its type, its arguments or where to store the final output. Once the HSM has understood what is needed to satisfy the request, it dispatches the execution of the command to its functional units, which are a Key Manager to store keys and a Cryptographic Unit to handle any operation reported in sec. 3.2 and that involves AES-128. After executing the command, the HSM sends the results to the external domain through its Hardware Abstraction Layer. The driver can then store the output in the memory of the external domain and notify the Crypto Driver Object of the completion. In this way, the Crypto Driver Object can submit new requests as it can assume that the HSM is not busy anymore.

Treating the HSM as an FSM is due to the fact that, except for requests coming from the driver, there are no other operations that the HSM has to perform autonomously, without external input. Therefore, one can consider the HSM as a passive but responsive listener, ready to handle any request.

To visually represent the concepts discussed, Figure X.X illustrates the high-level architecture of the project.

4.2 The Driver

Once the high-level architecture of the project has been established, it is possible to delve into the details of the features that HSM Firmware provides; with this further knowledge, it will be possible to understand with a higher level of detail how requests are actually processed inside the HSM, eventually exploring the entire request lifecycle.

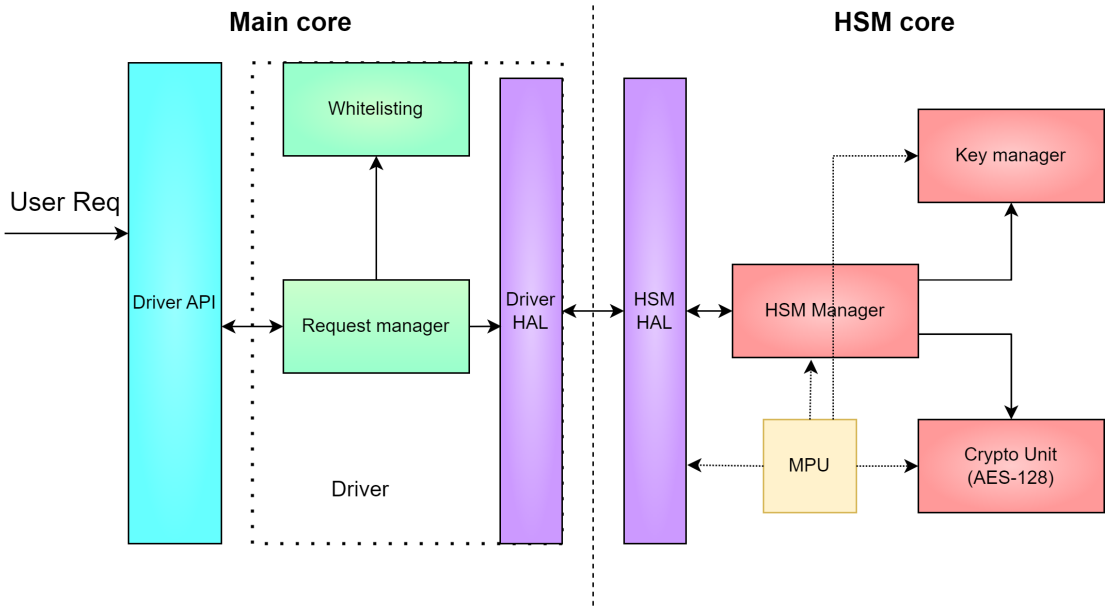

Figure 4.2: High-level architecture of the HSM Firmware project

4.2.1 The API

To describe the design choices of the project, we can start with the driver interface as the starting point, using a top-down approach. Additionally, since requests are generated externally by a system task, which corresponds to a Crypto Driver Object [2], the interface serves as the initial functional block that they need to navigate. As reported in sec. 3.1, AUTOSAR has already defined an API for Secure Hardware Extensions [4] to implement most of the features that are deemed necessary for this project. However, not all the API must be implemented since it also comprises functions to support additional features, such as Secure Boot and Random Number Generation, which are out of the project's scope. However, it is not necessary to implement the entire API as it includes functions to support additional features like Secure Boot and Random Number Generation, which are beyond the scope of this project.The API-defined functions currently supported by the HSM can be divided into three categories:

- Encryption/Decryption and MAC Generation/Validation, which require the Cryptographic Unit of the HSM to be processed:
  - `HSM_CMD_ENC_ECB()`: to encrypt a message in ECB mode. This mode is deemed unsafe as it allows to perform attacks based on the correlation between the output blocks; for this reason, only messages whose size is equal to one block can be encrypted in ECB mode. In fact, encrypting a

message in CBC mode is equivalent to encrypting it in ECB mode if the length of the message corresponds to the block size.

- `HSM_CMD_ENC_CBC()`: to encrypt messages in CBC mode, whose size must be a multiple of the block length, i.e. 128 bits. It is noted that if $k \times B_{len} < M_{len} < (k+1) \times B_{len}$, with $k \in \mathbb{N}$, $B_{len} = 128$ bits and M_{len} is the message length, then the message cannot be encrypted as is; therefore, they must be padded so that the final length is a multiple of the block length. According to AUTOSAR's specifications, it is expected that the HSM shall be able to detect the end of the message by finding the padding sequence at the end of the message. However, this feature has not been implemented; currently, it is required that the caller provides the input message size as an additional argument.
- `HSM_CMD_DEC_ECB()`: this function reverses the effect of `HSM_CMD_ENC_ECB()`, i.e. it implements AES-128 decryption in ECB mode; even this function can be used only on single blocks.
- `HSM_CMD_DEC_CBC()`: this function implements AES decryption in CBC mode. As well as for `HSM_CMD_ENC_CBC()`, messages must be padded correctly before being processed.
- `HSM_CMD_GENERATE_MAC()`: MAC generation (CMAC) using AES-128 as the block cipher. Here, messages can have any input size, it is up to the HSM to pad the message accordingly. This difference is due to the fact that padding messages for encryption in CBC mode can follow several padding strategies, such as PKCS#7 [40] or ISO/IEC 7816-4 [41]. This is not true for the MAC generation algorithm used in the project, i.e. CMAC-AES [33]. As a matter of fact, the original AUTOSAR's API specifications include a `message_length` argument to report the length of the message, which can be a non-multiple of the block length.
- `HSM_CMD_VERIFY_MAC()`: MAC verification (CMAC) algorithm using AES-128 as the block cipher as well. Given an input message, its length and its pre-computed MAC, the algorithm recomputes the CMAC and compares it with the expected one, eventually reporting to the caller whether there is a match between the two MACs.

- Key loading and update, which require the HSM Key Manager:
  - `HSM_CMD_LOAD_PLAIN_KEY()`: load a key in plain text, without any specific algorithm to forward the key to the HSM in a secure way. It is a function to be used rarely, in all the cases where it is needed to use a key that is not stored in the HSM. For instance, it is useful for Key Derivation, i.e. the process with which a secondary key can be computed starting

from a primary key. This technique can be used both for encryption and MAC generation and is especially useful to load keys inside the HSM, as described in sec. 4.3

  - `HSM_CMD_LOAD_KEY()`: load a key securely, by embedding it into an encrypted message whose key is the previous value of the key. In fact, HSM-stored keys can be identified by means of a numerical ID and any request to load/update/remove keys uses such ID to define which key is the request related to. All AUTOSAR-defined functions that imply to use of a certain key, e.g. MAC generation, use an ID to identify which key to use, without providing it in plain text over the channel. About key update, if we want to load key k_i, it is required to know its previous value. This approach guarantees authentication, as it can be supposed that whoever asks for a key update is also the owner of such key since they know their content. By encrypting the request, confidentiality can be provided. Finally, integrity is guaranteed thanks to the use of a CMAC appended to the encrypted request to detect whether it has been corrupted along the way.

- Status and control commands to configure the driver

  - `HSM_CMD_GET_STATUS()`: Retrieve the HSM Status Register [4], which informs whether the HSM is currently busy. Other information provided by this register, such as the results of the Secure Boot verification, is not used in this project.

For all the functions reported here, the arguments are not described in detail; however, further information about arguments, their types and the return codes is available in the source code documentation.

Although the HSM Firmware aims to be easy to integrate within the AUTOSAR Framework and therefore aims to comply with the SHE API specification as much as possible, the original specifications did not take into account certain aspects of driver management that were, in this case, considered necessary. For this reason, it was deemed appropriate to add functions to allow for configuring and managing requests, as well as for HSM and Driver initialization, status check and request post-processing.

The additional functions included in the current project are reported in the following:

- `HSM_DRIV_INIT()`: initialization function for the driver, which resets all global data, including callbacks for asynchronous communication. Since the HSM runs on a separate core, it launches a boot-time routine autonomously and this function waits for the HSM to be ready before returning.

- `HSM_IS_BUSY()`: checks whether the HSM is not busy anymore. It can be used for polling the HSM while waiting for the end of the currently processed request or for implementing synchronous communication, in case the Crypto Driver Object has no other task to perform and can poll the HSM without losing performance.

- `HSM_CALLBACK_SET()`: as described later during the description of the Request Manager, callbacks have a fundamental role in providing the asynchronous communication feature required by AUTOSAR. In particular, they allow defining custom code for post-processing of the results. Since the only task that can define such callbacks is the Crypto Driver Object of the HSM, it is possible to define additional code at run-time for handling security-critical results. Callbacks are defined per kind of operation, i.e. when set, the callbacks are executed every time a request of a certain kind is requested, e.g. after retrieving the results of MAC validation.

- `HSM_CALLBACK_UNSET()`: it reverts `HSM_CALLBACK_SET()`, i.e. it resets the callback associated with a certain operation. If a request is launched without setting a callback, no post-processing action will be taken, but the request will be executed nonetheless.

- `HSM_CMD_REMOVE_KEY()`: while AUTOSAR expects the capability of loading keys into the HSM, it does not define any specification to remove keys. This function aims to fill this gap by providing this additional feature.

- `HSM_CMD_DERIVE_KEY_ENC()`: key derivation function to create a derived key starting from an input key provided as an argument. It uses a compression function based on AES (AES-MP) that acts as a cryptographical hash function to generate a pseudo-random key. The choice of the algorithm is specifically dictated by AUTOSAR in [4]. Once generated, the newly-defined keys can be used as input keys for encryption and decryption.

- `HSM_CMD_DERIVE_KEY_MAC()`: by using the same algorithm of the key derivation function for encryption, it is possible to generate a derived key for MAC generation or validation. The main difference in comparison with the previous function relies on how the compression function is used. In particular, such a cryptographic hash function uses a constant value to be appended to the input key before compressing it. So, in order to generate two different keys starting from the same input, a possible approach consists of using a different constant value to append. Using separate keys for MAC generation and encryption of the same message is strongly recommended as it further reduces the correlation between the output data, such as the generated MAC and the encrypted message.

- `HSM_ISR_HANDLE_NOTIFY()`: an Interrupt Service Routine (ISR) that handles the completion of a request. This ISR is readily available in the main project header and can be directly included in the Interrupt Vector Table without any modifications. In addition to the presence of a security-critical core and the establishment of point-to-point communication between the external domain and the HSM, there is an additional hardware requirement imposed by the project. Specifically, the HSM must provide an interrupt line from the HSM to the external domain to signal the completion of a request. This requirement arises from the need to meet the asynchronous communication requirements, as an alternative strategy, such as polling, is incompatible with these requirements.

Moreover, all API functions that use key indices to access HSM stored keys have also a counterpart where the key shall be provided by the user. For instance, `HSM_CMD_ENC_ECB` has an equivalent function `HSM_CMD_ENC_ECB_WITHKEY` where the input key is provided as a byte array. However, these functions were originally developed to enable encryption and MAC generation in case the input key was not stored in the HSM. However, by using `HSM_CMD_LOAD_PLAIN_KEY`, it is possible to load a key once and then use it without exposing the key over the channel. Thus, these functions are currently deprecated.

The entire API, both custom and AUTOSAR-compliant, can be viewed in Fig. 4.3. Deprecated functions, together with the ISR, are not reported in the figure; in particular, the latter is not a callable function for the Crypto Driver Object since it is made accessible only to use in the Interrupt Vector Table at boot-time.

4.2.2 Request management

After familiarizing third-party developers with the interface for communicating with the HSM, they can proceed to submit requests to the device. At this stage, it becomes essential to delve into the process of analyzing and filtering these requests before they access the HSM.

The core of the driver of the HSM Firmware is the Request Manager. This software component represents all the internal logic of the driver and has various purposes:

- Checking the validity of the requests, e.g. verify whether the input arguments are in a valid range.
- Checking task authentication and authorization; since the Crypto Driver Object is the only task allowed to access the HSM, it is mandatory to identify who asked for the HSM.

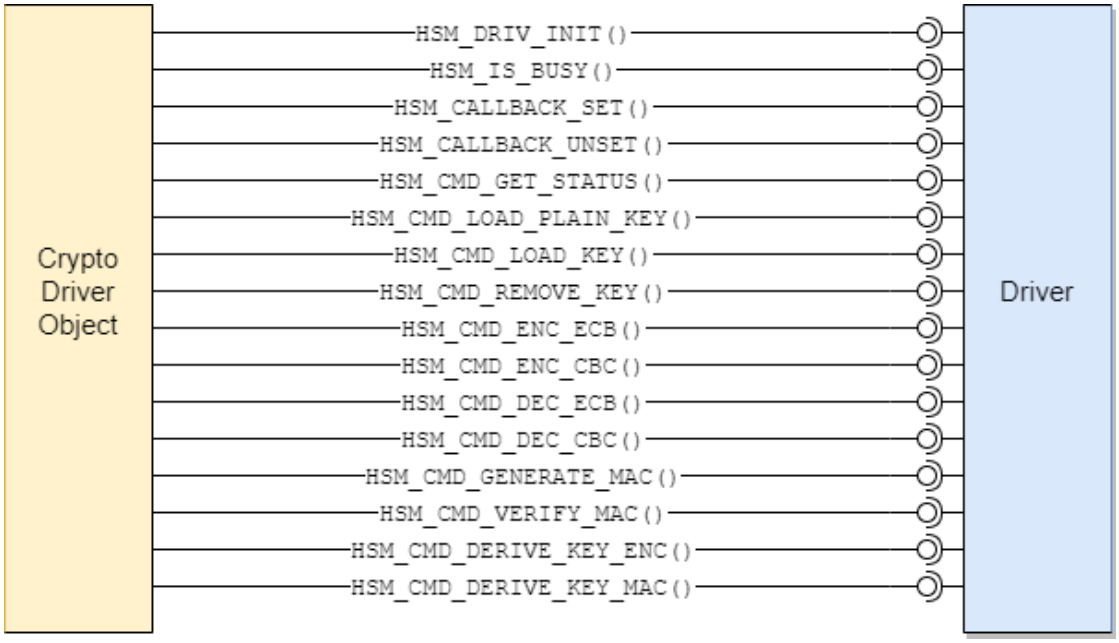

Figure 4.3: HSM Driver API

- Forwarding valid requests to the HSM.
- Processing request results and making them accessible to the task that initiated the request.

To determine whether a request is valid, the Request Manager needs to conduct a series of checks whenever a new request is received. In the following, the sequence of these checks is outlined:

1. Verify whether the caller task is the Crypto Driver Object by means of a whitelisting policy;
2. Check that the driver has been initialized, i.e. `HSM_DRIV_INIT` has been called previously;
3. Check whether the HSM has been correctly initialized as well;
4. Check that the core is currently in Privileged mode; this step further enhances the security of the firmware, by guaranteeing that not only the caller task corresponds to the one allowed, but also that no task with user privileges could access the HSM;
5. Verify if the HSM is currently occupied, meaning it is engaged in processing another request and has not yet completed its task.

6. Given that each request is accompanied by specific arguments, distinct checks are conducted on these arguments according to the command specifications;

Although the HSM cannot be utilized, unauthorized tasks may still try to obtain information about the driver's current state, including the status of ongoing requests and the initialization status of the HSM. The latter piece of information could be relevant in scenarios where the HSM handles Secure Boot verification. For example, if an attacker manipulates the Operating System image and the HSM indicates that the current RTOS image remains uncorrupted, the attacker may consider their strategy successful. However, such an occurrence is highly improbable since Secure Boot verification would detect any tampering with the RTOS image. Regardless, it is imperative to prevent unauthorized tasks from accessing any information pertaining to the HSM and the driver. By giving priority to task authorization checking, unauthorized access attempts would be prevented from obtaining any status information, except for receiving an indication that the driver rejected the request due to the initial check failure.

The initial verification is performed through a whitelist policy to determine if the calling task corresponds to the Crypto Driver Object.To implement such a policy, there are a few assumptions to make about this task and how it is allocated in memory. In particular, this check assumes that the task's memory allocation is contiguous and resides within a specific memory range without any interleaved memory. Fig. 4.4 displays the typical memory layout of a task, created by an Operating System once the latter starts.

It is noted that this task is defined at the system level, meaning that its existence is not dependent on user actions and should last for virtually the entire time from the moment the engine is turned on to when the vehicle is turned off. Moreover, it is recalled that this is the only task that is able to communicate with the HSM, since the Crypto Driver delegates the Crypto Driver Object to use this module [2] and does not use it directly. Thus, it can be assumed that such a task is always active.

Finally, given that it is a system-level task, embedded software developers that want to integrate this project inside an AUTOSAR-compliant firmware can define the Crypto Driver Object task within a specific address range, defined statically.

Starting from these assumptions, the whitelisting policy is based on a simple check using the return address of the caller function, whose behaviour is reported in Alg. 1 and depicted in Fig. 4.5.

In order to guarantee that the return address belongs to the caller task area, it becomes imperative to enforce a restriction on the compiler to refrain from performing any form of function inlining on all driver functions. Given that the Hardware Security Module (HSM) is exclusively employed for security-critical operations, where the primary latency arises from executing computationally-heavy

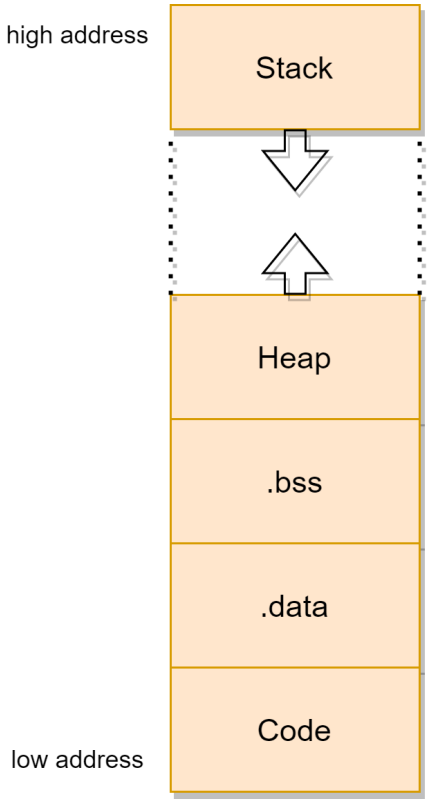

Figure 4.4: Task Memory layout

Algorithm 1 Whitelisting check

```
1:  procedure WHITELISTING_CHECK
2:      ▷ waddr_l ← lowest address of the whitelisted region
3:      ▷ waddr_h ← highest address of the whitelisted region
4:      ret_addr ← return address of the caller function
5:      if waddr_l ≤ ret_addr ≤ waddr_h then
6:          result ← True                          ▷ Authorized access
7:      else
8:          result ← False                         ▷ Unauthorized access
9:      end if
10:     return result
11: end procedure
```

requests rather than managing them, it can be reasonably assumed that this constraint does not impose a significant performance penalty.

Implementing a stringent access policy that prohibits any function external to the predefined whitelisted region from accessing the Hardware Security Module

significantly diminishes the driver's attack surface. This approach ensures that solely a trusted task possessing proper authorization can utilize the HSM, thereby fortifying the overall security level of the system.

However, it is worth noting that the design cannot guarantee security on its own: if a careless developer or an insider develops insecure system software, e.g. the Crypto Driver Object is easily tampered with, then the security of the HSM also suffers. Not surprisingly, AUTOSAR itself states that, in the absence of adequate security measures in the rest of the system, it cannot be guaranteed that the HSM is inviolable, even if such a module conformed to SHE specifications [4].

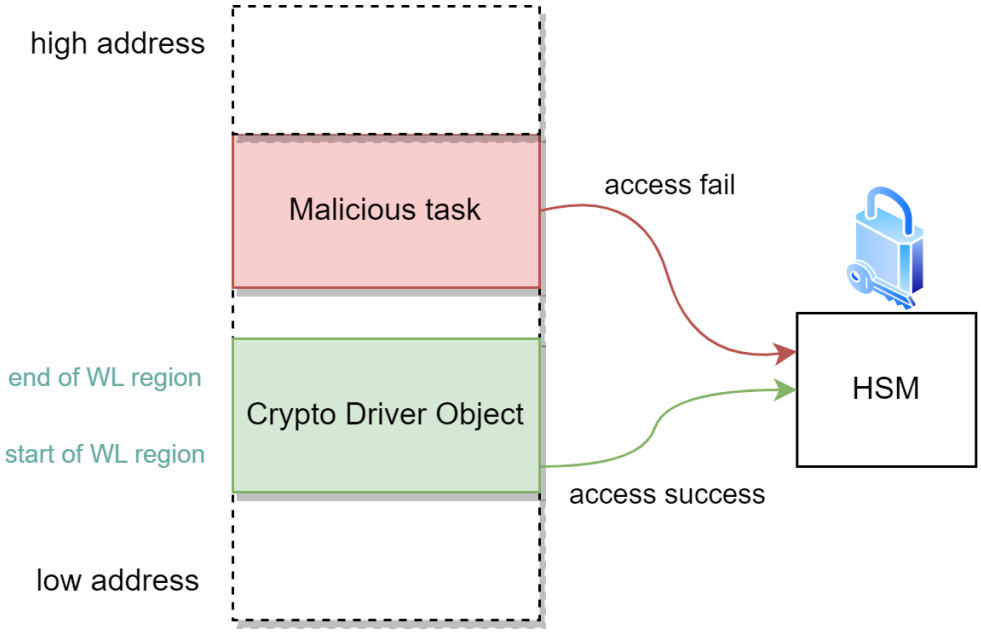

Figure 4.5: The whitelisting strategy

Assuming that a request successfully satisfies the initial verification conducted by the Request Manager, the subsequent phase involves formulating the request in a manner comprehensible to the Hardware Security Module (HSM). This entails encapsulating all input data provided by the Crypto Driver Object within a Request Packet, which encompasses pertinent metadata such as the Request Type and, when unknown beforehand, the size of the input arguments. Notably, certain inputs, such as messages intended for encryption in CBC mode or input messages for MAC generation, may exhibit variable lengths. In the present design, the HSM needs to possess knowledge regarding the size of the data it is about to process.

Once the packet is ready, the Request Manager can send it to the HSM. At this stage, the Hardware Abstraction Layer (HAL) assumes a crucial role in facilitating communication between the driver and the HSM. The HAL acts as an intermediary,

offering an abstraction layer that aims to minimize dependencies on the underlying hardware. Consequently, the driver can interact with the HSM through a logical interface that remains agnostic to the particular implementation details. Specifically, the interaction between the driver and the HSM is facilitated through a logical interface comprising the following components:

- A Status Register (9-bit long), which includes the AUTOSAR-defined Status Register [4], but also contains additional information used by the driver and not specificated by AUTOSAR, i.e. whether the HSM is successfully initialized or not. From the driver's point of view, this register is read-only.
- A Control Register (1-bit long) whose only purpose is to inform the HSM that a new request has been successfully received from the driver; only the driver can actively control this register.
- Two input buffers that are used in parallel when submitting a new request to the HSM. In particular, one fixed-size buffer contains all the request metadata that the HSM should read before analyzing the payload of the request itself, which is received through the other buffer, whose size is configurable, with a minimum acceptable size of 64 bytes
- Two output buffers with the same purpose and size as those mentioned above; using two buffers instead of transmitting only the raw output of the request allows informing the driver about the outcome, possibly communicating error codes in case of request failure.

Since every type of request expects a specific number of arguments with different lengths, input data is serialized according to the request type, as shown in Fig. 4.6. By conveying the request metadata through the input buffers, the HSM can deserialize the data and accurately interpret the payload before processing it.

Regarding the interface with the HSM, the responsibility of mapping the registers and buffers to the memory of the Hardware Security Module lies with the implementation itself, as the driver is agnostic with respect to their specific placement and access mechanisms. This task is instead delegated to the Hardware Abstraction Layer, which handles the management and configuration of these elements.

The Hardware Abstraction Layer (HAL) is one of the few software modules in the HSM Firmware project that requires customization before deployment on a specific target board. This is because the HAL heavily relies on the underlying hardware. As discussed in Chapter [chap. 3], the project aims for a high level of portability while meeting security-related constraints. To achieve this goal, it is important to isolate software components from hardware and the operating system, creating a clear separation between logical elements and system-specific code. As a result, the HAL is the only software module that requires developer intervention

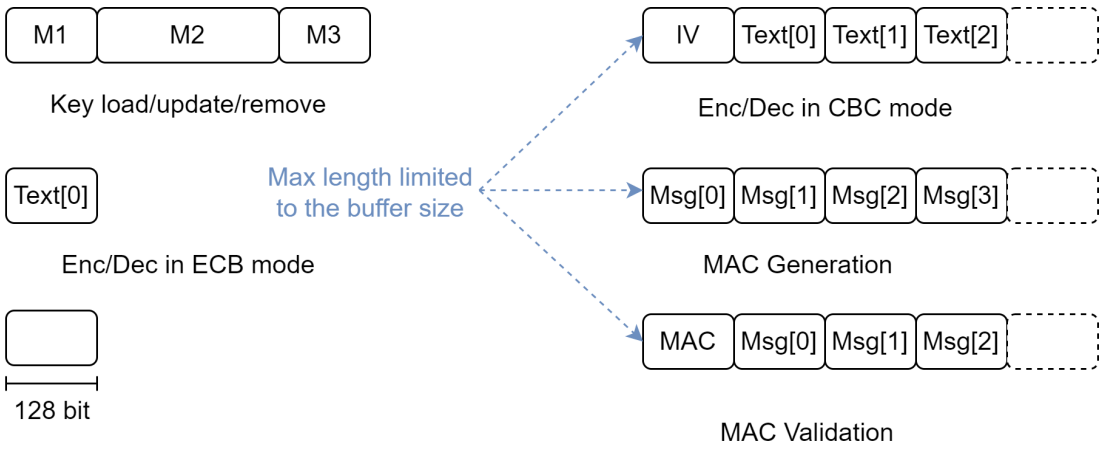

Figure 4.6: Data layouts in the buffer memory of the HSM

to ensure firmware compatibility with the target board, along with the Operating System Abstraction Layer (OSAL)

The HAL assists in forwarding requests to the HSM, which accesses them from its buffers along with the accompanying metadata. Upon detecting a new request, the HSM processes it according to its specific type. Once the results are generated, it is crucial to provide the output to the caller task for further use. Specifically, in the case of the Crypto Driver Object, the results are forwarded to the upper layers of the Security Stack, including the Crypto Driver, the Crypto Service Manager, and finally, the user task that initially requested the data [2]. To comply with AUTOSAR requirements for the asynchronous communication protocol between the HSM and the user [reference], requests need to be processed by the HSM in a non-blocking manner. As a result, when a task initiates a request, the driver returns control to the calling task under two circumstances:

- The request is invalid or unauthorized, thus the operation is aborted and the driver will not forward it to the HSM;
- The request is accepted and sent to the HSM.

In the latter scenario, the driver does not wait for the operation to complete. Instead, it relies on the HSM to notify the driver when the final output is ready, allowing the driver to retrieve it from the HSM buffers. This notification mechanism is implemented through an Interrupt Service Routine (ISR) that is triggered automatically by the HSM using a special interrupt line. HSM-compatible boards designed for real-time applications, such as automotive ECUs, require non-blocking high-latency operations to meet critical time constraints and prevent system failures. These boards typically have interrupt lines dedicated to event management between

the HSM and the driver, as interrupts offer an established approach for implementing event-driven architectures that are inherently responsive In the present work, the ISR is already defined and can be used by a future developer to include it in the Interrupt Vector Table of the target processor where the driver is deployed. In this way, minimal intervention is required to support the real-time communication between the HSM and the driver.

Actually, the behaviour of the Interrupt Service Routine (ISR) is straightforward, as it serves the singular purpose of activating a designated task specifically intended for the driver: the Notification Task, which intervenes during the last steps of the request lifecycle. In fact, when the HSM informs the driver about the completion of a request by triggering the special interrupt line, the ISR wakes up this task that arranges to extract the output from the HSM buffers by means of the Hardware Abstraction Layer. An additional purpose of the task consists of applying post-processing routines in case they have been defined for the current request type; once the Notification Task has post-processed the results, it is suspended until another request has been completely processed by the HSM.

It is worth mentioning that the driver and the HSM have only two instances of communication throughout the request lifecycle:

- When the request is forwarded to the HSM after passing all initial checks in the driver;
- After the request has been processed by the HSM, so that the driver can retrieve the results and report them to the Crypto Driver Object.

This approach minimizes the exposure of information moving from or reaching the HSM, reducing the information leakage and the effectiveness of a MITM attack.

To define tasks, the presence of a Real-Time Operating System (RTOS) is mandatory. However, the present work does not impose any restrictions on the choice of Real-Time Operating System (RTOS) to be used, aiming to maintain maximum flexibility within the project. For this reason, the Operating System Abstraction Layer defines a light software module whose purpose is to contextualize generic RTOS calls for the target system on which the driver is deployed. Indeed, it has been supposed that, while the HSM itself runs on a bare-metal configuration, the external domain leverages a consolidated RTOS. Not all the functionalities of an RTOS are required for the driver. Indeed, the OSAL implements only basic procedures to create tasks, check privilege levels and handle software signals, which are used to suspend and wake up the Notification Task.

In principle, it is possible to integrate the functionality of the Notification Task directly into the Interrupt Service Routine (ISR). However, this approach poses a scheduling risk as the ISR would hinder the execution of all tasks, including critical ones, until its completion. It is important to highlight that the Notification Task

primarily serves the purpose of transferring result data, and certain operations, such as CBC mode encryption, involve processing large volumes of data, resulting in an ISR with excessive latency. Consequently, the ISR has the potential to disrupt the overall system behavior and inadvertently cause priority inversion. To mitigate this risk, the Notification Task has been designed to prevent failures caused by the HSM Firmware

As mentioned above, the driver provides the further capability of applying post-processing algorithms to output data before returning it to the caller task. Specifically, the Crypto Driver Object can benefit from dynamically selected routines that are systematically applied after processing a request of a specific type, thus simplifying the internal logic of the Crypto Driver Object.

To enable this feature, the driver supports a set of callbacks, one per request type, that can be configured at run-time by the caller task. During the initialization phase, the callbacks are instantiated and associated with the driver, rather than with the specific task utilizing them, making them global in scope. When the Notification Task updates the caller's memory with the request results, it also checks whether a callback has been defined for that type of request. If that is the case, the output is post-processed according to the caller's policy. After this final step, the driver and the HSM are not busy anymore and can process new incoming requests.

It is important to highlight that all callbacks are globally defined and directly associated with a driver instance, rather than being specific to individual tasks. As a result, the callbacks are shared among all potential users of the HSM. This implies that when multiple tasks utilize the driver concurrently, it can result in inconsistencies in the driver's behavior Consider the scenario where two tasks utilize their respective callbacks to post-process a key update operation. Due to the lack of awareness between the tasks, one task may mistakenly assume that the current callback being used is the intended one, even though this cannot be guaranteed. Even defining the callback immediately before submitting the request would not resolve this issue. In a situation where two tasks simultaneously need to submit the same type of request, a race condition would occur, making it unpredictable which callback would ultimately be utilized after the operation. This condition is visually depicted in Fig. 4.7.

However, this problem is not a limitation for the driver in the current case. It is secure to assume that, if the driver is used in a project that is compliant with AUTOSAR's Security stack, then it is known that one and only one task will use the HSM: the Crypto Driver Object associated with the module itself.

Finally, Fig. 4.8 reports the entire lifecycle of a request from the point of view of the driver.

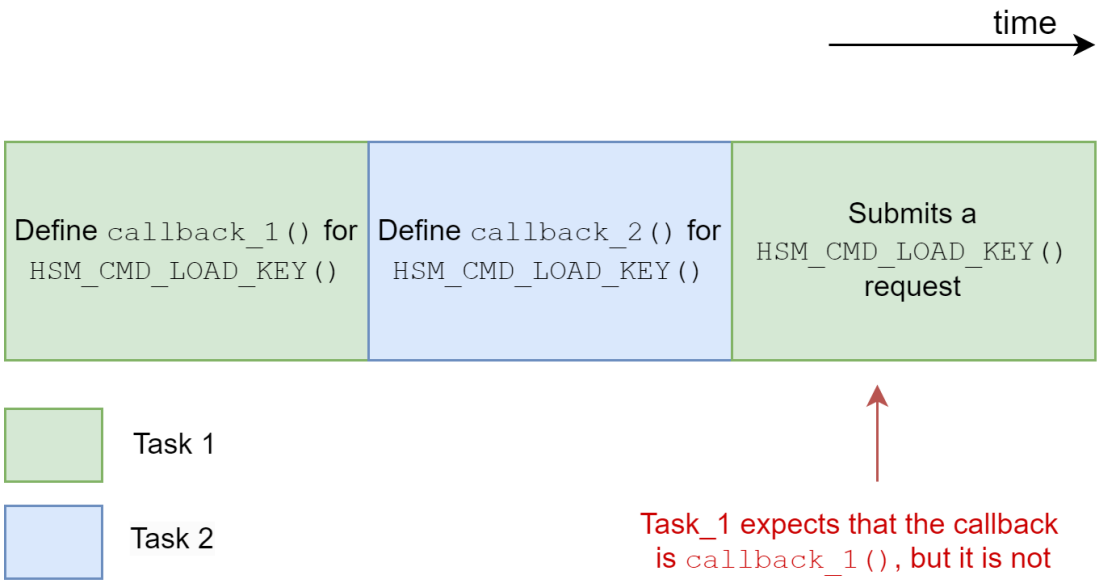

Figure 4.7: Race condition with callbacks if multiple tasks use the driver

4.3 The HSM

4.3.1 The HSM Manager

Once the driver successfully forwards the requests coming from the Crypto Driver Object, they undergo processing within the Hardware Security Module. It is within the HSM that the requests are executed and the corresponding results are computed. Indeed, the HSM module of the project represents the core of the peripheral and enables all the security-critical features required by the specifications.

Essentially, the only purpose of the Hardware Security Module is to fetch new requests coming from the driver, identify their type and arguments, execute them and eventually expose the results. Fig. 4.2 depicts the internal structural architecture of the module.

The HSM Manager holds all the internal logic to handle incoming requests; essentially, it is a Finite State Machine. This FSM is initialized in an Idle state, representing the fact that the HSM is waiting for a new request to be processed. When the driver issues a new request, it informs the HSM via the Control Register of the HSM, using the HAL. Then, the HSM wakes up and can start handling the request during the Fetch State. First, it fetches all metadata from one input buffer and all input arguments from the other. By accessing request metadata, the HSM can recognize the operation type and subsequently, it can interpret the content of the other input buffer correctly. Indeed, it is recalled that the request arguments

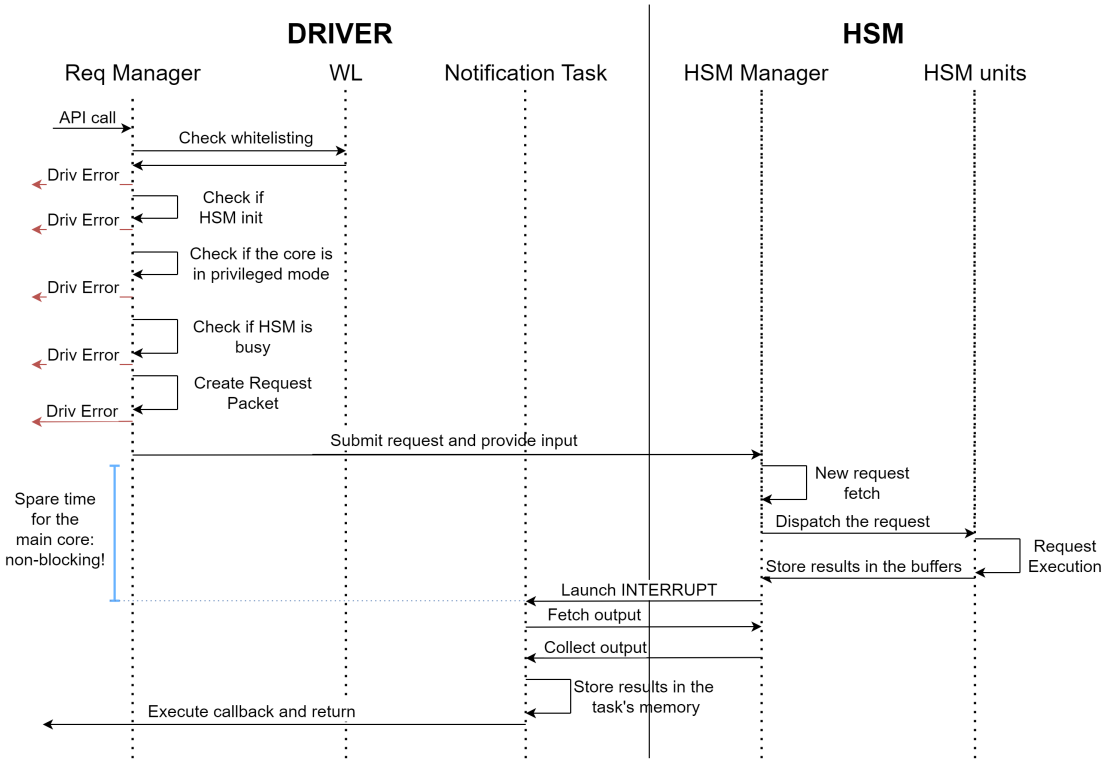

Figure 4.8: Sequence Diagram of the request lifecycle

on the buffer are placed differently depending on the request type, as represented in Fig. 4.6.

After the input has been fetched, the HSM analyzes it for early detection of invalid input data. It has been deemed necessary to check the validity of the input arguments as early as possible, to detect invalid requests that must be rejected. This approach, while avoiding needless processing, would limit the effects of a Denial of Service attack that would benefit from the HSM undergoing high-latency operations, which are reserved for valid requests.

Once the request is deemed valid, the HSM provides for dispatching them to the fitting functional unit of the HSM that shall ultimately fulfil it. In particular, the dispatching strategy follows a straightforward architecture where every kind of request is statically associated with a request handler, i.e. a function whose only purpose consists of fulfilling that specific type of request. So, the HSM enters the Execution State where the corresponding request handler processes the request and stores its results locally.

To provide the HSM Firmware with a high level of flexibility, this module has been designed with a generic architecture in mind for what concerns the requests that the HSM needs to support; in this way, future developers aiming to integrate

new request types can implement the additional logic without any overhead.

Finally, output data must be provided to the driver so that the original caller task can eventually access it. To achieve this goal, the HSM enters a Completion Stage, where the final output is loaded in one of the two output buffers and the updated request metadata is moved to the other. At this point, request metadata holds the return code that can be used by the driver and the caller task to retrieve the final status of the operation and, particularly, whether it succeeded. The last stage of the request lifecycle consists of triggering the special interrupt line connecting the HSM with the external domain so that the driver can finalize the request as soon as the HSM concludes its routines. After triggering this interrupt, the HSM returns to the Idle State and a new cycle of the FSM starts.

About error management, the HSM prevents any invalid request from reaching the Execution Stage; when it detects that the request must not be processed, the HSM skips the Execution Stage and enters the Completion State directly, which is still necessary as this latter stage is where all output is prepared to forward it to the driver. In this way, the driver can retrieve the results and recognize that the request ultimately failed.

Fig. 4.9 depicts the state transitions of the HSM Manager. The initialization phase represents the boot time of the HSM, where it configures itself to enter the nominal state cycle.

t is important to note that all the registers and buffers mentioned for the HSM are defined at a logical level rather than in hardware. This means that an HSM-compliant device is not necessarily required to support the exact hardware interface described. Instead, it is the responsibility of the developer to define the access mechanisms for these memory areas, adapting the HAL to the specific target board.

4.3.2 The Crypto Unit

After the HSM has effectively retrieved and examined an incoming request, it can assign it to either of its two functional units depending on the specific type of request. First, we will focus on the Cryptographic Unit, also known as the Crypto Unit. This functional block is primarily activated whenever the HSM is required to perform operations related to AES-128, such as:

- Encryption/Decryption in ECB mode;
- Encryption/Decryption in CBC mode;
- Key derivation for the key update;
- MAC generation and validation;

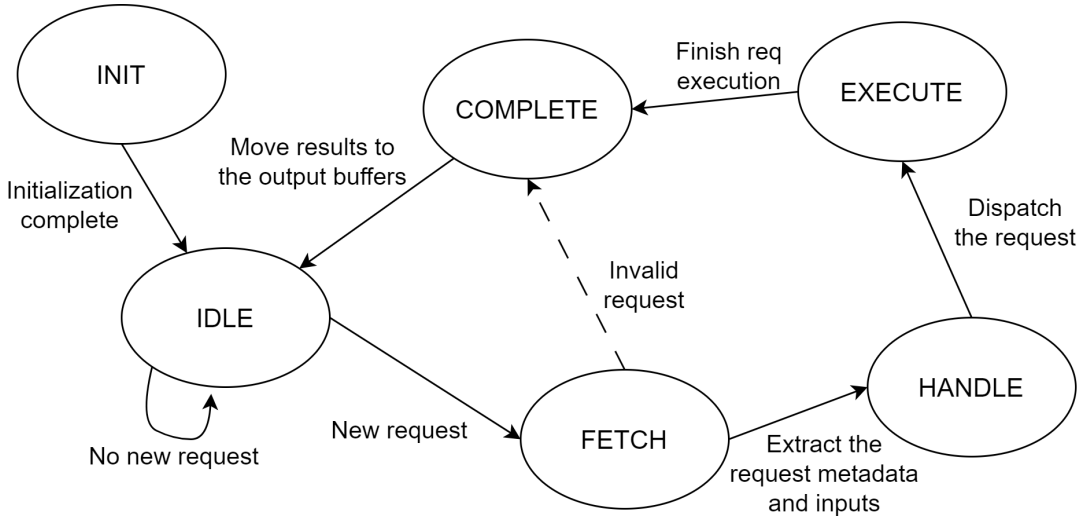

Figure 4.9: The HSM Manager as a Finite State Machine

Encryption and decryption, both in ECB and CBC mode, strictly follow the original definition of the AES-128 algorithm, with no further modification. Inside the HSM, messages whose size must be a multiple of the block length are processed by using a secret key that is chosen by the caller. In particular, an application task shall use an integer index that represents the target key to use. As described later in sec 4.3.3, the HSM holds a constant number of keys that are accessible via their index, so that no key is provided in plaintext by application-level tasks, as illustrated in Fig. 4.10.

About the message size, it is up to the caller to ensure that the input message size is an integer multiple of the block. To ensure this, padding such as PKCS#7 [40] can be used to detect the padding bytes easily. In the current work, it is assumed that user-level tasks provide properly padded messages, or at least the Crypto Driver Object pre-processes them so that the driver receives a padded input. Otherwise, the request is still forwarded to the HSM, but the latter will reject it.

As reported in sec. 3.2.2, block cyphers such as AES-128 can be used as a basis for MAC generation algorithms. Indeed, NIST illustrates how to build CMACs starting from AES [33]. AUTOSAR, starting from the latter guideline, defines the CMAC mode as the reference strategy for checking message authentication and integrity [4]. Consequently, the current project follows the same guideline as AUTOSAR to comply with SHE specifications.

When computing the CMAC for an input message, the HSM utilizes a secret input key to initiate the processing. In the current implementation, this key is securely obtained from the stored keys within the HSM. The CMAC generation requests accept an integer value as the identifier for the secret key, ensuring that

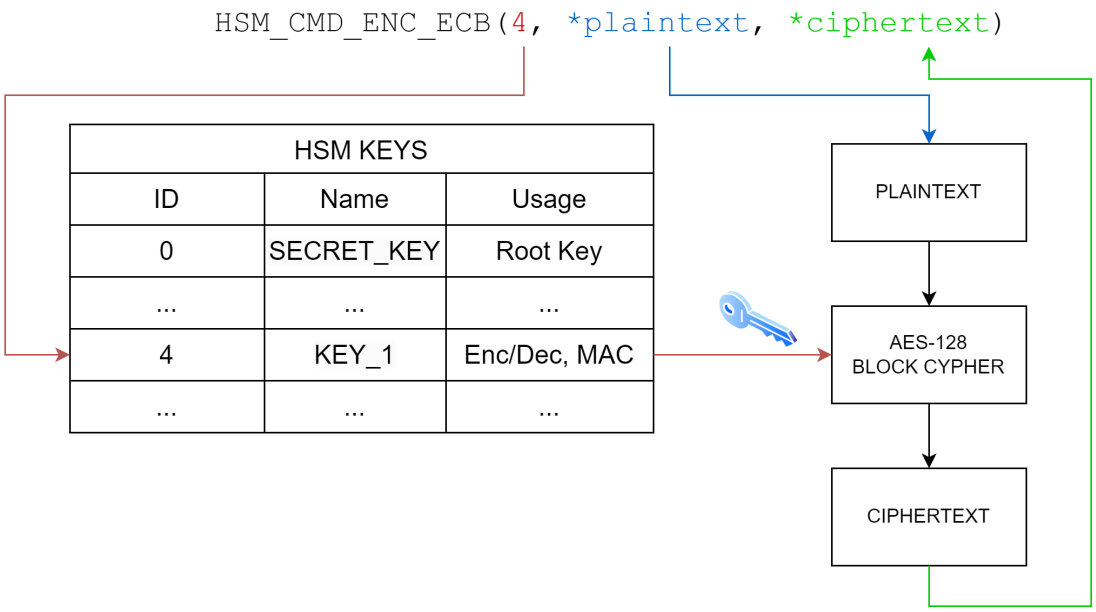

Figure 4.10: Example of Encryption in ECB mode inside the HSM (the API call is used to show how input arguments are actually propagated to the HSM, it cannot access the HSM directly)

no critical information is generated or transferred out of the Hardware Security Module. Once the inputs are defined, the algorithm proceeds by computing two subkeys. These subkeys are used in two distinct steps. A first key is used to apply AES-128 in CBC mode to the input message, while the second one is used during the last step when the latest output is further encrypted in ECB mode using the second key. In particular, the first key corresponds to the original input key, while the other is computed using such a key and an irreducible binary polynomial.

Given the two keys, the algorithm can process the message, whose size may not correspond to a multiple of the block length. If that is the case, a sequence of 10^j is appended to the last block of the message, where a^k stands for the repetition operator and the product represents a concatenation.

Then, the input message undergoes encryption in CBC mode using the provided key for the block cypher. Notably, the last block is subjected to a distinct processing method, where the second key is XORed with the last block prior to the application of the final block encryption. To visualize the mechanics of the CBC-MAC generation algorithm, a thorough reference can be found in [33].

Upon the successful implementation of MAC generation, incorporating support for MAC validation becomes a straightforward task. The MAC validation process involves taking a message and its corresponding MAC, recomputing the MAC

of the input message by using the MAC generation algorithm and subsequently comparing it with the provided MAC. If the computed MAC matches the provided MAC, the MAC validation algorithm affirms the validity of the inputs by returning a positive result.

Notably, when dealing with commands involving data of potentially infinite size, the HSM imposes a limitation by setting a maximum size for input messages. This constraint serves two crucial purposes: firstly, it establishes an upper limit for the latency of these operations, enabling an assessment of the HSM's real-time capabilities; secondly, it acts as a preventive measure against potential DoS attacks. By imposing a maximum size, the HSM mitigates the risk of being overwhelmed by requests that require extensive computations, which could otherwise hinder its ability to process subsequent requests. The current implementation allows developers to define the maximum allowable size of input messages, providing the flexibility to evaluate the trade-off between latency and reactivity for the specific implementation.

Finally, there exist boards supporting hardware accelerators for security-critical operations [42]. The current work would benefit from the possibility to leverage optimized hardware to perform computationally heavy operations such as encryption and MAC generation. For this reason, a future developer can decide whether to implement the Crypto Unit in software or in hardware; in the latter case, they shall define how to interact with the hardware accelerator and how to perform a single AES-128 encryption in ECB mode. Apart from that, all other derived operations, such as AES-128 in CBC mode or MAC generation, are automatically programmed to use AES-128 in ECB mode as their starting point. Considering that the algorithm is executed in hardware and that to trigger an encryption cycle only minimal peripheral configuration shall be prepared, it should be possible to use these hardware accelerators with very low overhead from the developer's perspective.

4.3.3 The Key Manager

Along with the Crypto Unit, the Key Manager is the other internal component of the HSM that is responsible for fulfilling requests from outside. In particular, this module focuses on the management of secret keys, their storage and use. According to AUTOSAR specifications [4], a Secure Hardware Extension shall define a constant number of secret keys, with different permissions and storage requirements. Indeed, the HSM-protected keys are not equal and can be distinguished into various categories:

- Keys for encryption, decryption, MAC generation and validation using AES-128;

- Critical keys, acting as master keys when updating other protected keys;
- Temporary keys used for intermediate computations, but that must not be used for other security-critical purposes.

Besides, since SHE specifications also define how to implement features such as Secure Boot and Pseudo Random Number Generation, they also include special keys for RNG seeding and to validate the Secure Boot procedure. However, they are out of the scope of the project and thus not reported further. The entire key management system strictly follows SHE specifications [4].

Within the scope of this project, there are 15 keys defined by AUTOSAR, with different purposes as explained above. Among these keys, there is a root key, called Secret Key, which is read-only, meaning its value is hardcoded and extracted from a unique identifier for the ECU. This has significant implications when it comes to key updates. It is noted that the mechanism to access this key is hardware-dependent, thus it is included in the HAL to allow a fast adaptation of the firmware to the underlying platform.

The update protocol, which is invoked when issuing a load, update or remove request, involves the target and an authentication key, where the latter must be used to verify the trustworthiness of the entity requesting the key update. Each key has a predefined set of authentication keys, established during the design phase and unchangeable thereafter. Consequently, a hierarchical dependency exists among the keys, as illustrated in [4], with the Secret Key acting as the root that enables access to the others. The use of an unpredictable unique identifier necessitates that the caller task possesses knowledge of this secret so as to access the HSM key storage, thus providing an additional layer of authentication. Without this secret, any task would be capable of potentially accessing and modifying the HSM keys, since all of them would be exposed without requiring theft or unauthorized acquisition.

The entire key update protocol is thoroughly described in [4]; it is based on the combination of several strategies to ensure protocol security such as:

1. Key derivation, achieved by using a special compression algorithm based on AES-128 applied with the Miyaguchi-Preneel scheme, called AES-MP; given the target key k_i to update, the requesting task must know the current value of k_i in order to modify it. Thus, starting from this value, the requester needs to compute two derived keys that shall be used for the next steps of the algorithm.

2. Encryption using AES-128, which is applied to the input request containing the new value for the target key. One of the derived keys is used in this step to encrypt the input message.

3. CMAC usage to verify message integrity, by using the AES-CMAC algorithm reported in sec. 4.3.2. Here, the other derived key is used to compute the CMAC of the input message.

During these steps and especially during key derivation, it is required to store intermediate keys without exposing them outside of the HSM. For this reason, the HSM Key manager offers a Ram Key, whose content can be loaded in plaintext; once loaded, this key is stored inside the HSM and can be used for any purpose, including encryption and MAC generation. It is worth noting that this key shall be used sparingly and only when strictly needed, e.g. when the user task must authenticate the key update request by uploading the current value of k_i for the subsequent key derivation step. Notably, the current value of k_i would be exposed while it is being loaded to the Ram Key; however, it is supposed that the communication channel between the external domain and the HSM cannot be observed by an external task.

Moreover, exposing the key implies a higher risk in the case where the attacker can spy the memory of the external domain by means of a debugger. For this reason, AUTOSAR mandates the implementation of debugger detection mechanisms to forbid any HSM operation in case it is present. In the current work, no debugger detection mechanism is implemented by default since it is hardware-dependent, thus defining it is up to the developer. A possible future work can consist of integrating a debugger detection mechanism in the Hardware Abstraction Layer of the HSM, thus facilitating this integration step.

About the keys, there are two additional keys, not expected by AUTOSAR, which shall be used to encrypt the keys themselves before storing them during a key update procedure. This additional overhead is deemed useful to prevent side-channel attacks consisting in accessing the HSM memory by means of physical tampering. As for the Secret Key, the mechanism to retrieve these two hardwired special keys is up to the specific implementation and is thus included in the HAL.

4.3.4 The Memory Protection Unit

After examining the functional units of the Hardware Security Module and their cooperation with the HSM Manager, we can now focus on the final component of the module: the Memory Protection Unit (MPU). Memory Protection Units (MPUs) are well-established technologies used for monitoring memory access. They offer a fast and reliable hardware-based solution.Essentially, MPUs consist of a software-programmed memory map that is stored in the peripheral. When a new load or store instruction is issued, the Memory Management Unit uses the MPU to check for the privileges and access rights of the caller. In particular, there exist MPUs that are able to check whether the target memory region is accessible in user mode or if, instead, it can only be used when the processor is in privileged

mode [39]. Additionally, it is possible to specify whether a region is read-only, write-only, or, in the case of code areas, execute-only [43].Currently, many Real-Time Operating Systems and processors include MPUs for handling security-critical operations [38]. The current work leverages the capabilities of widespread ARM Cortex-M cores to implement the HSM. Not surprisingly, ARMv7 cores such as many processors of the Cortex-M family already embed a Memory Protection Unit on their own [39][43]. Hence, it is possible to use the underlying hardware to implement memory protection in software, too.

Before integrating the MPU into the firmware, it is essential to determine the specific areas that the MPU needs to protect. Moreover, it is also required to select the granularity of the memory protection. Indeed, ARM-defined MPUs can accommodate up to eight memory regions for protection, with each region's size being a power of two, denoted as 2^k bytes, where $k \in \mathbb{N}$. For this reason, four non-overlapping memory regions are defined in the HSM Memory Map:

Code Area It holds all the code and the read-only data structures that could be stored in a ROM. It can only be used in read-only mode.

Data Area It contains all data that can be stored in a RAM, i.e. zero-initialized data, variables and, in general, any information that can be deleted at power-off. The RAM Key is also allocated in this memory region, in contrast with the other protected keys that are defined in the Flash Area.

I/O Area It represents the memory region that exposes the memory interface to the driver. Logical registers and I/O buffers of the HSM are allocated to this memory region. Even though the driver interface and the data area share similar privileges, as they are both R/W regions accessible only in privileged mode, the driver interface can be defined as a device region. Indeed, the microcontroller allows defining regions for memory-mapped peripherals, which shall not be cached because of their volatility. Moreover, memory-mapped peripherals must lie within specific address ranges, which are not compatible with those for RAM R/W regions [43].

Flash Area All protected keys must persist between one power-on to power-off cycle and the other, otherwise their content would be easily predicted at boot-time, encouraging attacks during the early stages of activity of the vehicle, i.e. while it is turning on. Thus, a non-volatile memory region for special keys is defined specially.

All these memory regions must be accessed in privileged mode, to guarantee a higher degree of internal security. Fig. 4.11 displays a typical Memory Map for an ARM Cortex-M core, based on the memory map of an STM32F4 microcontroller.

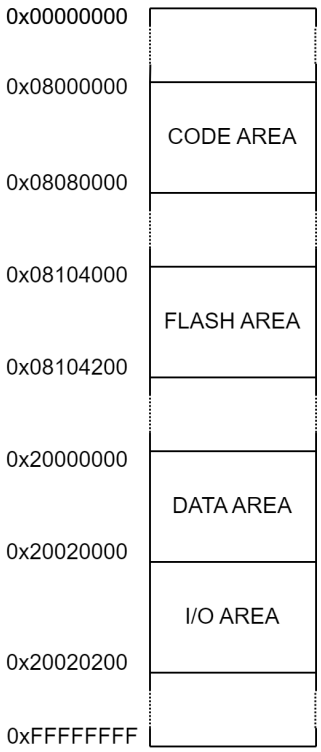

Figure 4.11: HSM Memory Map for an ARM Cortex-M core (based on an STM32F4 microcontroller)

Any ARMv7-M core supports up to eight memory regions in their MPUs [39]. Thus, in addition to the four main memory regions, a developer could define up to four custom regions for special use. For instance, memory-mapped cryptographic units for encryption and MAC generation could use a special address range that does not overlap with any of the other default regions. If that is the case, the developer needs only to define how many custom regions are required and which address range they are linked to.

It is worth noting that it is impossible to find a memory map that fits any existing board, even if they use the same microcontroller. Indeed, memory-mapped peripherals are hardware-dependent and so their address range is since there is no standard regulating address range allocation for peripherals. Hence, a future developer must know which is the HSM memory map and, for this purpose, they must define the address ranges themselves.

To enhance portability, the project simplifies this process by allowing developers to define only the address ranges. A predefined scatter file, following Armlink's

specifications, automatically retrieves these ranges and utilizes them during link-time before deploying the firmware to the target board. Once the HSM Memory Map is defined, it is possible to integrate the MPU into the firmware. In order to use the MPU, the latter is configured at boot-time, during the initialization stage of the HSM Manager. At configuration time, the MPU stores the statically defined Memory Map, which is immutable once the HSM boots, together with the access rights of each region. Then, after a successful configuration, the MPU is enabled: after this step, the MPU will seamlessly monitor memory access and recognize any attempt to use a memory location which is covered by no region in the memory map.

In the architecture of the HSM, the Memory Protection Unit acts as a safeguard during the execution of a request. Since the only access point to the HSM consists of its interface towards the driver and the latter updates this interface once a new request is forwarded to the HSM, it is during the execution of the latest request that the HSM could be attacked. A potential attack could consist of a memory overflow, which commonly occurs when invalid memory accesses are made beyond the permitted range. In such cases, the MPU can play a crucial role in detecting and promptly terminating the current request upon detecting this condition.

Throughout the execution of a request, the MPU remains constantly active, monitoring memory access and thwarting any unauthorized attempts. If illegal access is detected, the MPU promptly triggers an interrupt, which subsequently activates an ISR. The sole objective of this ISR is to raise a flag, used by the HSM to recognize that a memory access violation has taken place. Indeed, the HSM and the MPU work asynchronously, as memory violations could be detected at any point during the request lifecycle in the HSM. Thus, the HSM periodically polls this flag to check for occurred violations. If detected, the HSM Manager immediately stops the execution of the current request and skips to the Completion State, where the occurrence of an MPU violation shall be signalled to the driver.

Once successfully configured, the MPU is enabled, allowing it to actively monitor memory access and identify any attempts to access memory locations not covered by any region in the memory map. Otherwise, a potential attacker could paralyze the module by managing to forward a dummy illegal request to the HSM that would trigger a memory violation. To improve recovery after the detection of a memory access violation, the HSM especially checks for violations before high-latency operations, e.g. key update or encryption.

It's important to note that the driver, located in the external domain, lacks a memory protection mechanism.Indeed, it is supposed that the external domain will utilize an RTOS, which will incorporate its own mechanisms for memory protection. Furthermore, the memory map of the external domain is considerably more intricate and unpredictable compared to that of the HSM, as it necessitates knowledge of the architecture encompassing the entire software operating within the ECU. Therefore,

it is the responsibility of the developer to determine whether a memory protection policy should also be applied to the driver.

Chapter 5

Final implementation and conclusions

5.1 Implementation and Testing

After defining the fundamentals of the HSM Firmware project, how it fits within the automotive context and, in particular, the AUTOSAR specification framework, a working implementation of the project can be realized. So, this chapter proposes how the previously obtained design can be put into practice.

One of the hardware prerequisites specified in this project pertains to the board selection for deployment. Specifically, the HSM Firmware project is designed to be compatible only with boards that have the inherent capability to accommodate a Hardware Security Module. The project assumes the existence of a dedicated core where the HSM Firmware can operate concurrently with the external domain. Additionally, it is essential that a private channel exists between the HSM core and the external domain, exclusively allowing communication between the HSM Driver and the HSM itself. This ensures that malicious individuals are unable to eavesdrop on the inputs and outputs of incoming requests.

Regrettably, a suitable board was unavailable during the project's development, necessitating adaptations to enable development using a more commonly available board for validation. As a consequence, two different *Execution modes* have been defined, starting from the same design:

Simulation mode The version of the project resulted from the equipment constraints. In Simulation mode, the entire project runs on a single core, i.e. both the driver and the HSM share the same microcontroller. Notably, this mode implies that a scheduler must be present to allow the driver and the HSM to run concurrently.

Real mode The expected deployment of the project, where a suitable board is used and is possible to distribute the HSM code in a special core for security-critical operations, while the driver is usable in the external domain and acts as the only access point to the device.

The current work supports both modes; however, it requires following distinct deployment procedures to prepare the target board for hosting the project. These procedures are further elaborated in the source code.

Regarding the Simulation mode, since it runs on a single core, it is required to parallelize the execution of the driver and of the HSM; thus, a scheduler is needed. This additional requirement translates into the necessity to use a Real-Time Operating System (RTOS) whose scheduling capabilities can ease the subsequent adaptation of the project. Using an RTOS, we can define two separate tasks, one for the driver and one for the HSM, that can run in parallel.

In the current work, the RTOS has only the purpose to parallelize two separate tasks, no additional features are strictly required; so any solution would suffice, and choosing one RTOS over another comes down to a matter of choice. In the end, FreeRTOS was chosen, partly because it is free, open-source, and easy to access. Since the implementation of the Operating System Abstraction Layer (OSAL) of the driver still requires an RTOS, the driver can use it in order to handle its notification task. Thus, in the current implementation, the driver OSAL uses FreeRTOS primitives.

The entire project has been developed using μVision 5 [44] as the IDE for development and testing purposes. Moreover, this IDE has been used to emulate the target microcontroller during the earliest stages of the process, thanks to its emulation capabilities that can reproduce the behaviour of an ARM Cortex-M core.

More precisely, the deployment of the current project has followed two subsequent steps:

1. First, the project has been emulated using μVision 5 in order to mimic a Cortex-M3 microcontroller and validate the HSM Firmware. Interestingly, the IDE inherently supports several RTOSes, including FreeRTOS. Thus, it was straightforward to embed a Real-Time Operating System inside the emulated environment.

2. As a second and last step, the project has been ported to a STM32 Nucleo-144 board, which mounts an ARM Cortex-M4 microcontroller [45] and is also reported in Fig. 5.1. This microcontroller is compatible with the hardware requirements regarding the target architecture since it belongs to the ARM-v7 family [39]. For further reference about the board, see [45].

The final stage of deployment serves a dual purpose: not only does it involve adapting the design to a specific physical board to validate the design from a physical

perspective. Indeed, it is important to recall that one of the key objectives of the project is to ensure portability across multiple target boards, thereby minimizing the effort required by programmers to tailor the codebase for each individual target system.

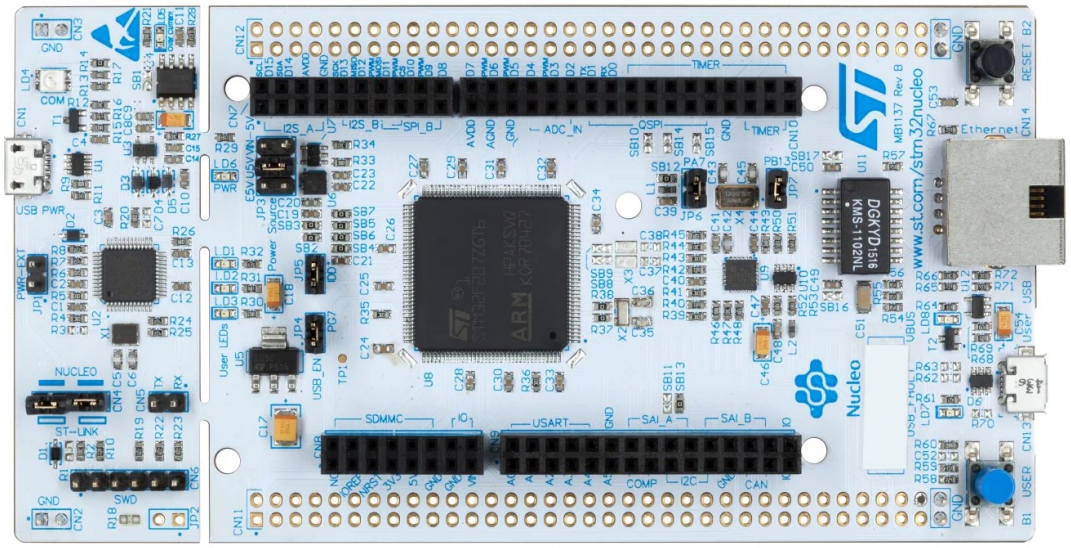

Figure 5.1: The STM32 Nucleo board used for the project (Source: *STMicroelectronics*)

The project was carried out successfully by modifying only the SW modules that were designed to be dependent on the underlying hardware and, therefore, customized by the developer. These modules include the Hardware Abstraction Layer, the Operating System Abstraction Layer, and the configuration files that define the memory map compatible with the board's memory layout.

About the codebase, only the library implementing AES-128 comes from third-party developers; it corresponds to the *Tiny AES-C* library that can be found in [46]. In particular, the library deals with AES-128 encryption and decryption in ECB mode and CBC mode. Other cryptographic operations supported by the HSM Firmware, such as MAC generation/validation and key derivation, were implemented from scratch with the library serving as a supporting tool. During both the development steps, it is fundamental to validate the work done thus far, in order to prove the effectiveness of the project. To this end, it is necessary to define a comprehensive test campaign ranging from individual components to operational scenarios, to validate the design on multiple levels of abstraction.

A comprehensive testing campaign has been defined, following a bottom-up approach:

1. Unit tests, involving every single functional unit without interaction with the

other submodules. They involved components of the driver, the HSM and the abstraction layers to facilitate software debugging during the subsequent testing stages. About the HSM, AUTOSAR provides input vectors to validate both cryptographic and key management functions [4]; they have been used during the test campaign.

2. Integration tests: in particular, a set of tests has been defined to validate the data flow inside the HSM, from the moment when a new request arrives to the time when output has been computed and is made available on the output buffer.

3. System tests: here, the entire request lifecycle is verified, for every possible request type and input. However, the MPU is disabled during these tests: since the MPU memory map assumes that only the HSM-related code is running on the target microcontroller, any activity of the driver or of the operating system could result in an MPU violation, which would not occur in case the system was deployed in Real mode. During these tests, the test routine impersonated the Crypto Driver Object, i.e. the routine was forcibly mapped to the whitelisted memory region, thus validating this authentication strategy. When simulating operational scenarios, the test routine interacted in a synchronous manner to simplify test design and implementation by using the `HSM_IS_BUSY()` function to recognize whether the driver was ready to process the next step. Hence, the driver can be used both in asynchronous and synchronous modes by means of this API function.

For this project, a custom implementation of the *ltest* framework by Martin Bloedorn [47] was utilized as a testing framework. This framework offers a JUnit-like testing environment for projects based on the C programming language. The current project provides, together with the HSM Firmware source code, both the testing framework and test suite used for validation in order to allow future developers to check that their design is compliant with the project functional specifications without the need of redefining a custom test campaign.

It is worth noting that all the tests described thus far only validate the functional properties of the project, without focusing on latency or worst-case execution scenarios. Indeed, the only purpose of implementing the Simulation mode is to enable software development and testing even if it is not possible to use a suitable board that follows the specifications reported in sec. 3.1. If such a board is not available, it is deemed worthless to delve into timing analysis and optimization.

5.2 Conclusions and Future Work

After introducing the HSM Firmware, how it has been designed and its final implementation with a subsequent testing campaign, the HSM Firmware has been thoroughly described. The success of the testing campaign and the capability of the system to be adapted to different hardware environments proves the portability of the system. These results encourage to further expand the number of boards that are officially supported in order to make the current work a reliable open-source project that vehicle manufacturers could use to speed up the development of security-critical components.

Moreover, by complying with the AUTOSAR framework, it should be possible to use the current project in AUTOSAR's Security stack reported in [2]. In particular, this framework can be considered as a reliable reference point for ECU software development in the automotive context; the current project aims to be integrated into such a secure software environment.

Interestingly, the testing campaign highlighted the capability of the current work to be used both in asynchronous and synchronous mode, thus making it slightly more adaptable to the internal logic of the Crypto Driver Object, which is the only task allowed to interact with the driver as by AUTOSAR specifications [2].

Also, the success of the testing campaign proves the functional capabilities of the system, especially in request handling, key management and cryptography. However, non-functional properties such as the worst-case latency for all operations could not be verified because of the unavailability of a suitable board satisfying the hardware requirements of the project. However, this did not stop the project development, which focused more on the product's functional features.

This constraint itself reveals a potential area for improvement in the current work: exploring and analyzing the project's performance using a board with at least two separate processors. In particular, one of them should be designed for security-critical operations or, at least, a private communication channel with the surrounding hardware; this property would ensure that the communication between the driver and the HSM can not be monitored by a malicious task.

However, the project can be subject to several improvements which have not been covered by the current work. Indeed, it is recalled that AUTOSAR's framework includes additional security features in its specifications for Secure Hardware Extensions [4], which could further improve the capability of the HSM Firmware project while providing software compatibility with SHE.

A notable example of such improvement is the support for Secure Boot verification [4], which consists of the capability to validate the image of the Operating System that is running on the ECU. Typically, this verification procedure involves a cryptographic unit that must implement MAC generation and validation functions, which are known for being a useful tool for checking the integrity of a given

bitstream.

In this case, the bitstream corresponds to the entire image of the Operating System, which can be condensed in a byte array with almost no memory overhead, e.g. 16 bytes in case the MAC is computed using the MAC generation algorithm involving AES-128. At boot time, the Secure Boot verification procedure computes the MAC corresponding to the detected OS image and, if it matches with the expected digest, the system boots as expected; otherwise, a major failure is detected and the system will not boot at all, thus preventing a potentially malicious OS image from controlling the system.

AUTOSAR's specifications for SHE [4] recognize the importance of checking the running OS image of the system; as a consequence, it expects that a thorough implementation of a Secure Hardware Extension must detect a Secure Boot condition. By leveraging the current cryptographic capabilities of the HSM Firmware project, i.e. the possibility to generate MACs using AES-128 and validate them subsequently, a future developer could focus on this feature without the overhead of designing a cryptographic unit from scratch.

Regarding the cryptographic unit, it is known that the HSM Firmware supports message encryption in CBC mode using AES-128; however, it is reported in sec. 4.3 that the current version of the firmware requires that the size of the input message to either encrypt or decrypt must be known a priori. This is due to the fact that there is no current support for padding of input messages, e.g. by using PKCS#7 [40].

Padding serves as a valuable mechanism to adjust the input length to conform to the expected protocol requirements, such as the 16-byte length in the case of AES-128. This adaptation allows for a broader range of inputs that can be processed by the algorithm or the HSM. Furthermore, certain padding schemes like PKCS#7 offer an additional advantage by exposing the last block of a message, eliminating the need to determine the message size prior to encryption. By leveraging padding, the HSM can accommodate varying input sizes more effectively and simplify the encryption process. This concept is further illustrated in Fig. 5.2 for clarity.

Therefore, the inclusion of padding could be considered a potential enhancement for future iterations of the present work. Given the widespread support for various padding schemes, it is anticipated that incorporating these schemes would seamlessly integrate with the existing environment.

At present, requests undergo processing within a single cycle. When a valid request is received by the driver, it is promptly forwarded to the HSM. Upon reaching the HSM, the inputs are immediately utilized to generate the corresponding output, which is subsequently sent back to the driver and, eventually, to the requesting Crypto Driver Object. It is important to highlight that this processing cycle necessitates a single data transmission from the driver to the HSM.

Nevertheless, there may arise situations where multiple cycles are necessary to

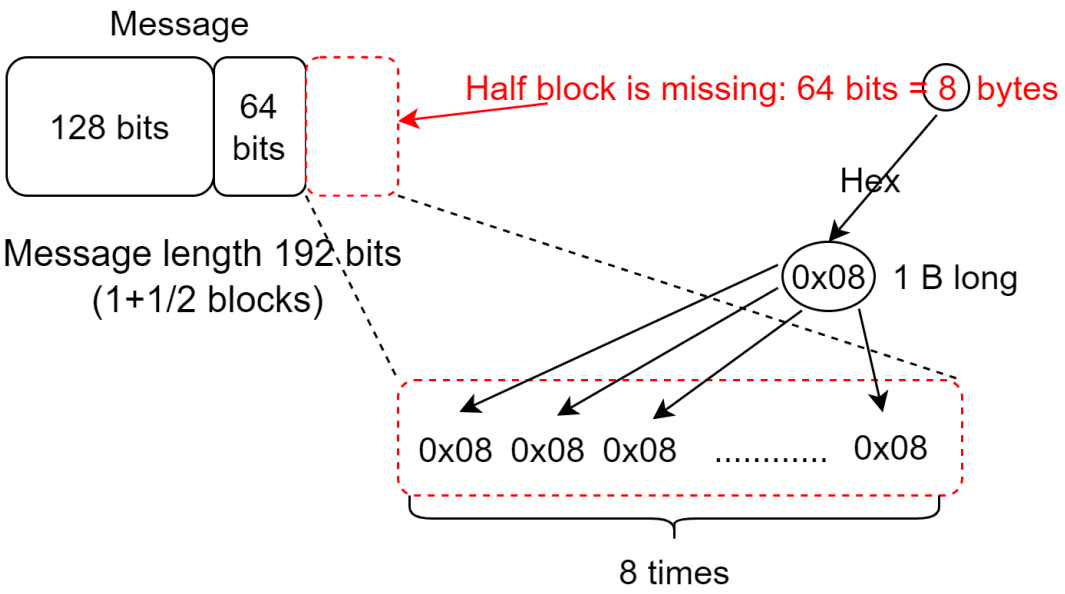

Figure 5.2: Example of message padding with PKCS#7 (It is noted that only the last message follows this pattern)

process all the input data. It is important to consider that the HSM's software buffer has a fixed capacity. Consequently, if a request is intended to be processed within a single cycle, all the input data must fit within this buffer. For request types with fixed-sized input data, such as key updates, this is typically not a critical concern as allocating sufficient memory in the buffer would be adequate. However, this issue can become critical in cases where there is a requirement, dictated by system specifications, to encrypt extensive volumes of data using CBC mode. It should be noted that CBC encryption can accommodate input data of varying lengths, with the developer determining the input length constraint as a trade-off with worst-case latency.

When faced with the limitation of extending the HSM buffers further, a potential solution to address this issue involves implementing a buffering strategy, enabling the processing of input data in multiple cycles. This approach involves multiple iterations and interactions between the driver and the HSM, continuing until the request is completed.

During each iteration, the driver transmits a portion of the input data that has not yet been processed to the HSM. The HSM then computes a partial output for the request and returns it to the driver. The driver, in turn, stores the partial output in the destination memory and prepares the input for the following iteration. This iterative process continues until all the input data has been processed.

It is important to note that this approach is suitable for processing input requests as a continuous stream of data, without the need for all input data to be available before processing. Specifically, in the current scenario, two operations, namely MAC generation/validation and encryption/decryption in CBC mode, benefit from buffering to expand the range of data they can handle. These operations rely on AES-128 encryption in CBC mode, which inherently processes data in a sequential manner, with each block being processed after its predecessor. Additionally, AUTOSAR provides specifications for handling errors when generating output in multiple iterations [4], demonstrating the compatibility of this strategy with the Hardware Security Module.

An additional feature that the AUTOSAR framework describes in [4] is the possibility to generate random numbers, either using a physical noise source or with deterministic algorithms that generate output with a sufficient degree of unpredictability, such as cryptographic functions. By generating unpredictable data, the HSM can use simple primitives to generate highly secure keys whose content is not easy to infer.

Similar to MAC generation and validation, AES-128 can serve as a source of pseudo-random numbers, provided that the PRNG implementation adheres to specific security requirements reported in [4]. The framework defines a separate section of the Secure Hardware Extensions API for PRNG-related functions, such as generating seeds and constructing new random values from them using the aforementioned encryption algorithm.

Finally, a final additional feature could be explored in future versions of the current work. This feature aims to improve the capabilities of the HSM to filter out suspicious requests coming from an unauthorized task. Currently, a whitelisting policy, reported at sec. 4.2, is the main strategy to recognize such requests and reject them. It is recalled that this strategy is based on a check on the return address of the task that called the driver: if and only if such address belongs to a trusted memory region, then the driver continues to process the request, eventually forwarding it to the HSM.

By implementing this strategy, the HSM can effectively detect unauthorized requests originating from memory regions not included in the whitelist.

Nevertheless, it is important to acknowledge that an attacker could potentially manipulate the trusted memory region, such as through code injection within the task. However, executing such an attack would require the Crypto Driver Object to have a vulnerability that could be exploited, such as a buffer or stack overflow. If such an attack were successful, the attacker could assume the identity of the trusted task and interact with the HSM. In this scenario, the attacker could submit invalid requests to render the HSM completely inaccessible.

In such scenarios, the HSM could identify the Crypto Driver Object as unreliable by monitoring the frequency of invalid requests from a specific memory region

within the whitelist. If the number of illegal requests surpasses a predetermined threshold, the driver may decide to reject all requests originating from that region, treating it as an unauthorized memory region. Although this approach does not fully neutralize the attack, it enhances the responsiveness of the HSM and enables it to handle requests from unaffected areas within the whitelist.

Finally, once a more complete HSM Firmware is available, Vulnerability Assessment and Penetration Tests could be performed to finalize the firmware validation also from a security perspective.

Bibliography

[1] *Self-driving Cars Market by Component (Radar, LiDAR, Ultrasonic, and Camera Unit), Vehicle (Hatchback, Coupe and Sports Car, Sedan, SUV), Level of Autonomy (L1, L2, L3, L4, L5), Mobility Type, EV and Region - Global Forecast to 2030*. White Paper. Research and Markets, Jan. 2022 (cit. on pp. xii, 5).

[2] *Specification of Crypto Driver*. Tech. rep. 807. AUTOSAR, Nov. 2019 (cit. on pp. xiii, 24, 33, 37, 43, 47, 67).

[3] *Specification of Crypto Interface*. Tech. rep. 806. AUTOSAR, Nov. 2019 (cit. on p. xiii).

[4] *Specification of Secure Hardware Extensions*. Tech. rep. 948. AUTOSAR, Nov. 2019 (cit. on pp. xiii, 14, 18, 19, 22, 23, 25–27, 30, 31, 33–35, 37, 39, 40, 45, 46, 53, 55, 56, 66–68, 70).

[5] *Specification of Crypto Service Manager*. Tech. rep. 402. AUTOSAR, Nov. 2022 (cit. on pp. xiii, 24).

[6] *Achievements in Public Health, 1900-1999 Motor-Vehicle Safety: A 20th Century Public Health Achievement*. MMWR Report 48(18); 369-374. National Center for Injury Prevention and Control, May 1999 (cit. on pp. 1, 2).

[7] C. J. Cleveland and C. Morris. *Dictionary of Energy*. Kidlington, UK: Elsevier Ltd., 2006, p. 473 (cit. on p. 2).

[8] *C-ITS: Cooperative Intelligent Transport Systems and Services*. 2023. URL: `https://www.car-2-car.org/about-c-its/` (cit. on p. 3).

[9] A. Alalewi, I. Dayoub, and S. Cherkaoui. «On 5G-V2X Use Cases and Enabling Technologies: A Comprehensive Survey». In: *IEEEAccess* 9 (Aug. 2021), p. 2 (cit. on pp. 3, 5, 6).

[10] D.A. Rosen, F.J. Mammano, and R. Favout. «An electronic route-guidance system for highway vehicles». In: *IEEE Transactions on Vehicular Technology* 19 (Feb. 1970) (cit. on p. 4).

[11] *IEEE Standard for Information technology – Local and metropolitan area networks – Specific requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments*. IEEE. July 2010. URL: https://standards.ieee.org/ieee/802.11p/3953/ (cit. on p. 4).

[12] H. J. Miller and S. Shaw. «Geographic Information Systems for Transportation». In: (2001). Ed. by Oxford University Press (cit. on p. 4).

[13] A. Papathanassiou and A. Khoryaev. «Cellular V2X as the Essential Enabler of Superior Global Connected Transportation Services». In: *IEEE 5G Tech Focus* 1 (June 2017) (cit. on p. 5).

[14] R. Sedar, C. Kalalas, F. Vazquez-Gallego, L. Alonso, and J. Alonso-Zarate. «A Comprehensive Survey of V2X Cybersecurity Mechanisms and Future Research Paths». In: *IEEE Open Journal of the Communication Society* 4 (Jan. 2023) (cit. on pp. 5, 7–10).

[15] A. Turley, K. Moerman, A. Filippi, and V. Martinez. *C-ITS: Three observations on LTE-V2X and ETSI ITS-G5—A comparison*. White Paper. NXP, 2019 (cit. on p. 5).

[16] R. Molina-Malegosa, J. Gozalvez, and M. Sepulgre. «Comparison of IEEE 802.11p and LTE-V2X: An Evaluation With Periodic and Aperiodic Messages of Constant and Variable Size». In: *IEEE Access* 8 (July 2020) (cit. on p. 5).

[17] A. Tigadi and M. Chandra. «Literature Review on the Future of V2X Communication in Connected and Autonomous Vehicles». In: *International Journal of Research and Analytical Reviews* 10 (Apr. 2023) (cit. on p. 5).

[18] *NIST SP 1800-25: Data Integrity: Identifying and Protecting Assets Against Ransomware and Other Destructive Events*. White Paper. NIST, Dec. 2020 (cit. on p. 7).

[19] A. Greenberg. *The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse*. Wired. Aug. 1, 2016. URL: https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/ (cit. on pp. 8, 10, 12).

[20] A. Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. Wired. July 21, 2015. URL: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ (cit. on p. 8).

[21] X. Wu, W. Zheng, X. Chen, F. Wang, and D. Mu. «CVE-assisted large-scale security bug report dataset construction method». In: *Journal of Systems and Software* 160 (Feb. 2020) (cit. on p. 8).

[22] *NHTSA Home Page*. 2023. URL: https://www.nhtsa.gov/ (cit. on p. 9).

[23] C. Schmittner and G. Macher. «Automotive Cybersecurity Standards - Relation and Overview». In: *Computer Safety, Reliability, and Security*. Ed. by A. Romanovsky, E. Troubitsyna, I. Gashi, E. Schoitsch, and F. Bitsch. Springer International Publishing, 2019, pp. 153–165. ISBN: 978-3-030-26250-1 (cit. on pp. 10–12).

[24] *SAE J3061 Cybersecurity Guide-book for Cyber-Physical Automotive Systems*. Tech. rep. SAE, 2016 (cit. on pp. 10, 11).

[25] *ISO 26262: Road vehicles Functional Safety Part 1-10*. Tech. rep. ISO: International Organization for Standardization, 2011 (cit. on p. 10).

[26] *ISO 21434: Road vehicles - Cybersecurity engineering*. Tech. rep. ISO: International Organization for Standardization, Aug. 2021 (cit. on p. 12).

[27] F. Oberti, E. Sanchez, A. Savino, F. Parisi, and S. Di Carlo. «PSP Framework: A novel risk assessment method in compliance with ISO/SAE-21434». 2023 (cit. on pp. 12, 15).

[28] C. Pott, P. Jungklass, D. J. Csejka, T. Eisenbarth, and M. Siebert. «Firmware Security Module: A Framework for Trusted Computing in Automotive Multiprocessors». English. In: *Journal of Hardware and Systems Security* 5.2 (2021), pp. 103–113. ISSN: 2509-3428. DOI: 10.1007/s41635-021-00114-4 (cit. on pp. 16–18).

[29] *Building a Secure System using TrustZone® Technology*. Tech. rep. ARM Holdings, Apr. 2009 (cit. on p. 16).

[30] *TPM Main Part 1 Design Principles*. Trusted Computing Group. 2011. URL: https://trustedcomputinggroup.org/resource/tpm-main-specification/ (cit. on p. 17).

[31] H. Raj et al. «fTPM: A Software-Only Implementation of a TPM Chip». In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 841–856. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj (cit. on p. 18).

[32] M. Wolf and T. Gendrullis. «Design, Implementation, and Evaluation of a Vehicular Hardware Security Module». In: Jan. 2012. ISBN: 978-3-642-31911-2. DOI: 10.1007/978-3-642-31912-9_20 (cit. on p. 19).

[33] *Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication*. White Paper. NIST, May 2005 (cit. on pp. 22, 28–30, 38, 53, 54).

[34] O. Abood and S. Guirguis. «A Survey on Cryptography Algorithms». In: *International Journal of Scientific and Research Publications* 8 (July 2018), pp. 495–516. DOI: 10.29322/IJSRP.8.7.2018.p7978 (cit. on p. 26).

[35] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray. *Advanced Encryption Standard (AES)*. en. Nov. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197 (cit. on p. 27).

[36] M. Dworkin. *NIST SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. White Paper. NIST, Dec. 2001 (cit. on p. 27).

[37] C. Shannon. «Communication Theory of Secrecy Systems». In: *Bell System Technical Journal* 28 (Apr. 1949), pp. 656–715 (cit. on p. 27).

[38] A. Lewis. *Benefits of Using the Memory Protection Unit*. FreeRTOS. Feb. 16, 2021. URL: https://www.freertos.org/2021/02/benefits-of-using-the-memory-protection-unit.html (cit. on pp. 30, 58).

[39] *Arm®v7-M Architecture Reference Manual*. ARM Holdings. 2021 (cit. on pp. 30, 58, 59, 64).

[40] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315. Mar. 1998. DOI: 10.17487/RFC2315. URL: https://www.rfc-editor.org/info/rfc2315 (cit. on pp. 38, 53, 68).

[41] *ISO 7816: Identification cards — Integrated circuit cards*. Tech. rep. ISO: International Organization for Standardization, 2005 (cit. on p. 38).

[42] *RM0090 Reference Manual*. STMicroelectronics. 2021 (cit. on p. 55).

[43] *Cortex-M3 Technical Reference Manual*. ARM Holdings. 2010 (cit. on p. 58).

[44] *Keil MDK Home Page*. ARM Holdings. July 2, 2023. URL: https://developer.arm.com/Tools%20and%20Software/Keil%20MDK (cit. on p. 64).

[45] *STM32 Nucleo-144 development board with STM32F429ZI MCU, supports Arduino, ST Zio and morpho connectivity*. STMicroelectronics. July 2, 2023. URL: https://www.st.com/en/evaluation-tools/nucleo-f429zi.html#documentation (cit. on p. 64).

[46] *Tiny AES in C*. Dec. 22, 2021. URL: https://github.com/kokke/tiny-AES-c (cit. on p. 65).

[47] *ltest testing framework*. Apr. 17, 2019. URL: https://github.com/MartinBloedorn/ltest (cit. on p. 66).