POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

AUTOSAR's transformers and MQTT for a fast and secure communication between cars and external networks

Supervisor
Prof. Edgar Ernesto Sanchez Sanchez

Company Advisor
Eng. Jacopo Federici

Candidate
Alessandro Salvatore

April 2022

“Nobody knows what’s gonna happen at the end of the line,
so you might as well enjoy the trip”
Manuel Calavera

Summary

As our cars become more and more complex, so do the electronics that make them work and keep them under control. In the near future communications from the embedded systems within cars, in particular Electronic Control Units (ECUs), to external networks will become more and more indispensable. This kind of communication, often referred to as "Car2X" communication, can have endless applications: signalling system faults and incidents to a road operator, communicating between cars to manage the traffic, telling crossing cyclists that a vehicle is approaching, paying the highway toll on the fly and so on.

Currently the most widespread standard for automotive ECU software, AUTOSAR, does not natively support a standardized way of achieving a fast and secure architecture for communicating with an external network.

To bridge that gap and design such an architecture, I started this thesis project during an internship at Brain Technologies: a consulting company, specialized among other fields in automotive and embedded systems, which lent me resources, knowledge and support for the project. The project went though three phases: the first phase consisted in building knowledge about the AUTOSAR standard and what it provides in terms of cybersecurity operations; then the project was designed and a proof of concept was developed; at last, some tests were conducted to verify the performances of the architecture.

The main tools that were used in the design of the architecture, as the title of the thesis suggests, are the MQTT protocol and AUTOSAR's transformers.

MQTT is a very lightweight and fast communication protocol, often used in Internet of Things (IoT) and embedded systems: this protocol was used since the architecture was designed for automotive ECUs and it is supposed to work also on cars travelling at high speed, so the speed of the connection and the low weight of MQTT were essential for the architecture.

MQTT is based on the publish/subscribe pattern, and it is constituted of two entities: the MQTT client and the MQTT broker. Each message sent through MQTT is associated a queue called "topic". A client, once connected to a broker, can either publish a message in a specific topic (by sending the message to a broker

together with the topic identifier) or subscribe to a topic to receive all the messages that will be sent on that topic. A broker accepts incoming client connections and when it receives a published message, it forwards it to all the clients subscribed to the topic of that message. The clients never communicate with each other, the only connection possible is that between client and broker.

Transformers are AUTOSAR modules that take some data and transform it in some way, hence their name: for example, they can serialize/linearize the data or add a header to them.

In order to secure the message to be transmitted over MQTT, a custom transformer (called "MQTTXf") was developed: the transformer encrypts the data, adds an authentication code and some additional identification information to it and formats it as an MQTT packet.

The cryptographic algorithms used for those operations are provided by other AUTOSAR modules which are part of the so-called "Crypto Stack": the MQTTXf transformer also serves to hide the complexity of the Crypto Stack modules by providing simple and immediate interfaces.

The transformer can be configured to use different encryption algorithms or operating modes, as well as different combinations of authentication and encryption: that way it can be tuned to achieve a different compromise between security and performance, but by default it uses the most secure configuration available.

Transformers can be used as libraries and they can be chained together to process data faster.

The architecture starts from the generation of a message: for example, a sensor detects a fault in the brakes or a specific module requires the geographical coordinates of the car position.

The exchange of those messages within a vehicle must be secure, as it must be protected by modifications due to hardware faults or communication hijacking. In order to guarantee message integrity and data authentication, some functions of a specific AUTOSAR module, the "Secure Onboard Communication" (SecOC) module, were used.

Then the message arrives to the ECU devoted to transforming it: it initializes the MQTTXf transformer and calls the transformer function passing the message and its length as parameters. The MQTTXf transformer is the core of the architecture and the main focus of my development work. In order to secure the message, it initializes the modules that handle cryptographic operations and key storage, fetches the needed keys and identification information, configures the tasks for the modules of the Crypto Stack, makes them process the operations and formats the message as per MQTT specifications.

The message now can be sent, securely, to another ECU which sole purpose is to forward received messages to an MQTT broker. This ECU, as soon as it receives the message, connects to the broker, sends him the message on a predetermined topic and closes the connection.
The broker forwards the message to all the consumer clients connected to that topic (e.g. a road operator or another vehicle), which proceed to decrypt, verify the authentication and integrity of the message and elaborate it as needed.

This architecture has been tested to see if it is up to the strict requirements derived from the identified use cases. The developed transformer was tested to see how different configurations of encryption modes and authenticated encryption combinations generate different overheads for the ECU and take a different time to process a message. Tests on the involved MQTT entities were also performed to verify that it is a communication protocol viable enough to be used on high speed vehicles.

This thesis serves as a compendium to all the work carried out during the six months of my internship at Brain Technologies.
It starts with an overview of the state of the art of all the standards and the technologies used; then it presents the works which explore how to secure the communications within a single vehicle; it proceeds with a detailed explanation of all the developed modules (in particular the MQTTXf transformer), of some use cases of the project and of a possible way of dealing with key distribution; lastly it ends with the results of the different tests conducted on the architecture and a look on possible improvements and further developments.

Table of Contents

List of Tables

List of Figures

Introduction

The vehicles we drive everyday are generally stand-alone systems which work on their own, without any mean for connecting to the external environment. However, in the future the control units that make vehicles work will heavily (if not solely) rely on communications with external networks to function [1].

But, if this is a far future perspective, it is also true that communications between vehicles and external entities can be employed in a more straightforward way for applications that can strongly improve our lives. In fact, many car manufacturers are testing different solutions for making a car talk with the external environment as it is crucial, for example, for autonomous driving or managing the traffic through swarm intelligence. This kind of communication is referred to as "Car2X" or "Car-to-X", with the "X" indicating an unspecified external entity.

But the fact that these solutions are being tested independently by different manufacturers, as they are proprietary solutions, makes it so that this communication cannot be carried out between different cars: the "X" cannot be a car from another manufacturer.

In fact, the most widespread standard for automotive control unit software, AUTOSAR, does not provide native support for achieving such a communication in a fast and secure way.

This thesis presents my design of a fast and secure architecture for Car2X communications.

The core tools of the architecture are AUTOSAR transformers and the MQTT protocol: the former helped me develop an easy interface for ensuring confidentiality, integrity and data origin authentication of the messages exchanged; the latter was used to send such messages to an external entity as fast as possible, given the time restrictions of a running car. A custom AUTOSAR transformer was developed and this made possible, as we will see, to hide all the complexity of the cryptographic modules provided by AUTOSAR and to treat it as a library that could be easily employed in other architectures.

The thesis starts with an overview of the state of the art of all the standards and the technologies used: from a look at the AUTOSAR standard and what it offers in terms of cybersecurity, to a description of the MQTT protocol and an explanation of the cryptographic concepts met within the project.

Chapter 2 explores how, thanks to dedicated AUTOSAR modules, we can achieve security in communications between ECUs, as there is no point in securing the communication with external entities if we do not secure in-vehicle communications first.

Chapter 3 proceeds with a detailed explanation of the architecture proposed, showing all the developed modules and in particular the custom AUTOSAR transformer; then some use cases of the project are shown, together with a possible solution for dealing with key distribution.

Chapter 4 introduces the different tests conducted on the developed modules for verifying both the overhead introduced by the transformer with different configurations and the efficiency of the MQTT protocol; then the results of these tests are displayed and analyzed.

Lastly, chapter 5 closes by summing up the presented work and by proposing improvements and further developments of the architecture.

Chapter 1

Background

In this first chapter we will take a look at the state of the art of the standards and functions used in the development of the main project. We will first look the AUTOSAR standard in general, then we will see specifically what it provides in terms of cybersecurity; next, we will explain the MQTT protocol and lastly we will talk about cryptographic algorithms.

1.1 AUTOSAR

This first section will provide a brief overview of the AUTOSAR standard as a whole with a focus on the different layers of its Classic Platform architecture. This must not be taken as a full explanation of the standard and some aspects that are not relevant to the rest of the document and to the developed work are left undetailed on purpose.

1.1.1 Electronic Control Units

An Electronic Control Unit (ECU), also known as an Electronic Control Module (ECM), is an embedded system in automotive electronics that controls one or more of the electrical systems or subsystems in a car or other motor vehicle [2]. There are many different types of ECUs that can address various tasks: from controlling the amount of fuel ignited to triggering the ABS, from coordinating parking sensors to adjusting the suspensions, and so on; they have every aspect of the vehicle under control and they are at the core of all our cars' functionalities. Modern cars have on average 70 to 100 ECUs [3], with some even reaching 150 (and the numbers will continue to rise) [4].

At the beginning all the ECUs were working independently in their context, then it was discovered that exchanging some useful information between them could improve the quality of the control system and so more and more communication buses were introduced; nowadays ECUs are interconnected with actual in-vehicle networks [1]. In order to fulfill their task and to communicate with others, ECUs rely on embedded software developed specifically to address their job; but different ECUs are made by different OEMs (Original Equipment Manufacturers), which means that code written for a specific control unit would not work on an ECU build by another manufacturer. Also, the increasing number and complexity of ECUs in modern cars makes the development of software for all the control units very long and expensive. Those problems raised the need for a standard that could separate the software from the specific ECU on which it runs: that is the goal of AUTOSAR.

Figure 1.1: An ECU [5]

AUTOSAR

Figure 1.2: AUTOSAR's logo

1.1.2 AUTOSAR history

AUTOSAR is a global partnership of leading companies in the automotive and software industry to develop and establish the standardized software framework and open E/E system architecture for intelligent mobility [6]. It started in August 2002 from a joint effort between BMW, Bosch, Continental, DaimlerChrysler and Volkswagen, with the first version of the standard being drafted the following year. From 2004 to 2013 the standard went through three phases of development, and since 2013 it is being continuously maintained and updated with new features: the last version to date, the one which we refer to and use in this document, is the R21-11 released in November 2021. Since 2002 the number of entities and partners that revolve around AUTOSAR has continued to grow and today 31 automotive OEMs are AUTOSAR partners, 21 of which covered over 80% of the total market revenue in 2019 [7]; in total AUTOSAR counts more than 300 partners scattered all around the world.

The main objective for which AUTOSAR started is to create a standardized middleware between the software applications and the ECU on which they run, thus making software widely independent from the hardware underneath and making code reuse possible; this middleware is referred to as “Basic SoftWare” (BSW). So what AUTOSAR provides to automotive developers is a common development methodology, which in practice translates to requirement specifications, application and basic software interfaces, exchange formats and description templates, means to ensure safety and security of the system and everything needed to document the standard; it is important to understand that it doesn’t provide any actual implementation of functions or applications but just the means and the directions that automotive developers must follow.

1.1.3 Classic and Adaptive Platform

The three phases of development and the further maintenance of the AUTOSAR standard built and are building the so-called “Classic Platform”. The Classic Platform is the standard for embedded real-time ECUs based on OSEK [8], which is a standards body that has produced specifications for an embedded operating system, a communications stack, and a network management protocol for automotive embedded systems [9]. This platform is based on a strict three layers architecture (the Application layer, the Runtime Environment and the Basic Software), which we will discuss later, and its four pillars are: functional safety (which includes communication security), efficiency, being field proven and being performant.

The need for more modern services (like automated driving) and support for different hardware than embedded ECUs (such as Graphic Processing Units for more efficient parallel processing) was the reason behind the release of the Adaptive

Platform in 2017. The Adaptive Platform is service-oriented and defines the means for building services that require high computing power but are not so strict in terms of real-time requirements (the opposite of the Classic Platform) [6]. In this platform there's no distinction between runtime environment and basic software: all the middleware between the applications and the underlying infrastructure (which could also be an hypervisor or a container) is wrapped inside a single layer called AUTOSAR Runtime for Adaptive Applications (ARA). The core of the Adaptive Platform is an operating system based on the POSIX standard, and its services are developed and written in C++ [8].

In 2016 it was also released the "Foundation" standard, which contains all the artifacts that the two platforms have in common, such as the objectives and the main requirements, the meta-model, the debugging specifications, the main protocols used, etc. [10]

For the scope of this document, since all the work done was developed and tested on embedded ECUs with precise real-time requirements, for now on we will only refer to the Classic Platform, and all further standard explanations will belong to that platform.

1.1.4 The layered architecture

As we already said, the Classic Platform distinguishes on the highest abstraction level between three software layers: Application, Runtime Environment and Basic Software which run on a Microcontroller. [11]

Aside from these layers there are also the AUTOSAR Libraries, which are a collection of state-less synchronous functions that can be called from any layer. There are 8 AUTOSAR Libraries, and most of them are for accomplishing complex mathematical calculi or for providing cryptographic algorithms (like the E2E Communication library, which we will cover in the next chapter).

1.1.5 The Basic Software

The Basic Software is further divided in four layers: Services (represented in blue in Figure 1.3), ECU Abstraction (in green), Microcontroller Abstraction (in red) and Complex Drivers. The Basic Software Layers are also divided into functional groups (also called "clusters"), based on the specific service they contribute to.

The Microcontroller Abstraction Layer (MCAL) is the lowest software layer of the Basic Software. It contains internal drivers, which are software modules with direct access to the microcontroller and internal peripherals, and its main task is to make higher software layers independent of the microcontroller (so the MCAL implementation itself is microcontroller-dependent). The main MCAL clusters are:

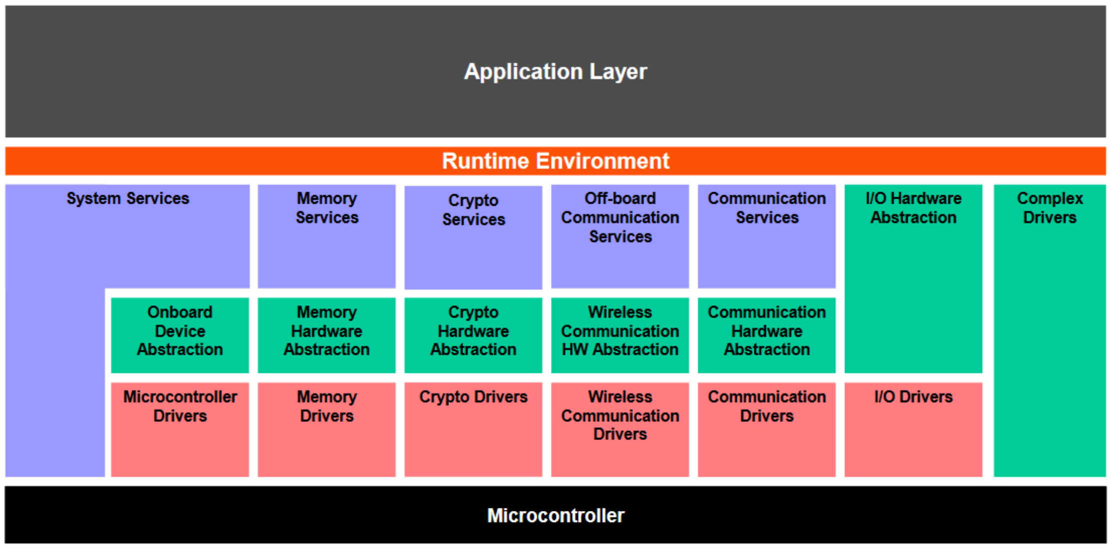


Figure 1.3: A view of the different AUTOSAR layers and BSW clusters

Communication Drivers (low level drivers for ECU onboard, e.g. SPI, and vehicle communication, e.g. CAN, but also for the Data Link Layer of the OSI stack), Memory Drivers (drivers for on-chip memory devices and memory mapped external memory devices) and Crypto Drivers (drivers for on-chip crypto devices, which we will later cover in detail).

The ECU Abstraction Layer (EAL), also called Hardware Abstraction Layer, interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an API for access to peripherals and devices regardless of their location (if internal or external to the microcontroller) and their connection to the microcontroller (port pins, type of interface): so it basically makes higher software layers independent of the ECU hardware layout. For example, the Memory Hardware Abstraction group abstracts the memory access from the specific memory location and so memories located in different places (e.g. an on-chip EEPROM and an external flash memory) can be accessed via the same mechanism.

The Services Layer (SL) is the highest layer of the BSW which also applies for its relevance for the application software. The Services Layer offers: operating system functionality, vehicle network communication and management services, memory services (NVRAM management), Diagnostic Services (including UDS communication, memory errors and fault treatment), ECU state management, mode management, logical and temporal program flow monitoring (Watchdog manager).

The BSW also contains the Complex Drivers (or Complex Device Drivers, CDD), where fall all the special purpose functionalities which are not already specified within the standard, and can span from the hardware to the RTE.

Based on the type of service provided, Basic Software modules (across all layers) can also be subdivided in the following groups (often referred to as “stacks”): Input/Output (I/O), Memory, Crypto, Communication, Off-board Communication, System. [11]

1.1.6 The Runtime Environment

The Runtime Environment (RTE) is the layer providing communication services to the application software (AUTOSAR Software Components and/or AUTOSAR Sensor/Actuator components). Above the RTE the software architecture style changes from “layered” to “component style”. The AUTOSAR Software Components communicate with other components (inter and/or intra ECU) and/or services via the RTE. The RTE provides the infrastructure services that enable communication to occur between AUTOSAR Software Components and acts as the mean by which AUTOSAR Software Components access basic software modules, including the OS and communication service. The RTE encompasses both the variable elements of the system infrastructure that arise from the different mappings of components to ECUs as well as standardized RTE services. In principle the RTE can be logically divided into two sub-parts: one that realizes the communication between software components and the other that realizes the scheduling of the software components.

The RTE provides different paradigms for the communication between Software Component instances, the main two are: sender-receiver (signal passing) and client-server (function invocation).

Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements that are sent by one component and received by one or more components; sender-receiver communication is one-way: any reply sent by the receiver is sent as a separate sender-receiver communication; it can be explicit (specific API calls are used to send/receive data) and implicit (send and/or receive operations are done automatically).

Client-server communication involves two entities, the client which is the requirer (or user) of a service and the server that provides the service: the client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary; the server, in the form of the RTE, waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client’s request; so, the direction of initiation is used to categorize whether a AUTOSAR software-component is a client or a server; client-server communication can be one-to-one or many-to-one. [12]

An important concept in communication between Software Components is data transformation, i.e. transforming (linearizing, serializing, encrypting. . .) data for making it ready to be communicated. Data transformation is performed by "transformers", which are part of the Basic Software and will be covered later in this chapter in detail, but transformers are coordinated and, if necessary, chained by the RTE.

1.1.7 The Application layer

In AUTOSAR, an application is modeled as a composition of interconnected Software Components (SW-C or SWC). An SW-C is an element that takes inputs from other components, elaborates them and sends outputs to other SW-Cs. A component has well-defined "ports", through which the component can interact with other components; a port always belongs to exactly one component and represents a point of interaction between a component and other components. A single component can implement both very simple but also very complex functionalities. It may have a small number of ports providing or requiring simple pieces of information, but can also have a large number of ports providing or requiring complex combinations of data and operations. AUTOSAR supports multiple instantiations of components. This means that there can be several instances of the same component in a vehicle system.

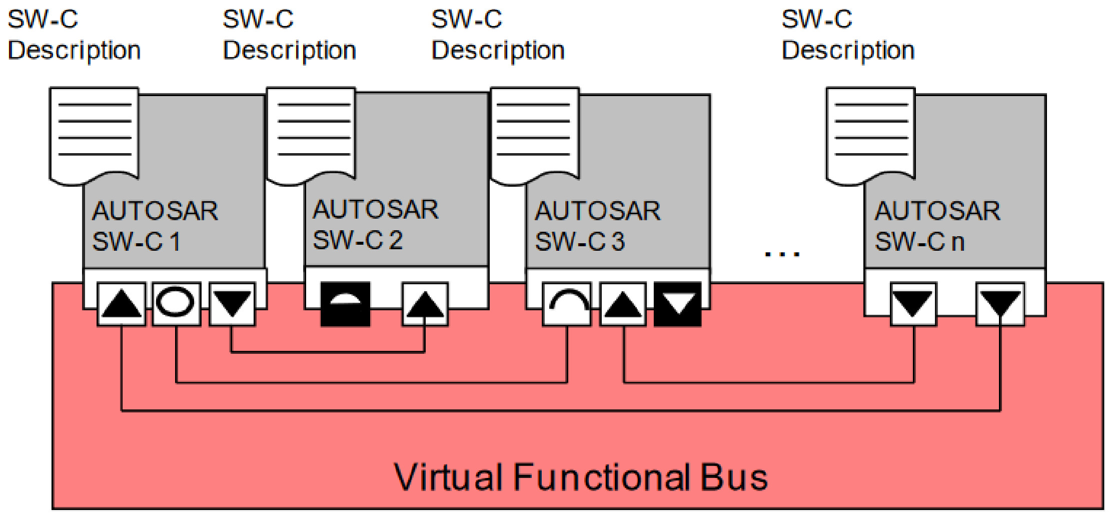


Figure 1.4: AUTOSAR components and VFB

The lower layer services used by SW-Cs, including the communication between them, is modeled at the Application layer as an abstract bus called “Virtual Functional Bus” or VFB so as to decouple the applications from the infrastructure. In practice, the VFB hides to the SW-C the underlying implementation of the RTE (or of multiple RTEs) and the BSW modules used by the component. The VFB specification needs to provide concepts for all infrastructure-services that are needed by a component implementing an automotive application; these include: communication to other components in the system, communication to sensors and actuators in the system, access to standardized services (such as reading to or writing from non-volatile ram), responding to mode-changes (such as changes in the power-status of the local ECU), interacting with calibration and measurement systems. [13]

1.1.8 Transformers

A transformer is a BSW module that takes data from the RTE, works on them and returns the output back to the RTE. It can both serialize/linearize data (transform them from a structured into a linear form) and transform them (modify or extend linear data, e.g by adding a checksum). Transformers are Service layer modules in the Communication Service cluster which provides communication services to the RTE. The transformers are executed by the RTE when the RTE needs the service which a transformer provides. Transformers can be stateless or stateful.

It is possible to connect a set of transformers together into a transformer chain. The RTE coordinates the execution of the transformer chain and calls the transformers of the chain exactly in the specified order. Using that mechanism, intra-ECU and inter-ECU communication is transformed if configured accordingly. The order of transformers in a chain represents the order on the sending side; the order on the receiving side is the inverse of the one on the sending side.

An example of inter-ECU data transformation is shown in Figure 1.5. In this example, an SWC sends complex data which are transformed using a transformer chain with two transformers (e.g. Transformer 1 serializes the data and Transformer 2 adds a checksum). On the receiver side, the same transformer chain is executed in reverse order with the respective retransformers. From the SWC’s point of view it is totally transparent for them which transformer are used or whether transformers are used at all. [14]

Transformers are divided into classes that include transformers which provide similar functionalities and must fulfill the same requirements. Currently 4 transformer classes are defined: Serializer, Safety, Security and Custom.

A “Serializer” transformer accepts complex data (either a Sender/Receiver data element or a Client/Server operation with its arguments) or no data from the RTE

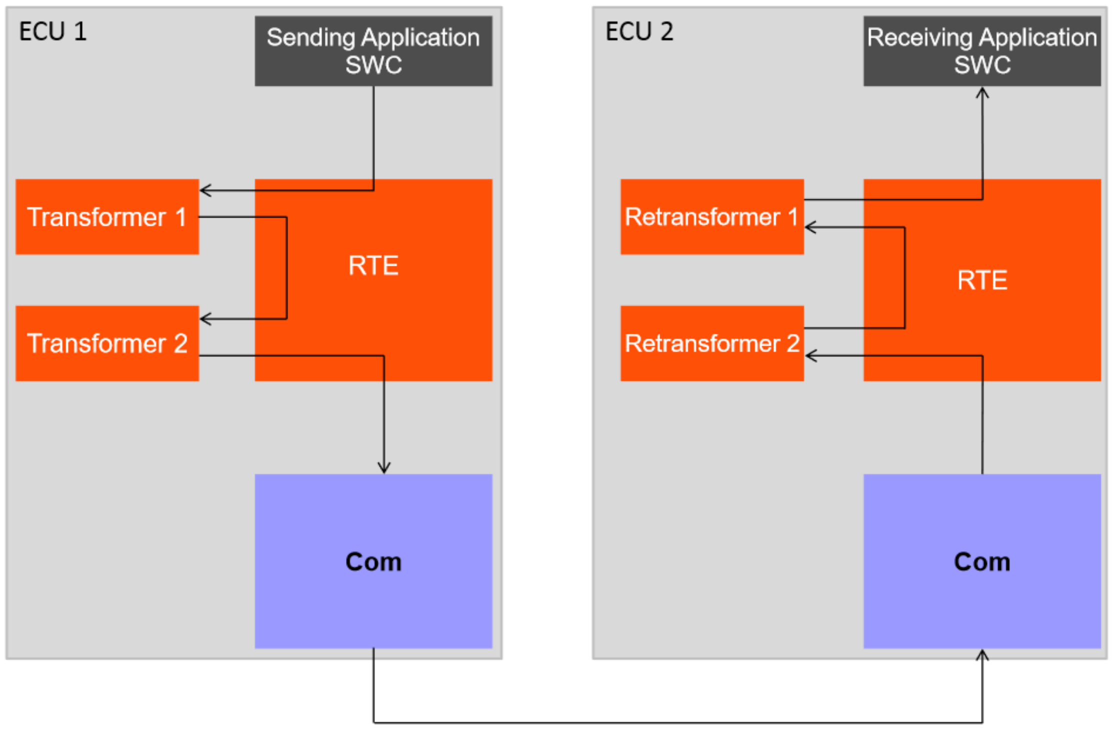


Figure 1.5: Transformer example for inter-ECU communication

and provides the resulting byte array as a signal (called ISignal) or part of a PDU (called IPdu), which can be then transmitted to the receiver by the COM stack; examples of Serializer transformers are the “COM Based transformer” and the “SOME/IP transformer”.

“Safety” transformers protect the communication against unintentional modifications that could change the order or the content of the data transmission.

A “Security” transformer to ensures the security of the bus communication by granting the authenticity, integrity and freshness of the transmitted data; an example of security transformer is the “E2E transformer” that makes use of the E2E Protection Library.

Custom transformers are not specified by AUTOSAR but can be specified by any party in the development workflow to implement a transformer which is not standardized, and are implemented as Complex Drivers. As a matter of fact, the core of the architecture designed and presented in this thesis is a custom transformer, and it will be covered in Chapter 3.

1.1.9 Transformers buffer handling

A transformer will usually work on the data and/or generate some protocol information which are stored in a header and/or footer of the output. Therefore it needs a place to write the result to: this place is called “buffer”. Transformers can work with two buffer handling modes: in-place buffer (the same memory space is used as input buffer and output buffer) and out-of-place buffer (two buffer are used, one for input and one for output).

The RTE allocates the buffers that are used by the transformers: it calculates the buffer size which is needed in worst case for the output [14].

For example, if a transformer with in-place buffering on the sending side is configured to add a header, the RTE is responsible for handing over a buffer which is large enough to store both the original data and the header: in this case the buffer grows between two transformers. To account for this, the RTE can have a buffer which stays the same size and is large enough to hold the output of the last transformer but only subsets of the buffer are handed over to the transformers depending on the buffer size needs of the specific transformers in the chain.

This can be achieved by pointers: a free space in front of the existing data to insert the header can be provided by the RTE by decreasing the pointer address which is handed over to the transformer. This adds a free space to the beginning of the buffer. [12]

In terms of security protection, buffer cleaning by the RTE is critical since uncleaned buffers could be read by unwanted entities (such as other custom transformers), especially when we’re dealing with transformer chains. So, the RTE must be sure to erase the content of each buffer between transformer operations.

1.2 Cybersecurity in AUTOSAR

Now we will talk about some of the means that AUTOSAR provides in terms of cybersecurity. Again, only the modules relevant to the developed work will be treated in detail.

1.2.1 Secure Hardware Extension

The Secure Hardware Extension (SHE), also referred to as Secure Hardware Extensions, is an on-chip extension to the microcontroller whose main purpose is to store and manage cryptographic keys, thus moving the control over the keys from the software domain into the hardware domain to protect them from software attacks. Other SHE’s goals include (but are not limited to): provide an authentic and trusted software environment, let the security only depend on the strength of the underlying algorithm and the confidentiality of the keys, allow for distributed key ownerships, keep the flexibility high and the costs low. In order to achieve these and other goals, the SHE not only performs key management but can also perform encryption/decryption, MAC generation and verification, compression, secure booting, random number generation and key generation.

The SHE is not required to be tamper-resistant, so it cannot replace highly secure solutions like Trusted Platform Modules (TPMs) or smart cards and can be used together with external Hardware Security Modules (HSMs). In the AUTOSAR layered architecture, being it a microcontroller extension, the SHE is placed in the Microcontroller layer. By default only the CPU can access the SHE, and SHE's only connection is the one with the CPU.

The SHE is divided in 5 blocks: a control unit called Control Logic (CL) that interfaces the CPU and handles all the other blocks; a RAM that stores the key and the state of the Pseudo Random Number Generator (PRNG); a ROM that contains a secret key inserted during chip fabrication and the SHE's identification number (UID); a flash memory that both contains pre-defined keys (e.g. the keys for secure booting) and also where other external keys can be stored; a block containing the AES implementation as well as correlated cryptographic algorithms (e.g. CMAC and Miyaguchi-Preneel).

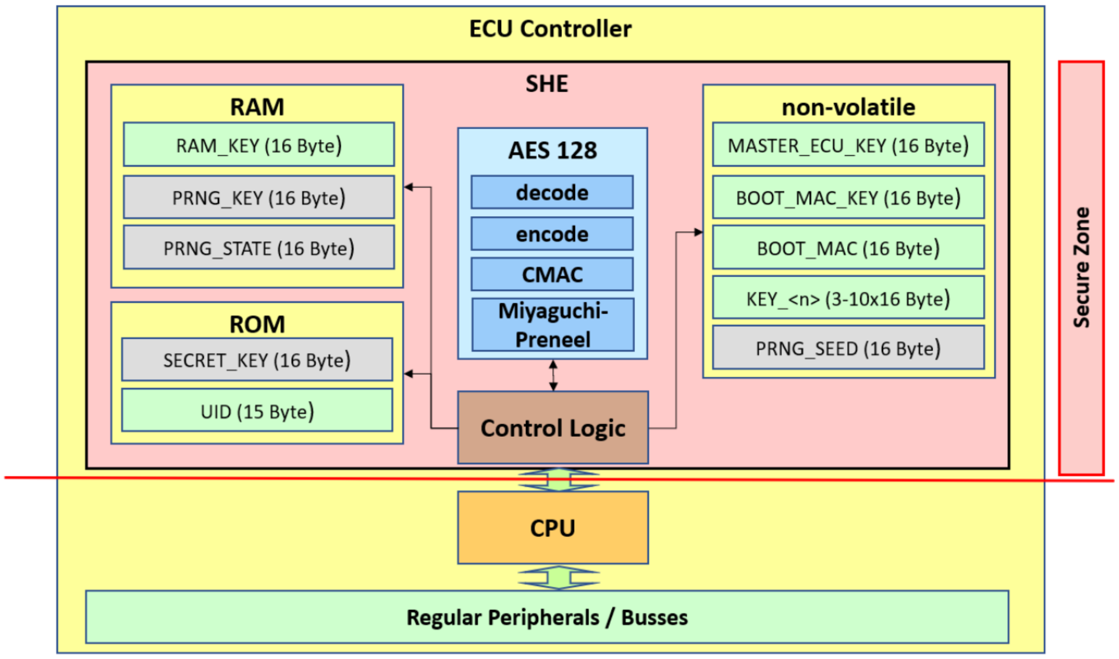


Figure 1.6: Detailed logical structure of the SHE

All cryptographic operations of SHE are processed by AES-128. For encryption and decryption SHE supports both Electronic Code Book (ECB) mode and Cipher Block Chaining (CBC) mode; note that ECB is generally considered unsecure, we will talk about that later. The SHE can also perform MAC generation and verification, implemented as a CMAC with AES-128.

SHE does not perform padding, it instead assumes that the dimension of the data its functions receive is a multiple of the block size of the algorithm. We will talk about cryptographic algorithms and padding later in this chapter.

Note that the SHE is not AUTOSAR's only resource for AES and MAC implementation and if a different implementation is needed (for example with a different key length or encryption mode) the user can resort to other modules and libraries (most notably dedicated Crypto Drivers and the E2E Protection library, both of which will be discussed later, as well as external Hardware Security Modules).

Another important function carried on by the SHE is secure booting, which is the process of checking the authenticity of the software on every boot cycle. This process is very important since historically attackers have often used modifying existing software or injecting new ones into the ECU to make it perform unwanted operations [15]. In the SHE, the secure boot process verifies an area of the memory against internal data of SHE and will lock parts of SHE if the verification fails. The verified part is identical to the first user instructions executed right after initialization of the microcontroller and is called "SHE Bootloader".

Note that secure booting does not directly protect the application software but can prevent a malicious application from using certain keys: to protect the software as well, a dependency between the software and the keys has to be generated, e.g. by encrypting parts of the software.

The SHE provides two different ways of performing the secure boot, "Measurement before application start-up" and "Measurement during application startup": in the former, the complete secure boot process is performed before the control over the microcontroller is handed over to the application (secure booting must always be finished before the CPU starts the application code); in the latter, which only has to be implemented on microcontrollers that support direct memory access, the secure boot process is only started before the control over the microcontroller is handed over to the application, but the actual measurement of the code is executed in parallel to the application start-up. [16]

1.2.2 Crypto stack

The Crypto stack is a set of functional groups, spanning across all three layers of the Basic Software, that are devoted to provide interfaces to cryptographic services such as symmetric encryption, hash computation, verification of asymmetrical

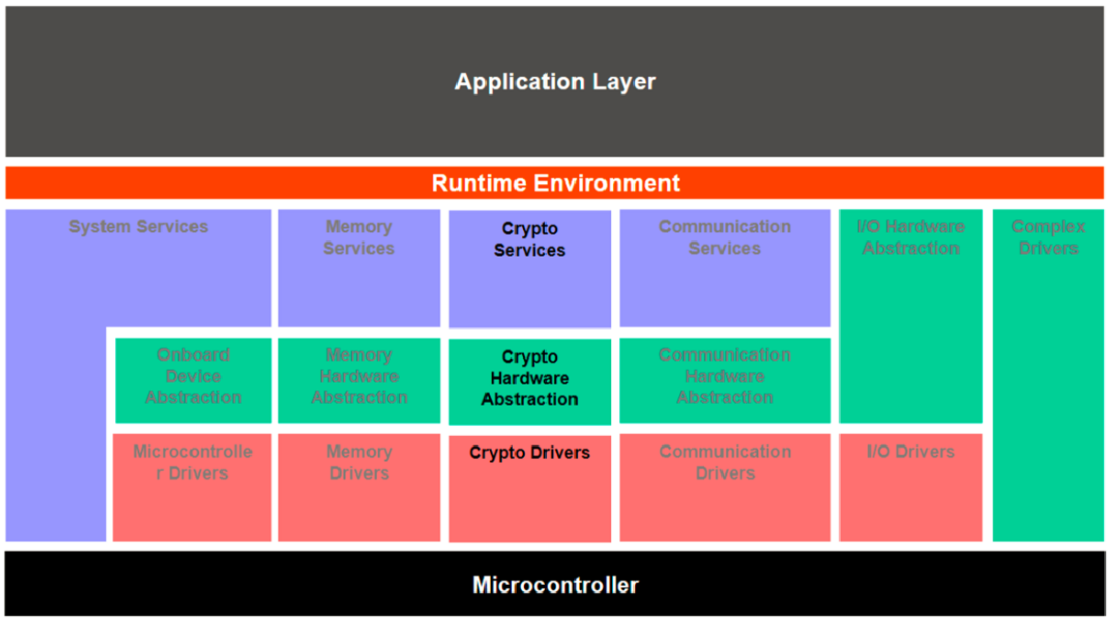


Figure 1.7: AUTOSAR layered view of the Crypto stack

signatures etc. The three functional groups that constitute the Crypto stack are, from bottom to top: Crypto Drivers, Crypto Hardware Abstraction and Crypto Services. [17]

The Crypto Drivers group (whose modules are called “CRYPTO”) is located in the Microcontroller Abstraction Layer. It holds the actual implementations of the cryptographic algorithms, both if they are provided via software (e.g. by a cryptographic library) and via hardware (e.g. by the SHE or an external HSM). Each ECU can have multiple CRYPTO modules (e.g. for dividing different sources of implementation and/or implementations from different vendors) and each module can have different Crypto Driver Objects (CDO) that provides a fixed set of crypto primitives; a crypto primitive is an instance of a configured cryptographic algorithm, and a Crypto Driver Object can only perform one crypto primitive at the time. The concept of several CRYPTO modules and several Crypto Driver Objects allows different and concurrent implementations of the same cryptographic services. Variants of CRYPTO modules with different optimization objectives may exist. For instance, the same hash algorithm might be implemented in two different CRYPTO modules, one by a faster (more expensive) hardware solution and the other by a slower (cheaper) software solution.

The Crypto Interface (CRYIF) is the main module of the Crypto Hardware Abstraction group, which is the group suited on the ECU Abstraction Layer. It provides a generic interface for the Service layer to the available CRYPTO modules and makes the access independent of the underlying Crypto Drivers. It receives requests from the upper layer, then maps and forwards them to the appropriate cryptographic operation in the particular CRYPTO. The CRYIF can operate several CRYPTO modules: since, for example, there could be different implementation for the same cryptographic primitive, the CRYIF provides a generic interface so that the access from the Crypto Service Manager does not need to distinguish between the actual CRYPTO implementations. External cryptographic drivers (that use different MCAL drivers) also belong to the Crypto Hardware Abstraction.

The most important module of the Crypto Services group is the Crypto Service Manager (CSM): it provides an abstraction layer which offers standardized access to cryptographic services for applications via the RTE port mechanism. Other BSW modules and CDD can use C-API calls provided by the CSM to use the cryptographic services. The CSM controls the concurrent access of one or more clients to one or more synchronous or asynchronous cryptographic services. Another important Crypto Services module is the Key Manager, which interacts with the key provisioning master (either in NVM or Crypto Driver) and manages the storage and verification of certificate chains [11].

1.2.3 The E2E Library and the E2E Protocol

The SW-C End-to-End Communication Protection Library, often just called "E2E Library", is a library that provides mechanisms for protecting safety-related communications against the effects of faults within the communication link (see Figure 1.9). Examples for such faults are random HW faults (e.g. corrupt registers of a CAN transceiver), interference (e.g. due to Electromagnetic compatibility), and systematic faults within the software implementing the VFB communication (e.g. RTE, IOC, COM and network stacks). By using E2E communication protection mechanisms, the faults in the communication link can be detected and handled at runtime.

The protection mechanisms are implemented in the E2E Library, which performs the following:

1. it protects the safety-related data elements to be sent over the RTE by attaching control data (e.g. a CRC),
2. it verifies the safety-related data elements received from the RTE using this control data, and
3. it indicates that received safety-related data elements faulty, which then has to be handled by the receiver SW-C.

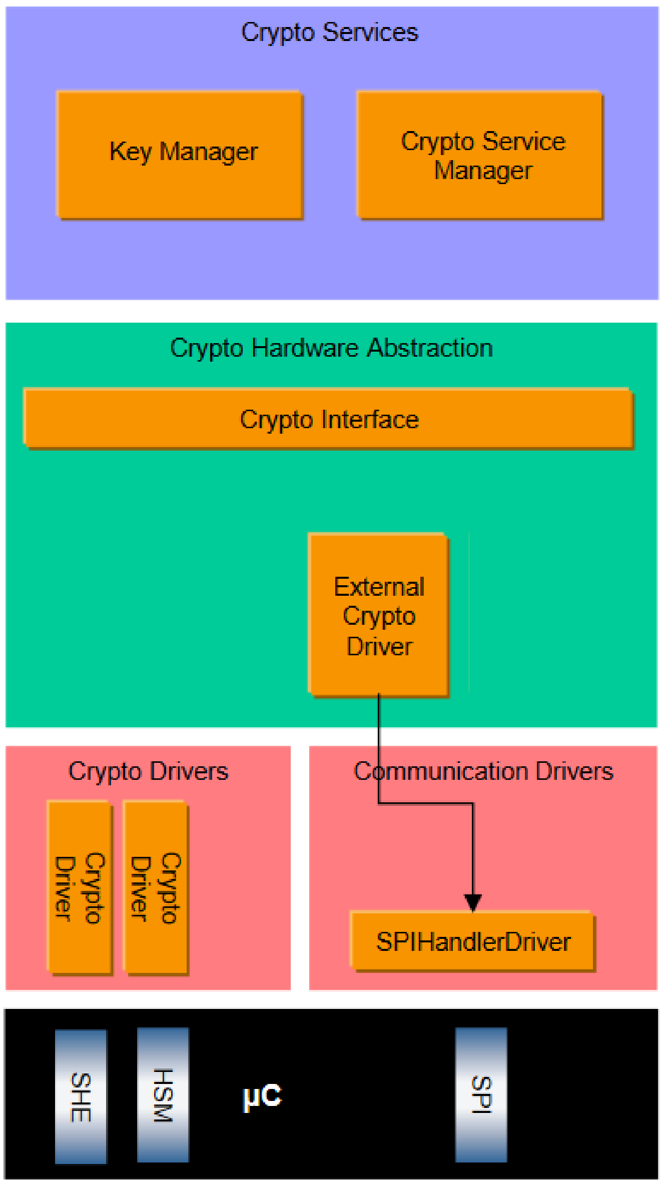
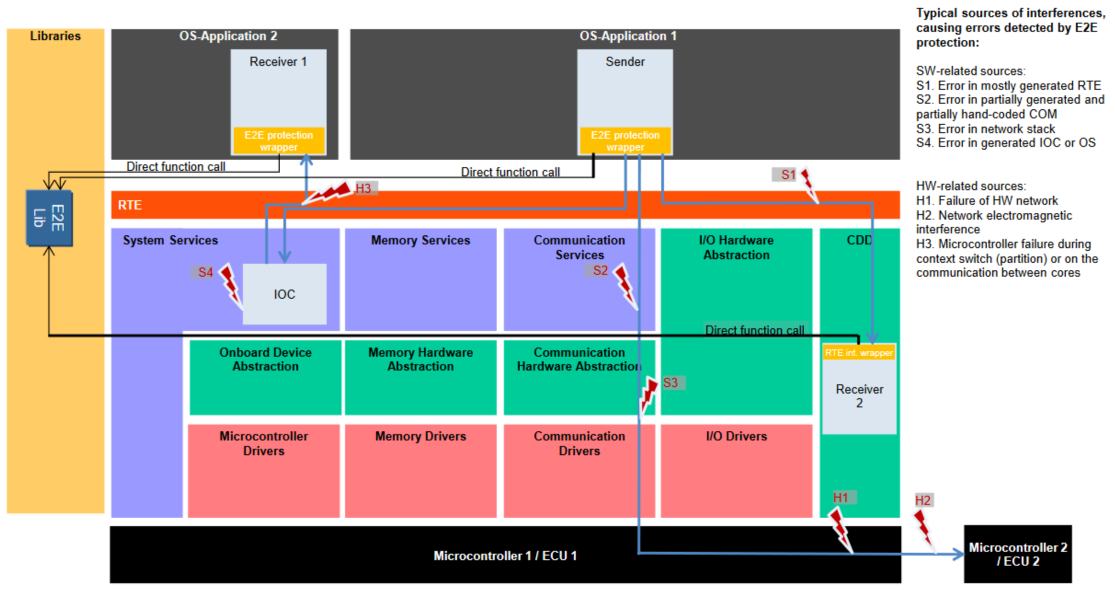


Figure 1.8: In-depth view of the Crypto stack

To provide the appropriate solution addressing flexibility and standardization, AUTOSAR specifies a set of flexible E2E profiles that implement an appropriate combination of E2E protection mechanisms. Each specified E2E profile has a fixed behavior, but it has some configuration options by function parameters (e.g. the location of CRC in relation to the data, which are to be protected).

The E2E library is invoked from the E2E Transformer, which we will see later,



Typical sources of interferences, causing errors detected by E2E protection:

SW-related sources:
S1. Error in mostly generated RTE
S2. Error in partially generated and partially hand-coded COM
S3. Error in network stack
S4. Error in generated IOC or OS

HW-related sources:
H1. Failure of HW network
H2. Network electromagnetic interference
H3. Microcontroller failure during context switch (partition) or on the communication between cores

Figure 1.9: Example of faults mitigated by E2E protection

but also from the "E2E Protection Wrapper" and the "COM E2E Callout" modules. Regardless of who calls the E2E library, the E2E protection is for data elements, so E2E protection is performed on the serialized representation of data elements; a data element (and the corresponding signal group) is either completely E2E-protected, or it is not protected: it is not possible to protect a part of it.

An appropriate usage of the E2E Library alone is not sufficient to achieve a safe E2E communication according to high security requirements: solely the user is responsible to demonstrate that the selected profile provides sufficient error detection capabilities for the considered network (e.g. by evaluation hardware failure rates, bit error rates, number of nodes in the network, repetition rate of messages and the usage of a gateway). [18]

The E2E Protocol defines how E2E protected communications must be carried out. It treats the callers of the E2E Library as a black-box entity called "E2E Supervision", that can be called both from a communication middleware (e.g. the RTE) and in a non-standardized way from other software (e.g. non-volatile memory managers, local IPCs, or intra-ECU bus stacks).

The protocol also defines the fixed set of mechanisms and configuration options of the packet layout for each one of the E2E profiles: among other configurations, each profile defines how the CRC must be calculated. There are many different CRC calculation routines to choose from (with different CRC sizes), and for all routines three calculation methods are possible: table based calculation (fast execution but larger code size), runtime calculation (slower execution but smaller code size) and hardware supported CRC calculation (device specific, fast execution and less CPU overhead) [19].

1.2.4 The E2E transformer

The E2E transformer is a “Safety” class transformer responsible for the invocation of the E2E Library based on the configuration of specific data element (I-signal) to be transmitted. The E2E transformer instantiates the E2E configuration and E2E state data structures, based on its configuration. All E2E profiles may be used to protect data. The E2E transformer encapsulates the complexity of configuring and handling of the E2E Library and it offers a standard Transformer interface. Thanks to this, the caller of the E2E transformer (which is the RTE) does not need to know the E2E Library internals.

On the sender side, the E2E transformer E2E-protects the data. On the receiver side, the E2E transformer E2E-checks the data, providing the result of the E2E-checks through the RTE to the SW-C. If a receiving SW-C does not read the transformer return codes, it is fully transparent to the communicating SW-Cs whether the data are E2E protected on the bus or not [20].

1.3 MQTT

MQTT is a broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. These characteristics make it ideal for use in constrained environments, for example where the network is expensive, has low bandwidth or is unreliable or on an embedded device with limited processor or memory resources [21]. Example of such environments are communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium [22].

The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Its features include:

- The publish/subscribe message pattern to provide one-to-many message distribution and decoupling of applications
- A messaging transport that is agnostic to the content of the payload
- A small transport overhead (the fixed-length header is just 2 bytes), and protocol exchanges minimised to reduce network traffic
- A mechanism to notify interested parties to an abnormal disconnection of a client using the Last Will and Testament feature

It also support three Quality of Service levels for message delivery:

- "At most once" (QoS 0), where messages are delivered according to the best efforts of the underlying TCP/IP network. Message loss or duplication can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
- "At least once" (QoS 1), where messages are assured to arrive but duplicates may occur.
- "Exactly once" (QoS 2), where messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

MQTT was designed by Andy Stanford-Clark (IBM) and Arlen Nipper in 1999 for connecting Oil Pipeline telemetry systems over satellite. Although it started as a proprietary protocol it was released Royalty free in 2010 and became an OASIS standard in 2014 [23].

The acronym initially stood for “MQ Telemetry Transport”, MQ referring to the “MQ Series” (a product IBM developed to support telemetry transport). Nowadays “MQTT” is no longer considered an acronym, it simply is the name of the protocol [24].

Figure 1.10: MQTT logo

1.3.1 The publish/subscribe pattern

The publish/subscribe pattern (also known as pub/sub) provides an alternative to a traditional client-server architecture. In the client-server model, a client communicates directly with an endpoint. The pub/sub model decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers). The publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. The connection between them is handled by a third component: the broker.

The MQTT connection is always between one client and the broker. Clients never connect to each other directly. To initiate a connection, the client sends a CONNECT message to the broker. The broker responds with a CONNACK (Acknowledgement) message and a status code. Once the connection is established, the broker keeps it open until the client sends a disconnect command or the connection breaks [24]. A client and a broker could also connect on top of a TLS connection.

The broker is responsible for dispatching messages between the publisher and the rightful subscribers. Each MQTT message includes a topic. A client publishes a message to a specific topic (by sending the message and its topic to the broker) and MQTT clients subscribe to the topics they want to receive. The MQTT broker uses the topics and the subscriber list to dispatch messages to appropriate clients [25].

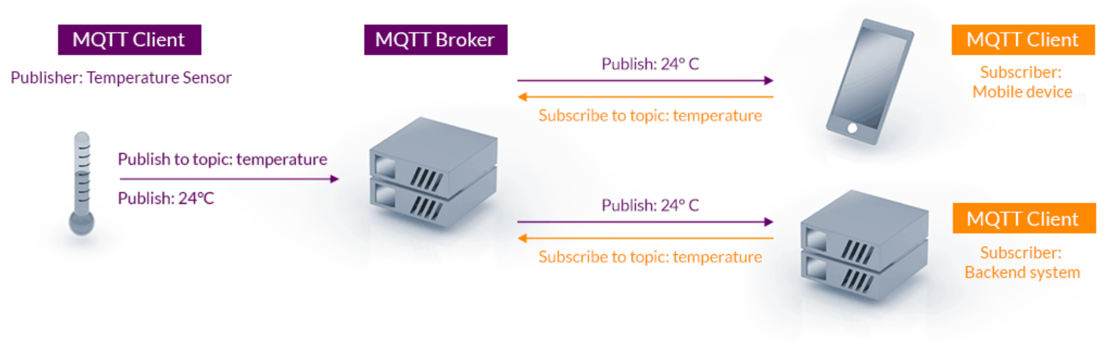

MQTT Client
Publisher: Temperature Sensor
MQTT Broker
Publish to topic: temperature
Publish: 24°C
Publish: 24° C
Subscribe to topic: temperature
MQTT Client
Subscriber: Mobile device
Publish: 24° C
Subscribe to topic: temperature
MQTT Client
Subscriber: Backend system

Figure 1.11: An example of MQTT usage

Let's follow the example in Figure 1.11 [26]: we have 3 clients and 1 broker. Two of the clients (the mobile and the backend system) are subscribers subscribed to the topic "temperature". The third client (the temperature sensor) publishes the message "24°C" on the topic "temperature". The broker then sends this message to the two subscribers.

1.3.2 MQTT packet format

MQTT messages contain a mandatory fixed-length header (2 bytes) and an optional message-specific variable length header and message payload [27].

The fixed header consists of the Control Field and the variable Remaining Length field: the Control Field contains 4 bits for the packet type (e.g. CONNECT or PUBLISH) and 4 bits for the control flags, i.e. 1 bit for the duplicate flag (used when re-publishing a message with QOS or 1 or 2), 2 bits for the Quality of Service level and a RETAIN bit (used to instruct the server to retain the last received PUBLISH message and deliver it as a first message to new subscriptions); the Remaining Length field is of variable length between 1 and 4 bytes, and contains the number of bytes following the length field (including the variable length header and payload) [28].

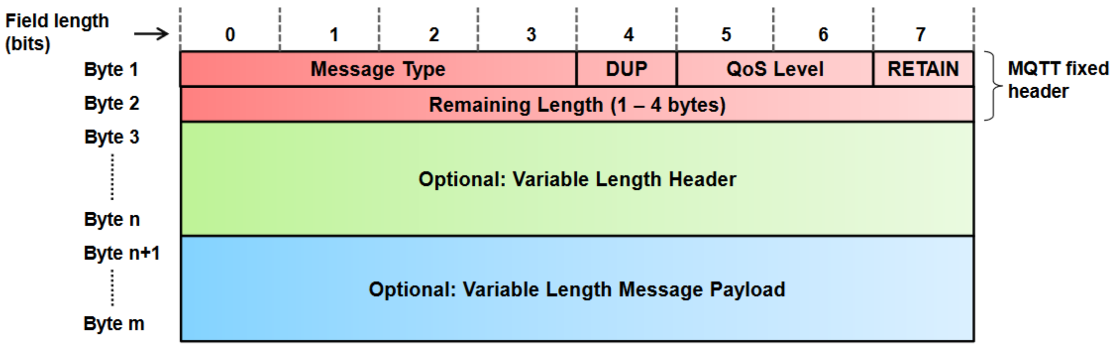


Figure 1.12: Format of an MQTT packet

The minimum packet size is just 2 bytes with a single byte control field and a single byte packet length field (e.g. the disconnect message is only 2 bytes). The maximum packet size is 256MB.

As mentioned previously the variable length header field is not always present in an MQTT message. Certain MQTT message types or commands require the use of this field to carry additional control information, for example the name and version of the protocol or the keep (connection) alive time. The topic of a PUBLISH message is also specified in the variable length header.

The payload can contain different information based on the type of message, for example: in a CONNECT message it can contain the ID of the client and optionally its username and password; in a PUBLISH message it contains the message to publish; in a SUBSCRIBE message it contains the topic to subscribe to, together with its length.

1.3.3 MQTT-SN

Due to the large number of sensors in IOT Sensor networks, these sensors are mainly wireless, are operated by a low power battery, have very limited processing power and storage, can send messages with a limited payload size and are not always on.

MQTT-SN (MQTT for Sensor networks) was designed specifically to work on wireless networks and, as far as possible, to work in the same way as MQTT. It uses the same publish/subscribe model and can be considered as a version of MQTT.

The main differences involve:

- reduced size of the message payload
- removing the need for a permanent connection by using UDP as the transport protocol
- pre-defined topics with topic IDs used in place of topic names
- off-line keep alive procedure for sleeping clients [29]

1.3.4 MQTT implementations

There are more than 100 different MQTT implementations [30]. The main differences between the implementations are what platform and/or operating system they run on, in which programming language they are available and if they implement the MQTT client and/or the MQTT broker. We will focus on the three implementations that were used while developing and testing the designed: Paho, Mosquitto and mqttools.

Paho is Eclipse's implementation of the MQTT client. It is available in a large number of programming languages and platforms, most notably it is available for embedded C/C++. For embedded systems it also provides the implementation of the MQTT-SN client. [31]

Mosquitto is also developed by Eclipse and provides the implementation of the MQTT broker as well as a library for implementing MQTT clients. It is written in C and it is available for the main desktop operating systems (Windows, MacOS and Linux) as well as for the Raspberry Pi. [32]

MQTT Tools (project name "mqttools") is a Python 3 library developed by Erik Moqvist. It provides both the MQTT client and the MQTT broker, and it can run on anything that supports a Python interpreter [33].

Table 1.1 summarizes the main differences of these three implementations, highlighting: their developer, their programming languages, whether they support embedded systems, whether they implement the client and/or the broker, whether they implement MQTT-SN and whether they support TLS connection [34].

	Paho	Mosquitto	mqttools
Developer	Eclipse	Eclipse	Erik Moqvist
Languages	C, C++, Java, Python et al.	C	Python
Embedded	Yes	No	No
Client/Broker	Client	Broker	Both
MQTT-SN	Yes	No	No
TLS	Yes	Yes	Yes

Table 1.1: MQTT implementations comparison

1.4 Cryptographic algorithms

We will now have a look at the different cryptographic processes and algorithms that we will deal with.

1.4.1 Encryption

Encryption is the cryptographic transformation of data (called "plaintext") into a form (called "ciphertext") that conceals the data's original meaning to prevent it from being known or used. If the transformation is reversible, the corresponding reversal process is called "decryption," which is a transformation that restores encrypted data to its original state [35].

There are two kinds of encryption: symmetric encryption and asymmetric encryption. In symmetric encryption (also known as secret key encryption) there is only one key, and all communicating parties use the same (secret) key for both encryption and decryption. In asymmetric (or public key) encryption, there are two keys: one key is used for encryption, and a different key is used for decryption; one of the two keys is kept private (hence the "private key" name), while the other is shared publicly for anyone to use (hence the "public key" name). [36]

Symmetric encryption is primarily used to achieve confidentiality, i.e. to ensure that no one can read the message except the intended receiver, while asymmetric encryption is primarily used for authentication (the process of proving one's identity), non-repudiation (proving that the sender really sent this message) and key exchange (the method by which crypto keys are shared between sender and receiver) [37].

We will focus on symmetric encryption since it is the only kind used in the presented proof of concept.

1.4.2 Symmetric Encryption

Symmetric encryption algorithms are divided into stream ciphers and block ciphers.

Stream ciphers encrypt bits individually. This is achieved by adding a bit from a key stream to a plaintext bit. There are synchronous stream ciphers where the key stream depends only on the key, and asynchronous ones where the key stream also depends on the ciphertext [38]. An example of stream cipher is RC4 [39].

Block ciphers encrypt an entire block of plaintext bits at a time with the same key. This means that the encryption of any plaintext bit in a given block depends at least on every other plaintext bit in the same block [38]. Block ciphers can have different block sizes (mainly 64 and 128 bits) and different key length. Examples of block ciphers are DES (64 bits block and 64 bits key, of which 56 effective [40]) and AES.

1.4.3 Block ciphers: AES

After DES was proven to be flawed (mainly due to its small key length), in 1997 the National Institute of Standards and Technology of the United States (NIST) started the research for a new block cipher standard that would have taken the name of Advanced Encryption Standard (AES) [41]. After 3 years, in October 2000, the quest was over: the chosen algorithm was Rijndael, named after the two Belgian scientists that created it (Vincent Rijmen and Joan Daemen) [42].

While the original Rijndael specified that the block length and the key length can be independently specified to any multiple of 32 bits from 128 bits to 256 bits, AES fixed the size of the block to 128 bits and the key length to 128, 192 or 256 bits [43].

1.4.4 Block ciphers: modes of operation

Block ciphers can work with different modes of operation, i.e. different ways to deal with plaintexts that are bigger than the size of a block [44]. There are many different modes of operation [45][46], but for our purposes we will focus on three of them: ECB, CBC and CTR.

Electronic Codebook (ECB) is the simplest mode of operation: it consists in dividing the plaintext in blocks of the cipher's block size and encrypting them individually. While this is simple and fast (since the encryption of the single block can be parallelized), it is considered not secure: the main problem is that in this way equal plaintext blocks produce equal ciphertext blocks, which leads to very easy pattern recognition. This can be well seen in Figure 1.13 where, from left to right, we have the original image, the same image encrypted with ECB and the image encrypted with another mode of operation [47].

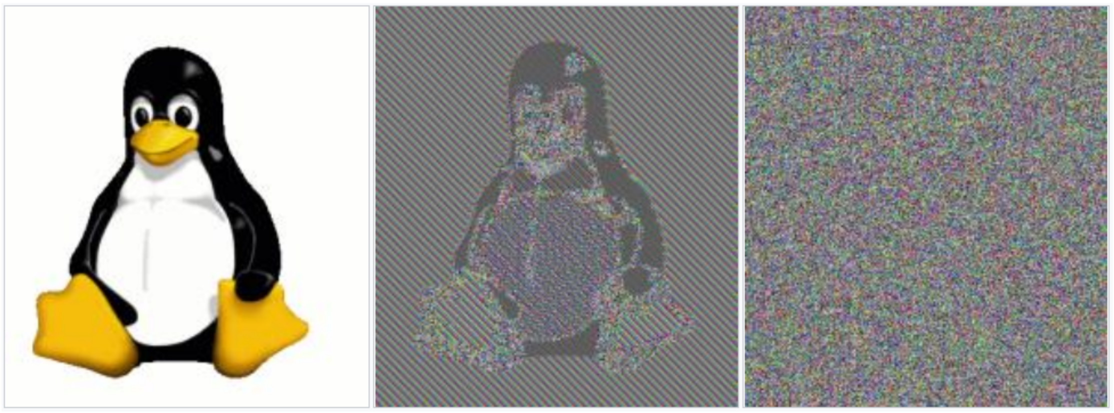


Figure 1.13: Pattern recognition problem of ECB

Cipher Block Chaining (CBC) is the most commonly used mode of operation. In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. The first block is XORed with a unique initialization vector (IV). The main drawback of CBC is that it is not parallelizable, since each block has to wait for the encryption of the previous one.

Counter (CTR) mode turns a block cipher into a stream cipher. Starting from a unique nonce, it generates the next keystream block by concatenating it to successive values of a "counter" and then encrypting it. The plaintext is then XORed with the result of this encryption. The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular. CTR mode is parallelizable, and since the counter changes from block to block we don’t have the same data pattern problem of ECB. [48]

When the size of the plaintext is not a multiple of the cipher’s block sizes, some modes of operation (like ECB and CBC) require adding data at the end of the last block to make it reach the appropriate block size: this is called “padding”. There are different padding schemes, i.e. different ways of telling that this additional data is just padding and is not relevant to the plaintext, for example PKCS#7: each added byte is the number of total bytes to add, e.g. if there must be added 3 bytes the padding would be 0x030303 [49]. It is worth mentioning that for ECB and CBC an alternative to padding is Ciphertext Stealing (CTS) [50].

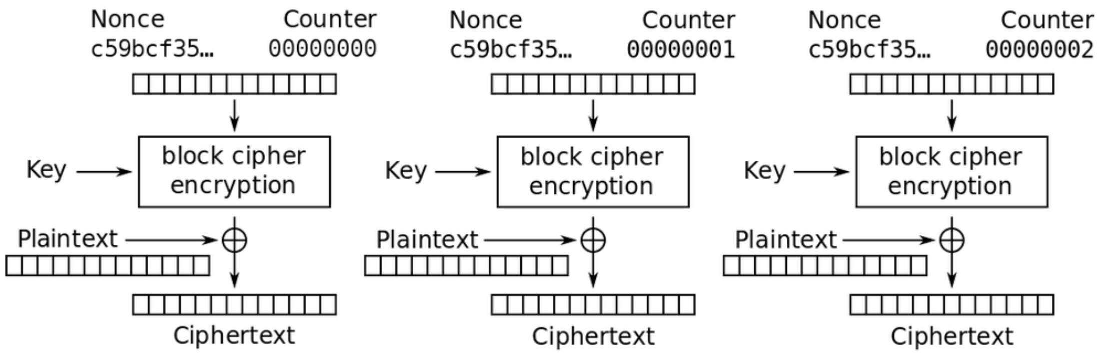


Figure 1.14: Counter (CTR) mode encryption

1.4.5 Message Authentication

In information security, message authentication or data origin authentication is the property that a message has not been modified while in transit (data integrity) and that the receiving party can verify the source of the message [51]. Message authentication can be achieved by adding some information to the message that the recipient can check to verify that the message is authentic; these information can be computed with asymmetric (digital signatures) or symmetric (MACs) means [52]. We will focus on the latter.

1.4.6 Message Authentication Code

A Message Authentication Code (MAC) is created by computing a checksum with the message content and the shared secret. A MAC can be verified only by a party that has both the shared secret and the original message content that was used to create the MAC [52]. MACs can be generated either by means of a hash function (like in the case of HMAC [53], where the key is intermingled with the message before hashing) or via a block cipher (e.g. CMAC and CBC-MAC).

1.4.7 Authenticated Encryption

There are many ways to combine authentication and encryption for achieving both message authentication and confidentiality for the same message, the ones of interest for us are: Encrypt-and-MAC (E&M), MAC-then-Encrypt (MtE), Encrypt-then-MAC (EtM) and Galois/Counter Mode (GCM) [54][55][56].

In Encrypt-and-MAC the ciphertext and the MAC are computed separately and then sent together. This is the approach used by the SSH protocol and it is not proven to be strongly secure without some minor modifications.

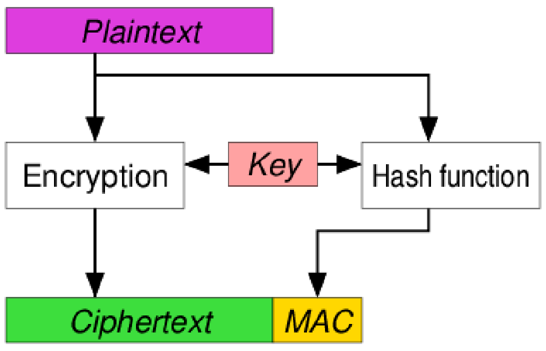


Figure 1.15: Encrypt-and-MAC

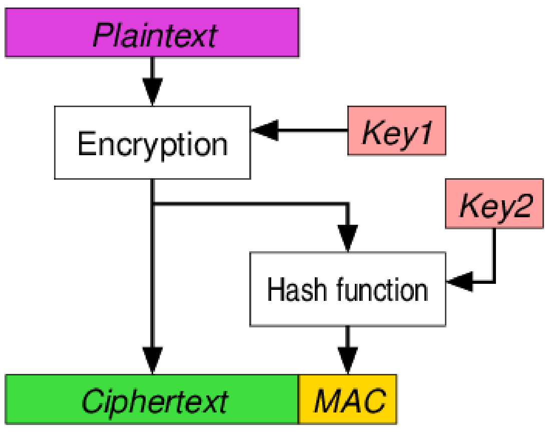


Figure 1.16: Encrypt-then-MAC

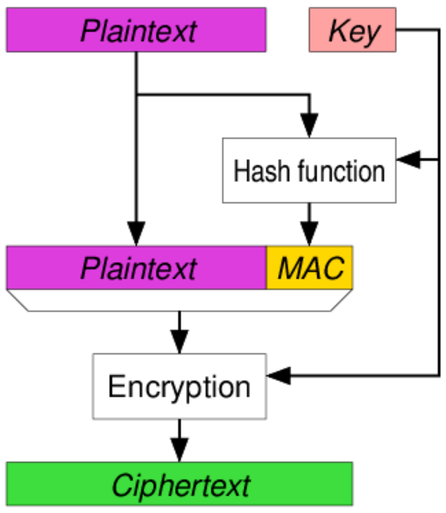


Figure 1.17: MAC-then-Encrypt

In MAC-then-Encrypt the MAC is produced based on the plaintext, then the plaintext and MAC are together encrypted to produce a ciphertext based on both. It is used by SSL/TLS.

In Encrypt-then-Mac the plaintext is first encrypted, then a MAC is produced based on the resulting ciphertext; the ciphertext and its MAC are then sent together. Used in IPsec, this is one of the most secure modes if the MAC algorithm is strong enough.

Lastly, Galois/Counter Mode is a joint algorithm that combines the Counter block cipher encryption mode with the Galois field multiplication for authentication. It is very secure and the computation can be parallelized, making it also very fast. This is also used in TLS since 1.2 [57].

Chapter 2

Securing in-vehicle communications

The architecture that will be presented in the next chapter regards the communication between cars and external entities, but for the whole architecture to be completely secure we need to start from securing the communication between components within the car: if that communication is not secure someone could manipulate the transmitted data at the source. That is why we need to be sure that the received data was sent by the rightful sender and has not been modified by anyone: in cybersecurity terms, we need data authentication and integrity.

In order to explore how to make such a communication secure, during my internship in Brain Technologies I took part in a project together with two other interns: Domenico Alessi and Muhammad Ibrahim. The goal of the project was to understand and evaluate at different layers how, in the 4.4 version of AUTOSAR, it is possible to develop software components that use signals sent from different ECUs in a secure way.

2.1 Design of the Application layer

The application of this project regards the design of the windshield wiper vehicle function of a car: based on some input data (e.g. the wiper speed setting chosen by the driver), the application decides if the wiper should stay put or should move with a certain velocity. All the work related to the application layer of this project was carried out by Domenico Junior Alessi [58].

2.1.1 Paradigms and tools used

The development of the application layer was realized with Model-based design approach (MBD): a design paradigm that overcomes the shortcomings of traditional development processes.

In a traditional development process, usually a system engineer defines the overall system specification and presents it as a design document to software engineers, who will have the task of implementing those ideas into a fully working solution. However, the main problem with this approach is the fact that in most cases the ideas presented by the system engineer via the specification document may widely differ from the implemented software. Even the most detailed and diligently prepared type of documentation may not always guarantee that the design document generated by the system engineers would be fully understood and accurately understood and interpreted correctly by the implementing software programmers.

With Model-based design, complicated systems can be created by using mathematical models representing system components and their interactions with their surrounding environment. System engineers can design and simulate the model using a model-based design tool and present the ideas behind the solution as a working model of the system. This model can then be used as the input to an automatic code generation tool [59].

The most important tool used in designing the application layer is Simulink: a platform for Model-Based Design that supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems [60].

To help modeling AUTOSAR components, Simulink was used together with the “AUTOSAR Blockset”, which provides apps and blocks for developing AUTOSAR Classic and Adaptive software using Simulink models and can map Simulink models to AUTOSAR software components. AUTOSAR Blockset also provides blocks and constructs for AUTOSAR library routines and Basic Software (BSW) services and can simulate the BSW services together with the application software model [61].

2.1.2 The model itself

The application consists of many sensors, a controller and an actuator, all in form of software components. The sensors retrieve some data from the ECU Abstraction Layer and send them to the controller. The controller elaborates those signals and, following the behavior of a stateflow chart, outputs a command to the actuator; this command tells the actuator if the wiper should stay put, if it should move at a

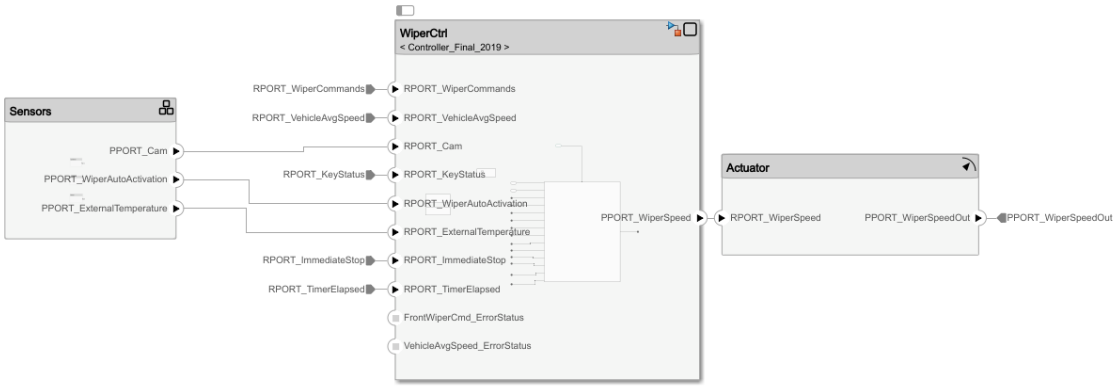


Figure 2.1: The model of the application in Simulink

low speed or if it should move at high speed. The actuator receives that command and sends it to the motor of the wiper through the Basic Software.

Some examples of signals received by the controller are: the position of the wiper (sent by the cam sensor), the external temperature, the speed of the vehicle, and the wiper lever setting chosen by the driver (stop, speed 1, speed 2 or automatic).
All those signals are sent to sender/receiver ports on the controller, making it possible to treat them as triggers: this way the controller can be set to work only when a signal is sent to it, instead of having to poll the bus.

The controller has two runnables. The first one, executed at initialization, reads in memory two data: the temperature under which the wiper cannot work and the time it must wait after the vehicle ignition to start elaborating input signals.
The second one, triggered when receiving a signal, is the main runnable and implements the control algorithm: based on the received signals, it changes the active state and eventually changes the value of the output. For example: if it receives that the wiper should not move anymore and the wiper position is not the “parking position”, it makes the wiper go back to its initial position before setting its velocity equal to 0. [58]

2.2 The communication

The communications between components in this project are mainly carried out using CAN. In the AUTOSAR layered architecture, the modules that exploit and interface the CAN bus and protocol are part of the "COM Stack", an aggregate of clusters devoted to communications. The integration and utilization of the COM Stack within the project is the work of Muhammad Ibrahim [62].

2.2.1 The CAN protocol

Controller Area Network (CAN) is a serial communication protocol which supports distributed real-time control and multiplexing, and it is used in vehicles and other control applications [63]. It was developed by Bosch in 1983 to be used in cars to exchange information between different electronic components; now it is used in nearly all application fields: industrial, medical, white goods and so on [64].

From a physical point of view, in a CAN bus all the communicating devices are connected to the same two wires, labeled CAN-High and CAN-Low. All the devices must use the bus at the same speed. In a normal situation, the two wires carry a two-level signal, perfectly specular, and whenever one is high the other one is low [65].

The CAN protocol is a message-based protocol, whose messages are very small [66], and all the transmitted messages are broadcasted to every connected node [67]. The maximum transmission rate supported by CAN is 1 Mbit/s [65].

Because of the bandwidth requirements of the automotive industry, the CAN data link layer protocol needed to be improved. In 2011, Bosch started the CAN-FD (flexible data-rate) development in close cooperation with carmakers and other CAN experts. The improved protocol overcomes CAN limits: data can be transmitted at a faster rate than with 1 Mbit/s and the payload (data field) is now up to 64 bytes long and not limited to 8 bytes anymore. In general, the idea is simple: when just one node is transmitting, the bit-rate can be increased, because no nodes need to be synchronized. [68]

2.2.2 The COM Stack

The AUTOSAR COM Stack comprehends functional groups from all the Basic Software Layers.

At the MCAL layer we have different COM drivers for every different communication interface that the microcontroller supports, whether they're for onboard (e.g. SPI) or vehicle (CAN, LIN) communication.

At the ECU Abstraction layer we have a group of modules which abstracts from the location of communication controllers and the ECU hardware layout: for all communication systems a specific Communication Hardware Abstraction is required.

Lastly, at the Service layer we have the Communication Services modules, whose purpose is to provide a uniform interface to the vehicle network for communication, network management and diagnostic communication. The three existing transformers (E2E, SOME/IP and COM based), as well as the Secure OnBoard Communication module, are part of the Communication Services functional group.

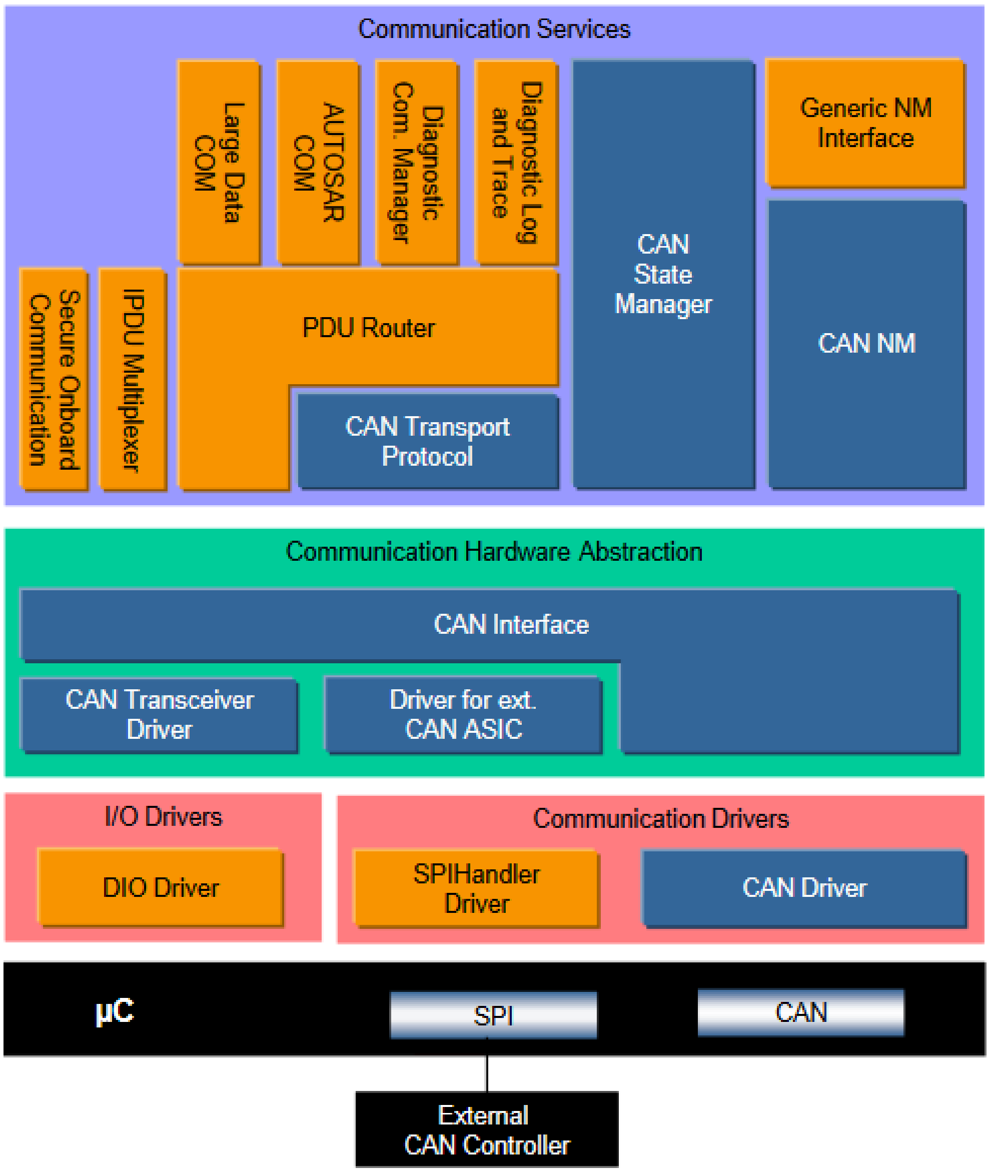


Figure 2.2: View of the COM Stack with the CAN Communication Stack modules highlighted

Within the COM Stack we have a series of modules devoted to CAN bus communications: since they span through all the BSW layers, they're referred to as "CAN Communication Stack". The CAN Communication Stack supports both the Classic CAN communication (CAN 2.0) and the CAN FD communication (if supported by hardware). [11]

2.2.3 Hardware and tools used

The CAN communication is being integrated and tested on two NXP S32K144 boards that act as ECUs. The configuration files were generated with EB Tresos and the Basic Software is being programmed in C with the NXP Design Studio IDE. Both the board and these tools will be presented in the next chapter as they were used in the main proof of concept as well.

2.3 Securing the communication

As we already said, securing the communications between ECUs within a car means to ensure data authentication. Data authentication is achieved by means of cryptographic algorithms, be them symmetric or asymmetric, that append some information to the data that can be used to ensure it was sent by the expected sender. There are two main problems to address: how to use such cryptographic algorithms and how to distribute to the ECUs the keys that they use. The first problem was addressed thanks to an AUTOSAR module of the COM Stack: the “Secure Onboard Communication” module; for the second problem the solution proposed is to rely on a CAN architecture called "TAURUM P2T".

2.3.1 Secure Onboard Communication

The Secure Onboard Communication (SecOC) module is a Service layer module that provides resource-efficient and practicable authentication mechanisms for critical data on the level of Protocol Data Units (PDUs, also called I-PDUs, i.e. messages transferred between the layers of the AUTOSAR stack [69]).

To achieve message authentication by default this module uses a symmetric MAC, but AUTOSAR also specifies how to use it with asymmetric solutions (i.e. digital signatures); both MAC calculation and digital signature algorithms are provided to this module by the Crypto Stack. The MAC added to an I-PDU is called “Authenticator”.

The SecOC modules also provides freshness verification: an external Freshness Value Manager (FVM) may add a Freshness Value to the I-PDU calculated from a counter or a timestamp; this way the messages is protected against replay attacks. An I-PDU that requires authentication is called “Authentic I-PDU”, and an Authentic I-PDU with attached an Authenticator and optionally a Freshness Value is called “Secured I-PDU”.

Let’s see the flow of the communication between a sender and a receiver using the SecOC module: on the sender side, the SecOC module creates a Secured I-PDU by adding authentication information to the outgoing Authentic I-PDU.

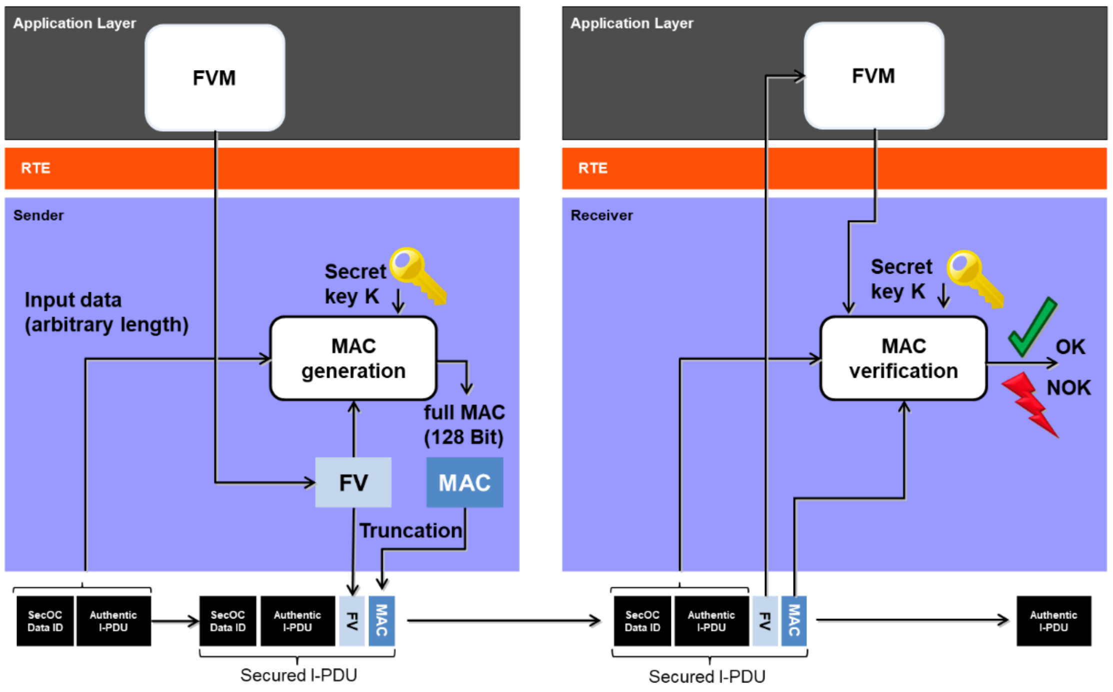


Figure 2.3: Message Authentication and Freshness Verification flow

Regardless of if the Freshness Value is or is not included in the Secure I-PDU payload, the Freshness Value is considered during generation of the Authenticator. When using a Freshness Counter instead of a Timestamp, the Freshness Counter should be incremented by the Freshness Manager prior to providing the authentication information to the receiver side.

On the receiver side, the SecOC module checks the freshness and authenticity of the Authentic I-PDU by verifying the authentication information that has been appended by the sending side SecOC module. To verify the authenticity and freshness of an Authentic I-PDU, the Secured I-PDU provided to the receiving side SecOC should be the same Secured I-PDU provided by the sending side SecOC and the receiving side SecOC should have knowledge of the Freshness Value used by the sending side SecOC during creation of the Authenticator. [70] [71]

2.3.2 TAURUM P2T

Since the calculation of the MAC requires a secret key, each car needs several keys for messages of every communication that requires data origin authentication and integrity. For big car maker companies, that sell millions of cars per year, the investment for securely distributing many keys to that many cars is considerable.

The solution to this problem would be to handle key creation and distribution within each car: that's where the TAURUM P2T architecture comes in handy.

TAURUM P2T is an Advanced Secure CAN-FD Architecture that was designed and developed by the Control and Computer Engineering Department of the Politecnico di Torino together with researchers from PUNCH Torino S.p.A. The main idea of TAURUM P2T is to split the data communication and the key distribution in two separated CAN networks (see Figure 2.4).

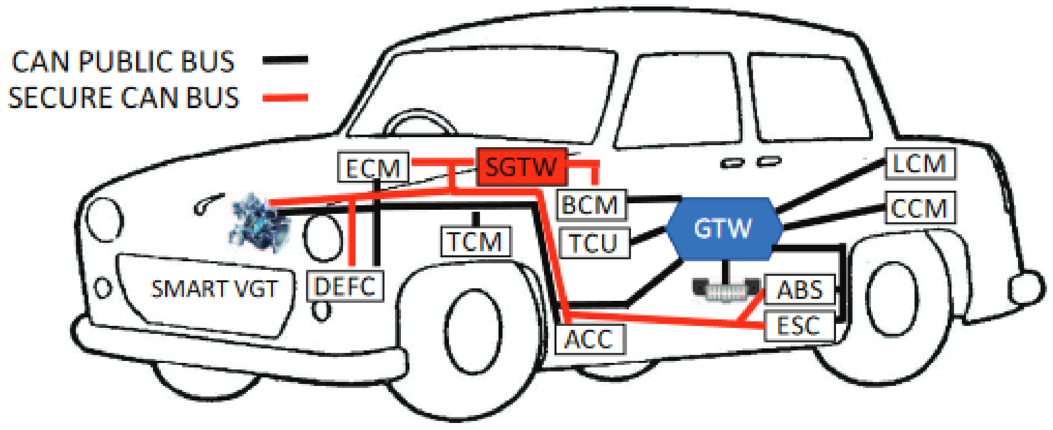


Figure 2.4: TAURUM P2T split networks

The Public CAN network (depicted in black) and accessible through the standard CAN Gateway (CGTW) transmits the standard vehicle CAN traffic. The Secure CAN network (the red one) exchanges sensible information to handle shared keys and security violations. All frames in this second network are encrypted, and the network is only accessible through the TAURUM P2T Secure Gateway (SGTW). The SGTW establishes privilege levels and manages secret keys required to compute MAC signatures. The keystones of TAURUM P2T are:

- a sharing key mechanism governed by the SGTW and able to define separated trust zones;
- an SGTW controlled sub-domain management of the bus for ensuring segregation;
- a rolling MAC secret key infrastructure to implement a countermeasure to MitM and replay attacks.

The keys are generated at vehicle initialization after the SGTW performs a network discovery. [72]

Chapter 3

Proposed approach

This chapter will be about the actual project designed, starting from why it was designed; we will see an overview of the whole architecture and then specifically we will talk about the modules developed, with a particular focus on the custom transformer. At last, we will see some of the real-world applications of the architecture and a solution for dealing with key distribution.

3.1 The need for this architecture

As our cars become more and more complex, the networks between the many ECUs are growing larger and the communication between them is vital for the vehicle and hence for the passengers. But in the future cars will rely more and more on the communication with the outside, be it other cars or external operators [1].

At some point vehicles will be entirely dependent from external networks, but in the near future small applications could definitely improve our lives and so different car manufacturers are developing and testing their own solutions for implementing this kind of communication, which is generally referred to as “Car2X”: for example, Volkswagen designed a system based on the Wifi-p wireless standard that can allow traffic hazards or emergency service vehicles to communicate their position to nearby cars [73]; Audi’s “C-V2X”, that uses the 4G/5G mobile communication standards, aims at improving the security of schoolchildren and construction workers by telling cars when they’re approaching schools and school buses or construction sites [74].

These solutions don’t account for communications between cars, as they are proprietary solutions and wouldn’t work with cars from different manufactures. However, these applications would highly benefit from a standardized protocol of communication as it would allow to have potentially every car on the roads connected to each other through the same infrastructure and let the drivers drive in a more secure and efficient way.

With that in mind, the main goal of the architecture designed was to be efficient and as secure as possible, and also to be easily standardized: for these purposes, it revolves mainly around two instruments, AUTOSAR's transformers and the MQTT protocol.

Currently AUTOSAR has specified only 3 transformers, all belonging to the COM stack, but has also given the requirements for custom transformers [14] that would be treated as Complex Device Drivers (CDD).

The MQTT protocol is performance-wise the cutting edge communication protocol for IoT devices and embedded systems alike, and as its entities are very lightweight it is the perfect candidate for being integrated into ECUs. The only problem is that, used as-is, it is not very secure, but we will see how to account for that.

3.2 Overview of the architecture

We will now have a look at the whole flow of a message from when it is generated to when it arrives at the consumer; this will include modules and solutions already treated in the previous chapter.

Figure 3.1 represents the whole schema here discussed. Note that the different colors on the ECUs represent different layers of the AUTOSAR architecture: the grey one is the Application Layer, the orange one is the Runtime Environment layer, the blue one is the Service Layer, the green one is the ECU Abstraction Layer and the red one is the Microcontroller Abstraction Layer.

First of all, we begin by a data collected by a sensor (or another component): it could be for example the information of a fault in the brakes system, or the geographical coordinates retrieved by a dedicated module. This data is sent to the Software Component that serves in our application as the Main SW-C: it could be the same SW-C that collected the data, it could also be another SW-C on the same ECU, but for the sake of explanation here it was treated as a separated SW-C that runs on a dedicated ECU (just called "Main ECU" in the schema).

The Main SW-C sends the data to the Runtime Environment, which calls MQTTXf: the RTE allocates a buffer of the appropriate dimension, puts the data in this buffer and calls a function of MQTTXf passing the buffer as argument.

MQTTXf is the transformer which is the core of our project, and we will see it in detail later; in general, it takes a buffer as argument, transforms it by encrypting and appending a MAC using functions from the Crypto Stack, formats it into an MQTT protocol packet and returns the message transformed to the RTE (on the same buffer if we are using on-place buffering, like in this example, or in another buffer in case of out-of-place buffering).

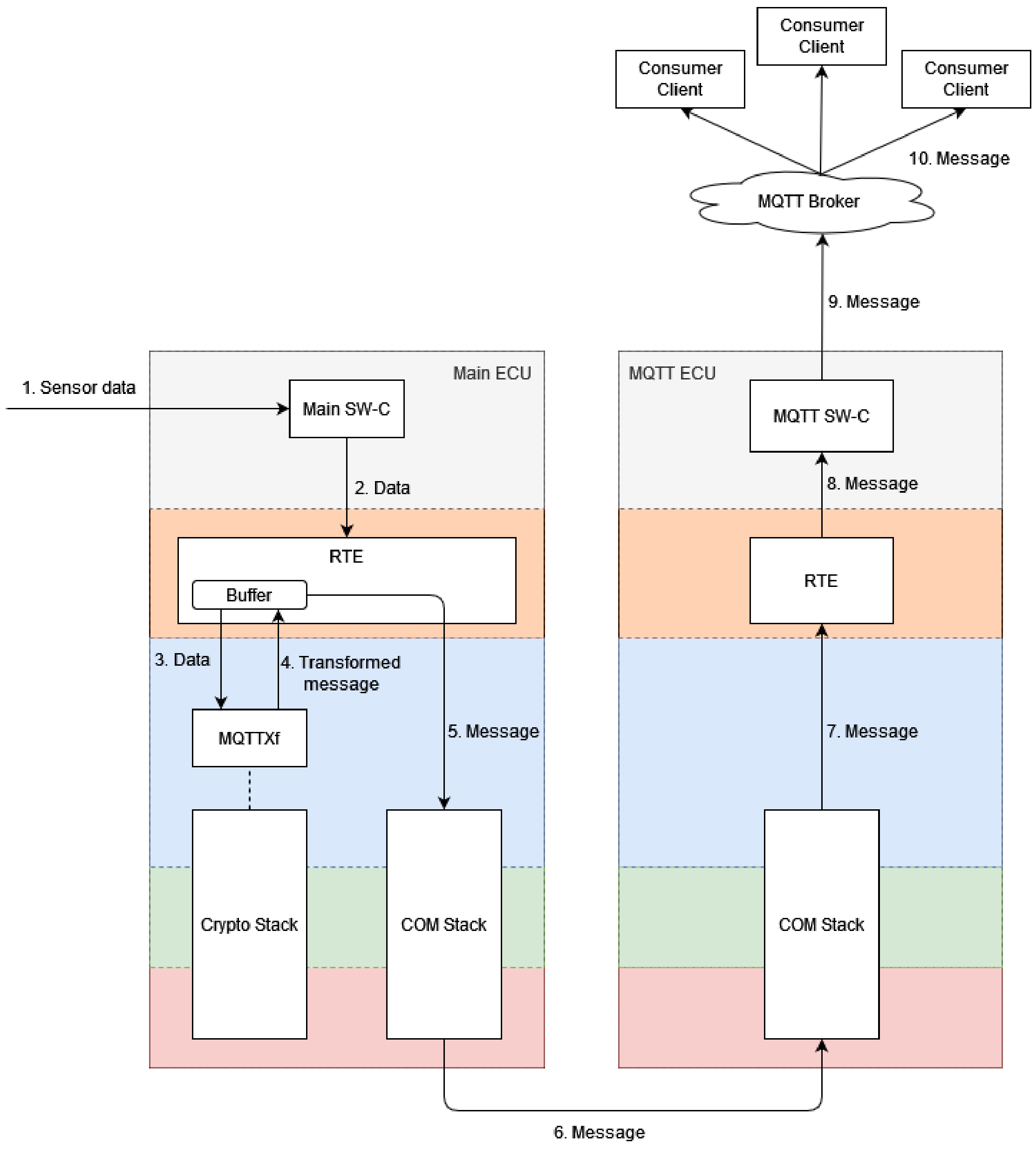


Figure 3.1: Message flow from ECU to MQTT Broker

The RTE then passes the transformed message to the COM Stack for it to be sent to the entity devoted to communications with external networks: again, this recipient could be another SW-C on the same ECU, and the RTE could also just send the message back to the same Main SW-C, but in order for the schema to be more clear (and to achieve a certain separation of concerns) the communication with external networks is handled by a dedicated application in a dedicated ECU (which in the schema is called MQTT ECU).

The MQTT ECU receives the message through the COM Stack to the RTE, which sends it to the appropriate SW-C (called MQTT SW-C in the schema). This SW-C connects to an MQTT Broker and publishes the message in a specific topic. The Broker then forwards this message to the consumer MQTT Clients subscribed to the same topic, which proceed to decrypt and verify the authentication of the message and elaborate it. All the details of the functioning and message transmission of the MQTTXf transformer and the MQTT SW-C will be talked about later.

The MQTTXf transformer handles the security of the communication with the consumer MQTT clients, using keys shared between the Main ECU and the clients (we will see how), but the communication between ECUs and components also must be secure: we must ensure data origin authentication and integrity in order to know that each message was sent by the rightful sender and was not modified during the exchange. In our schema the passages where we must address this issue are the communication of the data from the sensor (or another SW-C) to our Main SW-C (passage 1 in figure) and the communication of the transformed message from the Main ECU to the MQTT ECU (passage 6).

To this purpose we will use the solutions explained in Chapter 2: for handling the authentication and integrity of the data we use the Secure Onboard Communication module, and for generating the keys used by this module for encryption and MAC calculation we use the TAURUM P2T architecture; the communication itself is carried out through the COM Stack.

3.3 The MQTTXf transformer

The custom transformer that will be presented here takes a buffer as an argument and does three things: it encrypts the buffer, calculates the MAC of the buffer and appends it on the buffer, then formats the buffer as an MQTT protocol packet optionally adding some identification information. The order of the first two operations, the level of security of the used algorithms and the additional identification information can be changed by configuring the state of the transformer. The inverse transformer function is also defined.

As we can see in Figure 3.1 it is, like all other transformers, part of the Service Layer of the Basic Software. The name “MQTTXf” was chosen because it is used to make a message ready to be transmitted over MQTT, and “Xf” is standard AUTOSAR notation for transformers (e.g. the COM based transformer is called

“ComXf”). As this is a custom transformer, and is one of a kind, the numeration in all its nomenclature is not explicit (as it should by AUTOSAR specifications).

Using a transformer has many advantages for our architecture over using the Crypto Stack directly from the RTE. First of all, the Crypto Stack as we will see is not very simple and straightforward to use: it expects several preliminary configurations and each single operation (for example the encryption of a message) requires different function calls; moreover, the SHE must also be directly interfaced with for retrieving keys and optionally the identification information. Our transformer hides the complexity of the Crypto Stack and the SHE under a simple and clear function.

The transformer can also be configured for achieving a different compromise between security and performance, but by default it works with the most secure configuration without needing to change anything (thus following the “Establish secure defaults” OWASP principle [75]).

Another advantage of this transformer is that, being it a transformer, can be chained together with other transformers: for example it can be chained with the COM based transformer, thus making communications faster than by having the encryption and authentication separated.

3.3.1 Tools and hardware used

Let’s know talk about the tools used for developing this transformer and the hardware it was tested on.

Starting from the hardware, the transformer was debugged and tested on an NXP S32K148 Evaluation board [76]. It mounts a 32-bit Arm Cortex-M4 microcontroller and has 8MB of integrated flash memory. As communication interfaces, it presents connectors for CAN, LIN and UART/SCI. It was debugged using the OpenSDA debug adapter with an USB interface.

The transformer was developed in C language, following as much as possible the "Secure coding" directives of the ISO-21434 standard [77]. The two main programs used were EB Tresos Studio and NXP S32 Design Studio. EB Tresos Studio, by Elektrobit, is a tool for ECU Basic Software configuration, validation and generation [78]; it was used for generating the configuration files needed for integrating the Crypto Stack modules in the project. NXP S32 Design Studio is an IDE, based on the Eclipse IDE, for automotive and Arm-based microcontrollers software development [79], and was essential for developing, debugging and deploying code on the board.

Let's now have a brief look at the specific modules used for developing the transformer; of course, since we used an NXP board, the NXP implementations of the AUTOSAR modules were used, together with some custom NXP modules: in particular, the latest available version of the NXP implementations at the time of developing was version 1.0.0 revision HF01_D2109 of AUTOSAR version 4.4 modules. All functions and functionalities of the modules used will not be thoroughly presented, but one complete example of call flow and hierarchy will be later discussed.

From the Crypto Stack, the modules used were the Crypto Service Manager (CSM) of the Service Layer, the Crypto Interface (CRYIF) of the EAL and the Crypto Driver (CRYPTO) module of the MCAL (we already talked about these in Chapter 1); some of the functions from those modules were slightly modified to fit our needs. The Crypto Stack in this NXP drivers relies on the Software Hardware Extension (SHE) for the cryptographic algorithms implementations. The other modules used were NXP custom modules: the RTE module, for handling the buffers, the DET module, useful for memory operations, the BASE module, which contains common files/definitions needed by the MCAL, and the ECUC and RESOURCE modules, used by EB Tresos for configuration purposes.

3.3.2 Configuration

The transformer is configured by feeding a struct of a custom type ("MQTTXf_ConfigType") with all the required parameters to the initialization function of the transformer ("MQTTXf_Init").

Since this project used NXP's implementation of the AUTOSAR modules, which relies on the SHE, it was possible to test only the algorithms used by the SHE. So the configuration parameters of the transformer account only for the algorithms available, but extending the parameters is very easy.

The parameters that can be changed are:

- the encryption algorithm: the default and only one supported by the SHE is AES-128; can be 0 in case of no encryption
- the encryption mode: default is CBC, SHE also supports ECB; can be 0 in case of no encryption
- the MAC calculation mode: SHE only supports CMAC; can be 0 in case of no authentication
- the encryption algorithm used by the MAC: SHE only supports AES-128; can be 0 in case of no authentication

- the authenticated encryption mode: the possibilities are no encryption and no authentication, only authentication without encryption, only encryption without authentication, encrypt-and-MAC, encrypt-then-MAC, MAC-then-encrypt (for the difference between those modes refer to Chapter 1: authn encryption); by default is encrypt-then-MAC
- the additional data attached for identification: can be nothing, the license plate or the Vehicle Identification Number (VIN, an ISO standard for uniquely identify vehicles [80]); by default it's the VIN.

3.3.3 Exposed interfaces

We will now have a brief look at the functions provided by the MQTTXf transformer, then in the next chapter we will see in detail how an example of such a function works with a determined configuration. The nomenclature of the functions is standardized by AUTOSAR, with the "transformerId" field substituted by "Transform" as this transformer is one of a kind.

The transformer also defines a type, "MQTTXf_ConfigType", whose fields were described in the previous section and so they will not be further discussed.

- *MQTTXf_Init*: it takes a configuration structure of the aforementioned type "MQTTXf_ConfigType" as a parameter and initializes the state of the transformer according to this structure.
- *MQTTXf_DeInit*: it takes no parameters and deinitializes the transformer.
- *MQTTXf_Transform*: it takes as parameters a buffer and its length, reads the state of the transformer and, if required, encrypts the buffer (by first applying PKCS#7 padding if necessary), calculates its MAC, appends the MAC to it, adds the required identification information and formats it according to the MQTT packet format; the cryptographic algorithms are provided by the Crypto Stack, and the keys used are taken from a specific location on the SHE well-known to the transformer; it returns "E_OK" (which is 0) if everything went well and an error-specific code otherwise; it is also possible to call the function with out-of-place buffering, in which case it also requires an input buffer and its length as parameters.
- *MQTTXf_Inv_Transform*: the inverse of the transform function; it takes as parameters a buffer and its length, reads the state of the transformer and, if required, verifies the MAC appended to the buffer together with the additional information and decrypts the buffer; the cryptographic algorithms are provided by the Crypto Stack, and the keys used are taken from a specific location on the SHE well-known to the transformer; it returns "E_OK" (which is 0) if

everything went well and an error-specific code otherwise; it is also possible to call the function with out-of-place buffering, in which case it also requires an input buffer and its length as parameters.

3.3.4 Example of behavior

We will now see an example of how a function call works internally, as this will serve to prove how much easier and efficient is the use of this transformer instead of calling the Crypto Stack and the SHE manually. For the sake of simplicity and shortness, the case that will be presented requires the transformer only to encrypt the message, without authenticating it, in ECB mode: this is because encryption is simpler in terms of calls to the Crypto Stack, and ECB mode doesn't require an initialization vector, but keep in mind that just with authentication the calls are double in number.

The numbers of the operations relate to the numbers in Figure 3.2. Of course, except for the transformer functions, all other functions are written as pseudo instructions (that could also hide more than one function call or operation) as to not overcrowd the diagram, but a brief explanation of all the flow will now follow.

The actors in place are the Runtime Environment (RTE), the transformer MQTTXf, the Secure Hardware Extensions (SHE) and the Crypto Service Manager (CSM).

1. The RTE creates the structure for configurating the transformer; it sets the authenticated encryption mode to just encryption and the encryption mode to ECB, leaving all other values as default.

2. The RTE calls the transformer initialization function passing the aforementioned configuration; this is necessary here because we are not using the default configuration, which is the most secure.

3. The RTE allocates the buffer, copies the message to encrypt into it and defines its length.

4. The RTE calls the transform function of the transformer with in-place buffering, passing as parameters the buffer and its length.

5. The transformer reads its state to know what functionalities are required.

6. It then initializes the SHE.

7. It also initializes the CSM.

8. The transformer then asks the SHE the key that it will use for the encryption. The specific key is retrieved with a sequence number (well-know by the

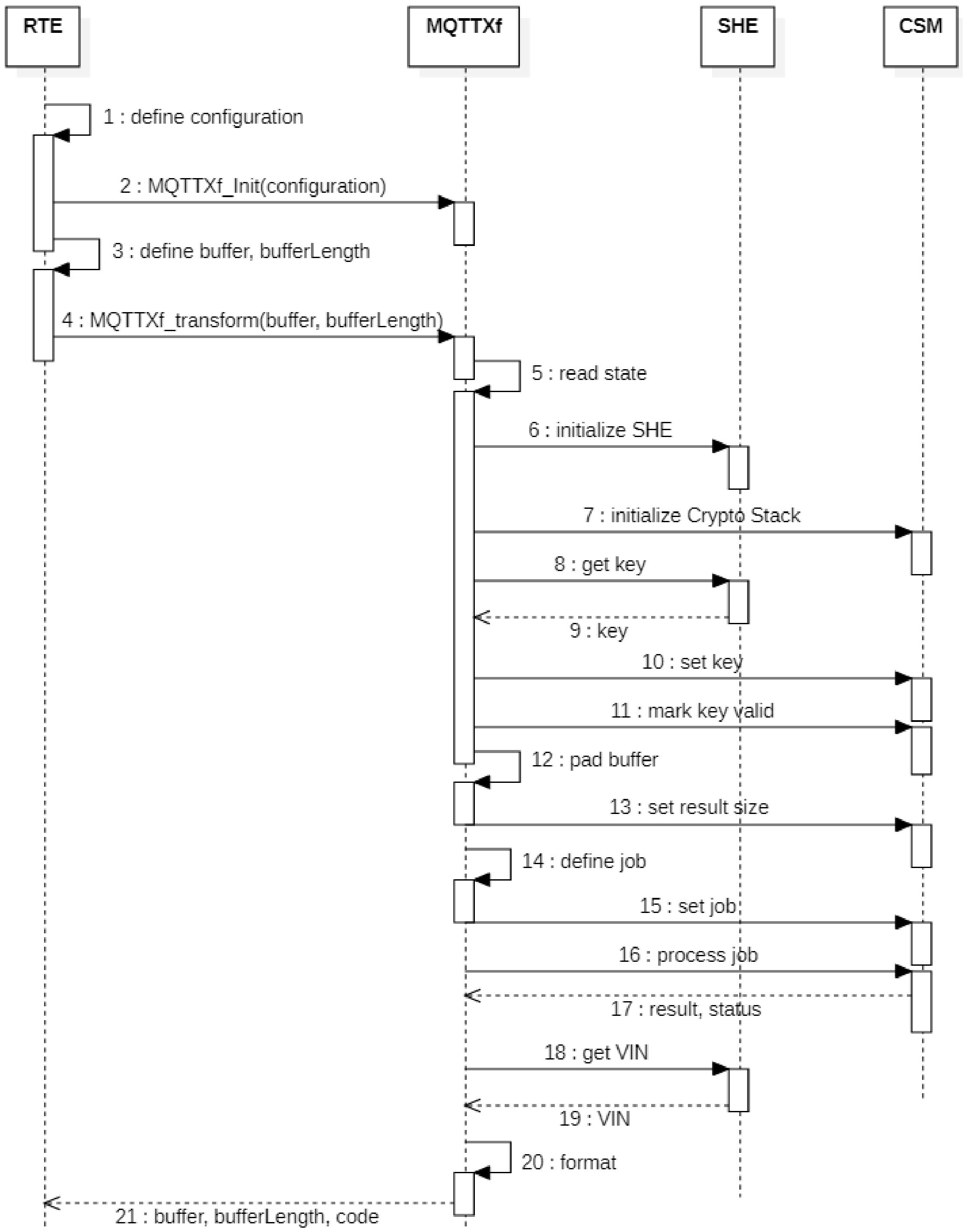


Figure 3.2: Sequence diagram of the MQTT_Transform function

transformer) that uniquely identifies the particular key that the SHE must fetch.

9. The SHE returns the required key to the transformer.
10. The transformer communicates the key, its size, its type and the ID associated to it to the CSM.
11. It then marks the key as valid for the Crypto Stack so it can be later used by future requests; this is useful when both encryption and authentication are requested, but it's done nonetheless.
12. It proceeds by checking if the size of the buffer is a multiple of the encryption algorithm block size, and applies PKCS#7 padding otherwise.
13. The transformer communicates to the CSM the max length in bytes of the buffer where it can put the result of the encryption.
14. The transformer creates a structure that serves as configuration for the Crypto Stack to know the specifics of the operation to perform: for example, it contains the pointer to the message plaintext and the pointer to the space allocated for the cyphertext (in our case they are the same buffer).
15. The transformer communicates that structure to the CSM.
16. The transformer then tells the CSM to perform the encryption with the specified parameters.
17. The CSM returns the encrypted message and a status code.
18. The transformer asks the SHE the Vehicle Identification Number. It's not usually a duty of the SHE to keep the VIN, but for simplicity's sake here it is stored like a key.
19. The SHE returns the VIN to the transformer.
20. The transformer adds the VIN to the buffer and formats the message as an MQTT packet.
21. The transformer returns the control to the RTE, together with a result code.

3.4 Other implemented modules

We will now have a brief look at the implementation of the MQTT SW-C and the MQTT Broker.

3.4.1 The MQTT SW-C

This module was not the main purpose of the project, hence it was not implemented as an AUTOSAR standard SW-C, but it was very important for testing the implementation of the transformer as it was the only way its functionalities could have been verified.

The SW-C consists in an Arduino script, written with the Arduino IDE [81], and was being run on an ESP8266-01S Wi-Fi module [82] which acted as the MQTT ECU.

The script first of all connects the ECU to the Wi-Fi, then it polls the Serial interface until it receives a message. It then loops a connection request to the MQTT Broker, and as soon as it is connected, it subscribes to a given topic and publishes the received message to it. It then closes the connection with the Broker and returns to poll the Serial interface. The MQTT messages are sent with QoS 0.

The Wi-Fi connection was handled through the "ESP8266WiFi" library [83], while the MQTT functionalities were provided by the "ArduinoMqtt" library [84], which is based on Eclipse Paho.

3.4.2 The MQTT Broker

For verifying the functionalities of the transformer, a very simple Mosquitto MQTT broker was built, configured for having no authentication and using the default listener on the default port. No SSL/TLS connection was used at that time.

3.4.3 The consumer client

For testing purposes also a consumer client was developed, in order to acknowledge that the messages were really sent and that the inverse transformer function worked. The client was developed as a Python script, using "mqttools" for the MQTT functionalities.

The client connects to the broker, then subscribes to the same topic the MQTT ECU is subscribed and as soon as it receives a message it performs the following operations:

1. Prints the received message as-is on the screen
2. Isolates the Vehicle Identification Number and prints it on the screen
3. Then it calls a function which performs the same operations as the inverse transformer function of the MQTTXf transformer, passing the received message and its dimension as parameters

4. The function performs the verification of the MAC by separating the MAC from the encrypted message, calculating the MAC on the encrypted message and checking if it is the same as the one sent

5. If the MAC verification is not successful, it returns a specific error code

6. Otherwise it proceeds with the decryption of the message

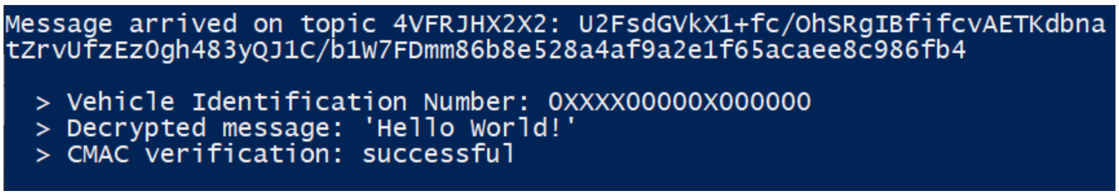
7. If the decryption is not successful, it returns a specific error code

8. Otherwise it changes the parameters to the decrypted message and its length (as with in-place buffering) and returns 0

9. The main checks the return value of the function

10. If it is not 0 it prints the error

11. If it is 0 it prints the decrypted message and a line that indicates a successful MAC verification



```
Message arrived on topic 4VFRJHX2X2: U2FsdGVkX1+fc/OhSRgIBfifcvAETKdbna
tZrvUfzEzOgh483yQJ1C/b1W7FDmm86b8e528a4af9a2e1f65acaee8c986fb4

  > Vehicle Identification Number: 0XXXX00000X000000
  > Decrypted message: 'Hello World!'
  > CMAC verification: successful
```

Figure 3.3: Consumer client

In Figure 3.3 we can see an example of correct functioning of the consumer client.

3.5 Use cases

Communications between cars and external networks can have plenty of use cases. We will now describe some of them that are relevant for showing the different security and performance requisites that were taken in consideration during the testing of the proposed architecture.

3.5.1 Traffic control

One of the most useful possible applications of such an architecture would be letting car drivers know about very crowded roads so they can choose another path. This

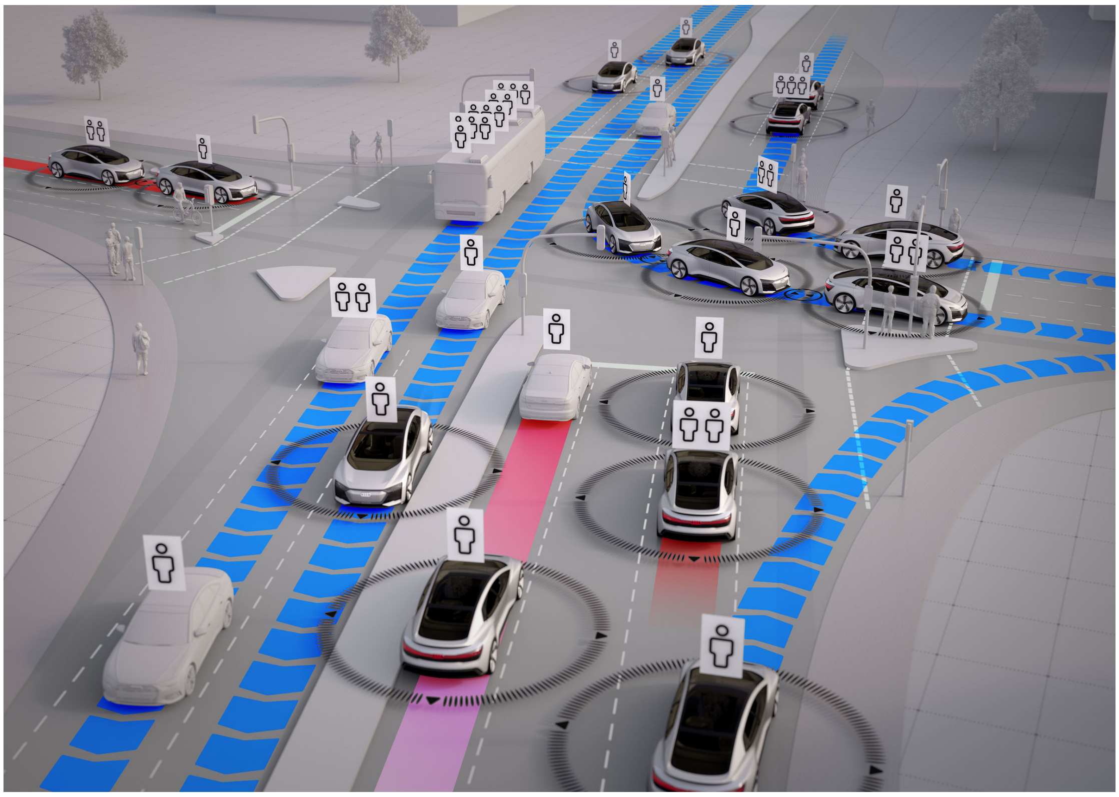


Figure 3.4: A scenario where some cars have Car2X functionalities [85]

feature of course already exists in satellite navigators, but it's not very accurate and it's not available everywhere.

Using communications between cars and external networks, such a feature could be achieved in two ways.
The first way is to have a centralized entity that collects all the info from cars within a geographical radius; then it intersects such information and updates a map that shows the most crowded roads and suggests alternate paths; this map is provided to drivers for example via a mobile application. This way expects the use of said centralized entity which could be a bottleneck of the architecture.
The second, most ambitious way would be to rely entirely upon exchanging information between cars, eliminating the need for external entities: cars could broadcast their positions and by analyzing two successive messages, one could know the direction they're travelling to and their speed (or these info could be sent with each message). In this way each car could know the position and direction of all cars within a certain radius and calculate which path is the less crowded.

This application is also useful in another important scenario: sometimes when we're at a crossroads we hear the sirens of an emergency service vehicle but we don't know where it's coming from and so we don't know how to react. With this architecture we could know exactly where it is coming from, and we could have enough time to react properly [73].

This architecture however heavily relies on communication between cars, which we know are made by many different manufacturers that build them in very different ways. And since cars would exchange geographic coordinates (as well as optionally other information such as their speed and direction), if every car treats such data differently the architecture cannot work. That's why it would be important in such a scenario to have a standardized protocol for the data exchanged.

Whichever is the way chosen, in this use case the cars are exchanging coordinates in possibly high-speed situations, so the generation and transformation of the data should be as fast as possible. Fortunately, since we want that information to be known to all near cars, we don't care much about confidentiality, so encryption of the data is not needed (which saves some time).

3.5.2 Faults communication

Another way in which communications from cars to external networks could improve drivers' life is with faults communication: one or more ECUs inside the car could collect all the data about hardware faults (e.g. the breakdown of a sensor or the failure of a brake) and send them in real time to an external server. This server could analyze such data and decide if and how to intervene: for example if it isn't a crucial failure it could just inform the driver, or on the other hand if the fault can have (or already has) severe consequences it could alert some security services (like an ambulance).

This system can also benefit from the use of vehicle to vehicle communication: in case of serious failures that could led to the malfunction of the whole car, the fault could be communicated to near vehicles so they can move to a safer distance from the malfunctioning car or create a clear path to the emergency lane.

The performance and security requirements of this application could be different based on what information needs to be sent. If the fault is not crucial, the time restraints can be looser, but if the life of the driver and near ones is at stake, the communication should be as fast as possible. From a security point of view, in order to identify the particular car there may be the need to send to an eventual handler server (or security service) some information that we don't want to be disclosed (e.g. the license plate or the Vehicle Identification Number). For those reasons it is important to give the possibility of tweaking the performance and security feature to address for different needs and scenarios.

3.5.3 Highway toll payment

The last use case considered is the use of communications from car to external networks for paying the highway toll "on-the-fly" while driving and without needing any operation from the driver. The current electronic toll payment systems used in some highways (for example the Italian "Telepass" [86] or the Spanish "Via-t" [87]) expects the driver to drastically slow down when approaching a toll booth or gate.

Instead we want to let the drivers keep the speed they're travelling at and make the car carry out the payment procedure without them even noticing.

The security requirements of this applications are very strict, since we're exchanging information related to payments.

3.5.4 Implementation of the use cases

These use cases should not be considered as separate in a real-world scenario: they could all be part of the same interconnected vehicles architecture. Let's now discuss how such an architecture could be implemented, focusing on the MQTT actors involved.

While driving, MQTT ECUs of the vehicles are connected to MQTT brokers that, of course, accept subscriptions to topics and forward published messages to all vehicles subscribed to the same topic. The world is divided in geographical areas (not obligatorily of fixed size), and the range of each MQTT broker includes several geographical areas. Each geographical area corresponds to a given MQTT topic to which every vehicle that enters said area should subscribe. All vehicles in an area thus publish messages (about their position or their hardware faults or highway toll payment information) to the topic corresponding to the area they're in, and they're all forwarded messages from vehicles in the same area. To let the vehicles know the topic which they should subscribe to, there are different possibilities.

One possibility would be to have the drivers specify their destination on an external device (for example a mobile phone) and, based on the path to traverse, the device would preventively download all the topics of the areas that will be crossed (and nearby ones). These topics would then be transmitted to the MQTT ECUs. Of course in this way the Car2X functionalities would only be available for the path chosen, and won't be available if the driver completely changes the path.

Another solution would be to have the topics set in a specific format that could be calculated autonomously by each vehicle based on its coordinates. For example it could be used a system like Google Plus Codes [88], that associates an alphanumerical code to each latitude and longitude coordinate, or similar to "what3words" [89], that associates each 3x3 meters area to three unique dictionary words.

The last proposed solution is to have an MQTT Client for each geographical area, which we'll call "gateway", that whenever asked publishes the topic for that area in a well-known topic, which is the same for all gateways (for example it could be "topic_request"). When a vehicle enters a geographical area for which he doesn't know the topic, it could subscribe to this topic and send its coordinates to it; based on these coordinates, the gateway publishes the topic of the area, together with some info for identifying the vehicle, to the well-known topic.

3.6 Key distribution for Car2X communications

In this project there is an extensive use of encryption and authentication mechanisms for securing the communication between an ECU and an entity connected to an external network. But in order for the presented architecture to work, the ECU and said entity must share common secrets to use as keys for the encryption/decryption and MAC calculation/verification algorithms. Having those keys embedded in the ECUs at manufacturing time would be very costly (because big car making companies make tens of millions of cars with hundreds of ECUs each) and not very secure, since the compromise of a key would mean the compromise of all the communication until that key is physically changed.

A solution to this problem would be to have the key distribution happen "on-the-fly" while driving: that way we eliminate the need to securely embed millions of keys in cars ECUs and we could substitute the key as soon as it is compromised. To achieve this feature a really good solution is to use the architecture designed by Alessandro Di Vincenzo for distributing cryptographic keys in an Internet of Things (IoT) environment [90].

Let's have a brief look at this architecture and how it could be integrated in our use case.

3.6.1 Overview

Already existing solutions for protecting data over the MQTT protocol in the IoT world use pre-shared keys embedded within IoT devices, without the possibility of revoking the key when disclosed without physically accessing the devices.

The proposed architecture builds upon a method called "separation of knowledge": instead of having a single entire key previously embedded in a device, we embed only one part of the key in the device and we send the other part to the device via a secure channel. The two parts of the keys are then passed as arguments to a key derivation function (which, of course, must be non-invertible and resistant to collisions) and we obtain the final key. Without both parts of the key one cannot have the full knowledge of the shared secret, hence secure communication is not guaranteed.

This way, if the final key or the transmitted part of the key is disclosed, we are allowed to change only the transmitted part of the key without having to physically change the embedded part (or the whole device).

3.6.2 Implementation

For the sake of easier explanation, let's call K1 the part of the knowledge pre-shared between the interested parties and K2 the part transmitted over the secure channel.

The entity that distributes the transmitted part of the key (K2) is implemented as a web server, with a path associated to its IP address, that holds the variable part of knowledge of the IoT devices, encrypted. It also maintains a list of which each entry contains: the identifier associated to the IoT device for which a client requests the part of knowledge stored in the server, the name of the file which contains the key used to encrypt the requested part of knowledge, a Boolean indicating if the knowledge is still valid or if it's compromised and the deadline after which the stored knowledge must be considered not valid.

The request to the server for K2 is done via a mobile application called "secMQTT": this application lets each user connect all the IoT devices for which he wants to have K2 stored on the server. The flow of the secret transmission request from the user to the server is the following: the user selects the device for which he wants to retrieve K2 by clicking on it in the app; the app sends an http request to the server with the id of the device as path argument and with the timestamp, encrypted with the same key used by the server to encrypt K2, as body; the server fetches the key from the file associated to the requested id and decrypts the timestamp; then it verifies if the request is not too old (protection from replay attacks) and verifies in the list if the associated K2 is still valid; if those checks are passed, it sends K2, encrypted with the associated key, back to the app. K2 is then passed from the mobile phone to the IoT device via NFC.

3.7 Integration in our use case

The "separation of knowledge" model could be applied to our architecture in two different ways: through the car key or directly while driving.

In the former case, we expect the car to support digital keys (for example, Apple recently made it possible to add a car key to the Apple Wallet and to use it via IPhone or Apple Watch [91]). K1 would be embedded in the ECU, while K2 in the car key (thus it can be changed via mobile phone and NFC); then, when the driver

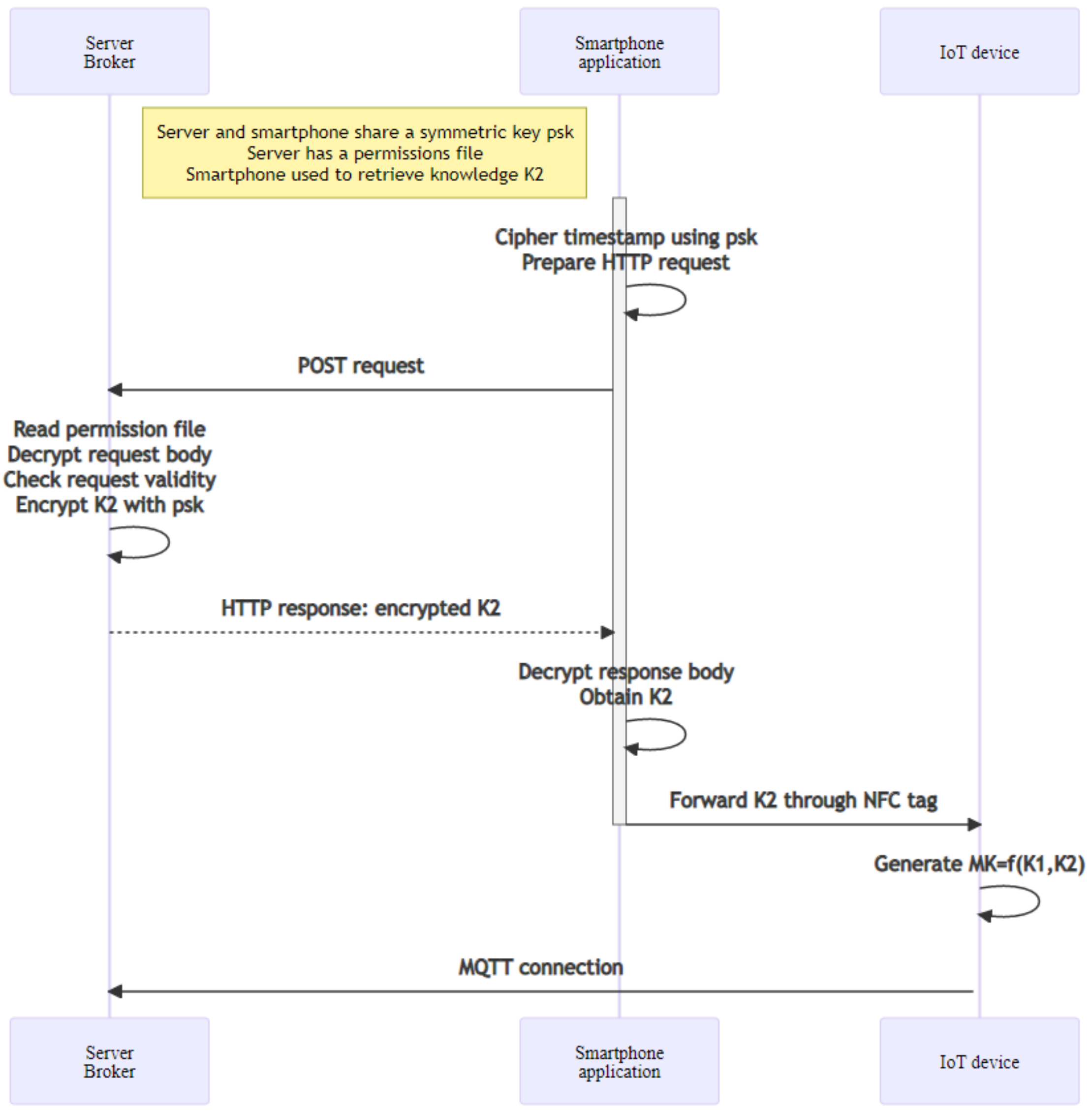


Figure 3.5: Sequence diagram of the request for K2

opens the car, K2 is transmitted to the ECU and from there the final key is built. This exchange is represented in the sequence diagram of Figure 3.6.

The other solution would be that the producer ECU and the consumer client could have K1 pre-shared and K2 sent to them directly while driving. The entity that securely sends K2 would be the same gateway that sends the MQTT topic to the ECUs of the cars that enter in a certain area.

We don't need a long-term key for communicating for a long time with the

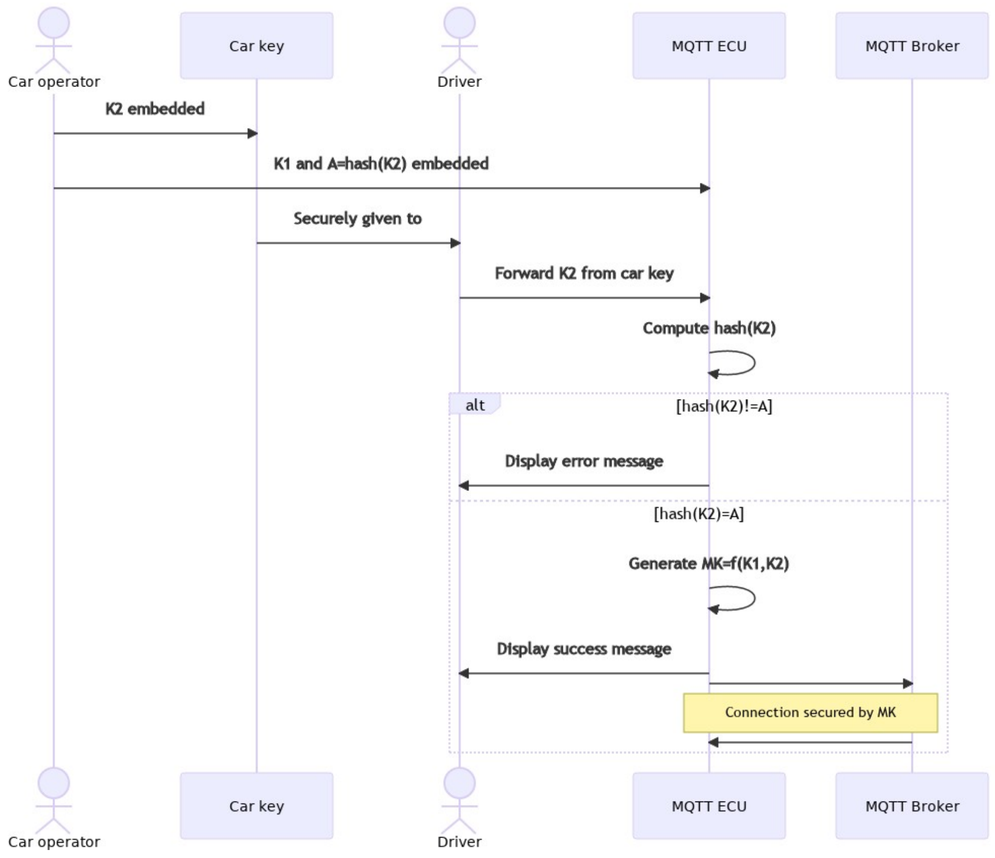


Figure 3.6: Key exchange through the car key

gateway, we use the key just for the communication happening while the car is in a specific gateway area and so K2 could just be generated on-the-fly by the gateway.

If K2 (or the final key) is disclosed, the gateway only needs to change that part by sending the new one to the car; otherwise we would have to physically change the key by tinkering with the ECU secure storage.

Chapter 4

Case study and experimental results

In this chapter we will have a look at some data extrapolated by different performance tests of the architecture shown in the previous chapter. The conducted tests are divided in two categories: tests on the transformer and tests on MQTT. The former category aims at showing how the MQTTXf transformer impacts the exchange of a message compared to sending that message in clear, and tests with different configurations of the transformer will be shown. The latter is for verifying that the MQTT protocol is actually as fast as needed.

4.1 Tests on the transformer

There will now be presented some results of tests on the performances of the MQTTXf transformer. The tests were conducted by registering two timestamps, one as soon as the data arrives at the transformer and the other when the transformer is returning it to the RTE (as we can see with the two red dots in Figure 4.1), and then calculating the difference between them.

These tests had three different goals: to show in general how much this transformer impacted the performances of the ECU (with different authenticated encryption modes), to see how much slower the safest encryption mode available (CBC) is compared to the faster but less secure one (ECB) and to verify if the performances of the transformer are affected by the amount of workload. Hence the results shown are hereby divided in three subsections.

All tests of this category were conducted in the same environment: the transformer was running on the NXP S32K148 board, which was debugged using S32

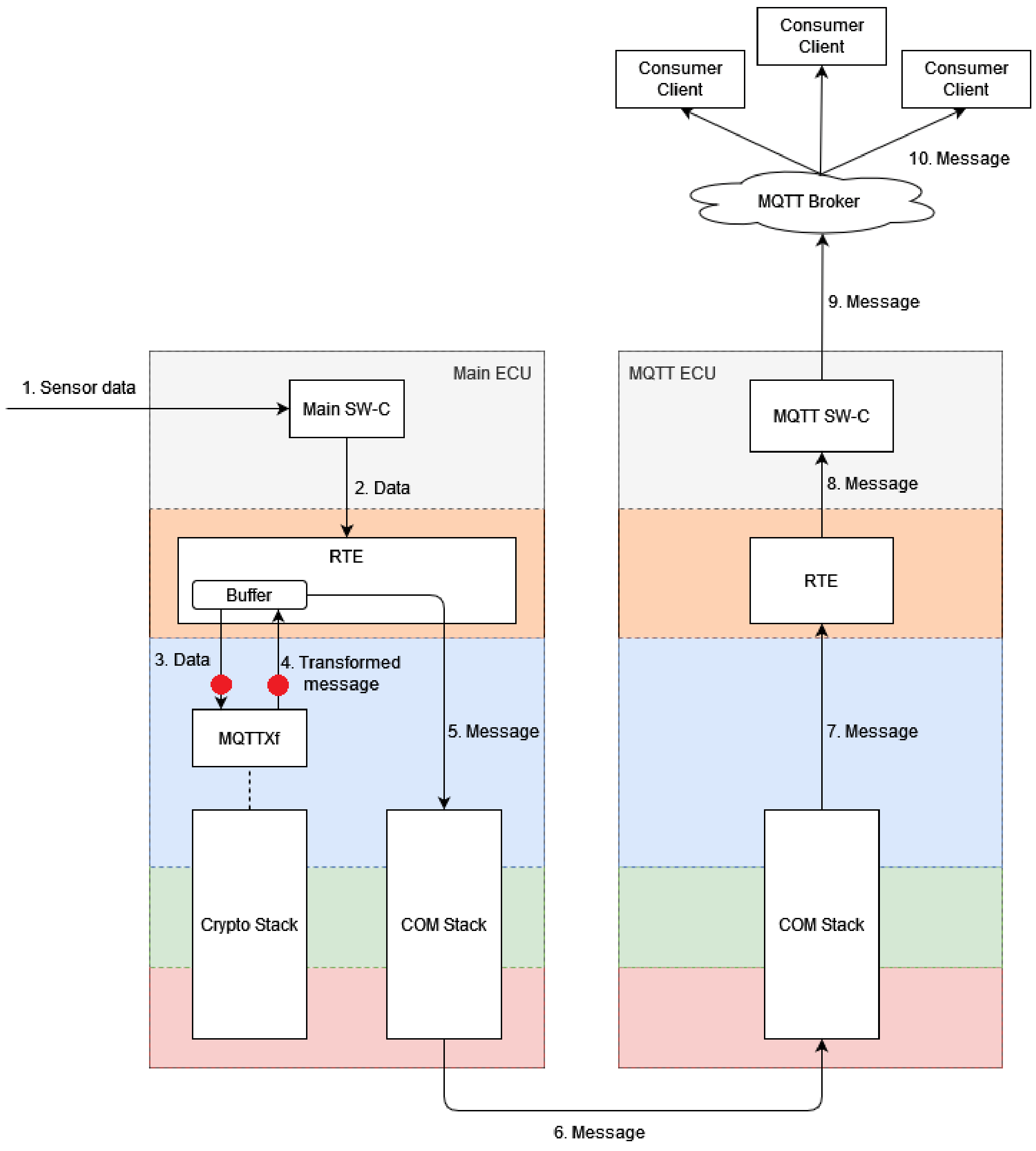


Figure 4.1: Points at which the tests were registered

Design Studio. A dummy RTE was created that just allocates the buffer, calls the transformer and cleans the buffer afterwards.

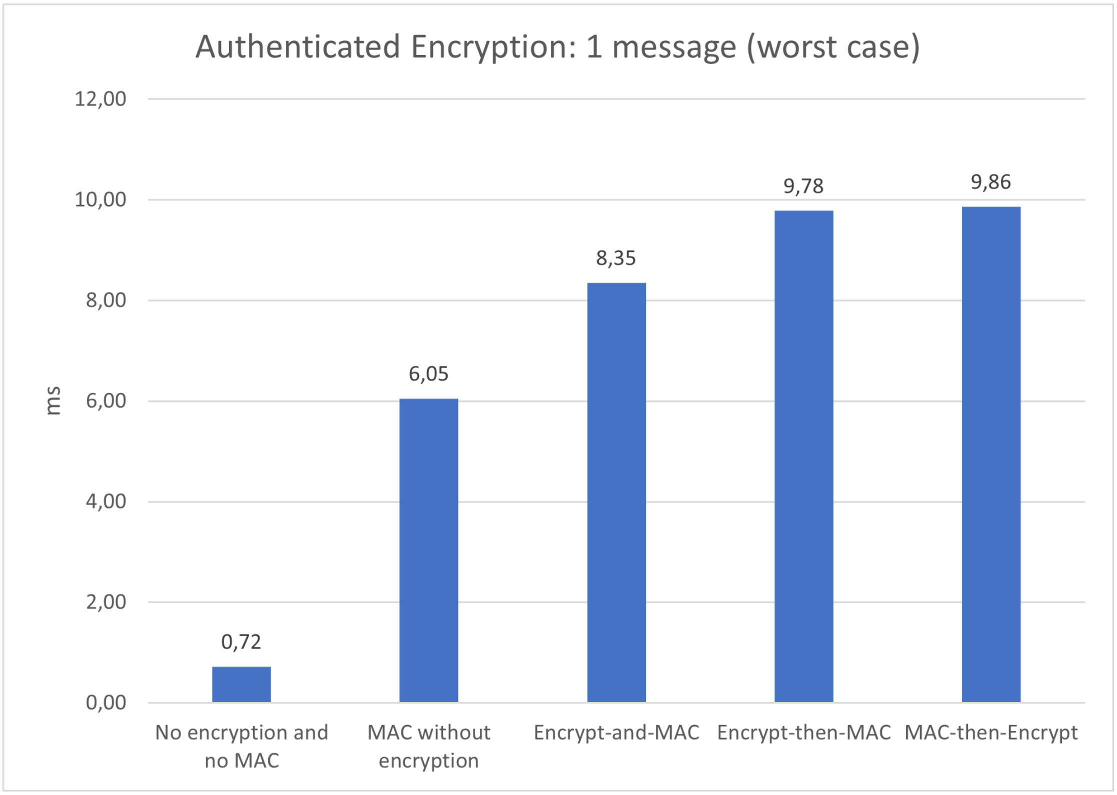


Figure 4.2: Elaborating one message with different AE modes

4.1.1 Authenticated Encryption mode

These results show how much time, expressed in milliseconds, the transformer took to elaborate messages with 5 different configurations for authenticated encryption; in figures 4.2 and 4.3, from left to right, we see the performance results for: no encryption and no MAC calculation on the message (just the formatting), authentication without encryption, Encrypt-and-MAC mode, Encrypt-then-MAC mode and MAC-then-Encrypt mode.

These results were extrapolated by 100 tests taken by asking the transformer to elaborate 100 messages: in figure 4.2 we can see how much time in the worst case it took the transformer to elaborate one message, while in figure 4.3 we see the sum of the times of all the messages. Except for the authenticated encryption mode, all other configuration parameters of the transformer were left as default; the initial messages sent were of 16 bytes length for easiness of testing and for not making the tests depend on the encryption mode, but we will see later how the transformer behaves with bigger messages.

The results show that, first of all, the encryption and authentication operations

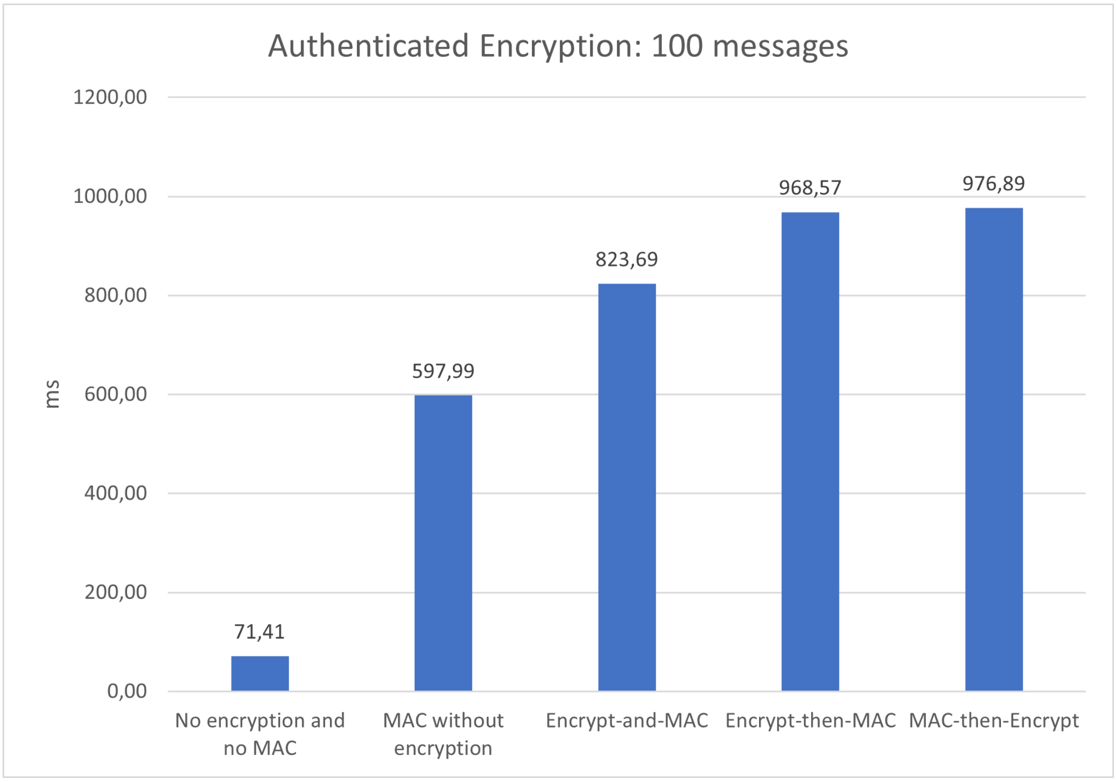


Figure 4.3: Elaborating 100 messages with different AE modes

are basically the sole factors of the time consumed by the transformer, as the elaboration of messages without them is almost instantaneous. In some use cases we saw that we don't need to have confidentiality, and only calculating the MAC of the message without encrypting it saves from 37% to 63% of the time, based on the authenticated encryption mode chosen. Having the encrypted message sent together with the MAC of the original message, since the two operations are done in parallel with some extent, is faster by about 16-17%. The Encrypt-then-MAC and the MAC-then-Encrypt modes take about the same time.

4.1.2 Encryption mode

Figure 4.4 shows the difference in performance between the two encryption modes available on the Secure Hardware Extension module: ECB and CBC. The tests were conducted with the transformer configuration all set to default except for the encryption mode, so the authenticated encryption mode used is Encrypt-then-MAC.

In the figure we see the worst case of the time to elaborate a message (among 100 messages) of two different lengths: 16 bytes on the left and 200 bytes on the right.

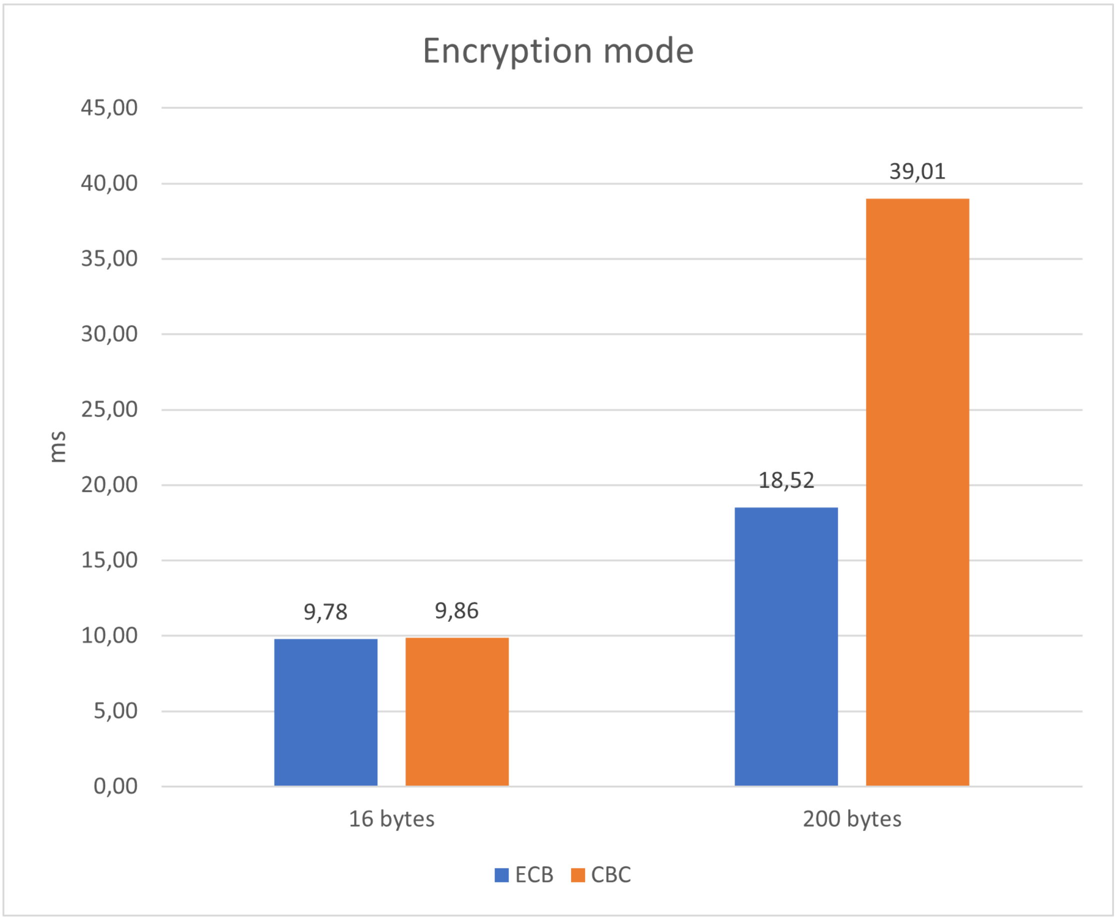


Figure 4.4: Encrypting 100 messages with different encryption modes

The first length was chosen as it is the length of a single block of the encryption algorithm. For the second length it was considered the use case of communicating a fault while also telling the coordinates and direction of the vehicle: communicating coordinates for up to 1cm precision and the direction of movement as an angle takes 24 bytes, the speed fits in one byte and 100 bytes were considered for the fault message: the total is 125 bytes, so 200 bytes were used for good measure.

The results on the left show that for one single block of data the two encryption modes take almost the same time (CBC takes a very little longer because it needs to fetch the initialization vector), so the previously shown results are unaffected by the encryption mode.

The columns on the right show that, for longer messages, CBC takes more than double the time compared to ECB, and for ECB a message which is 12,5 times bigger just takes less than 90% more time to be elaborated.

4.1.3 Performance loss in the long run

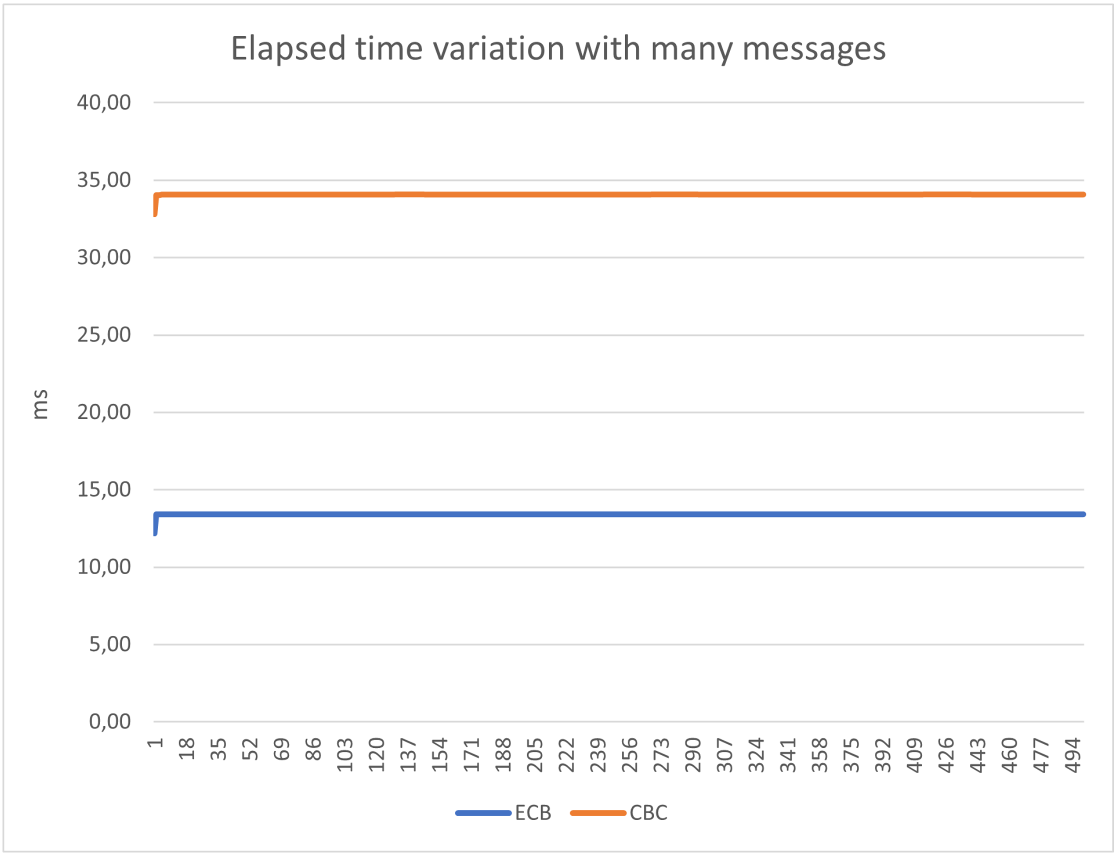


Figure 4.5: Encrypting many messages with different encryption modes

The tests whose results are shown in Figure 4.5 were conducted to see if the performance were affected by increasing the number of messages elaborated one after another by the transformer, in both encryption modes. As we can see well by the graph, there is no performance loss whatsoever, hence the previous tests conducted by elaborating 100 messages wouldn't have been different if conducted by elaborating more.

4.2 Tests on the MQTT protocol

The aim of these tests was to show actually how fast is the MQTT protocol, since it was chosen mainly for its speed: in particular for the speed of the connection which, as we saw in the first chapter, is fast because it just requires a single message of very small dimension.

The tests were conducted by making the MQTT ECU send 100 messages to an MQTT Broker, which forwarded them to an MQTT Client subscribed to the same topic, and calculating the difference between the timestamps of two successive messages: this way we could show if the protocol was fast enough for letting the ECU communicate its position 100 times in the same geographical area, which is of course an exaggerated number.

The dimension considered for the area, for test purposes, is the range of a 5G Pico Small Cell, which is at worst 100m [92]. The speed of the car considered is 150 km/h, which is the highest speed limit possible in Italy [93] (it could be allowed in certain conditions on determined highways, but normally the highest is 130 km/h).

These last two test conditions can't ever be seen in the real world, as cells with such small ranges would never be deployed on high-speed highways, but for the sake of testing we shall consider the worst case of all possible parameters.

4.2.1 Results

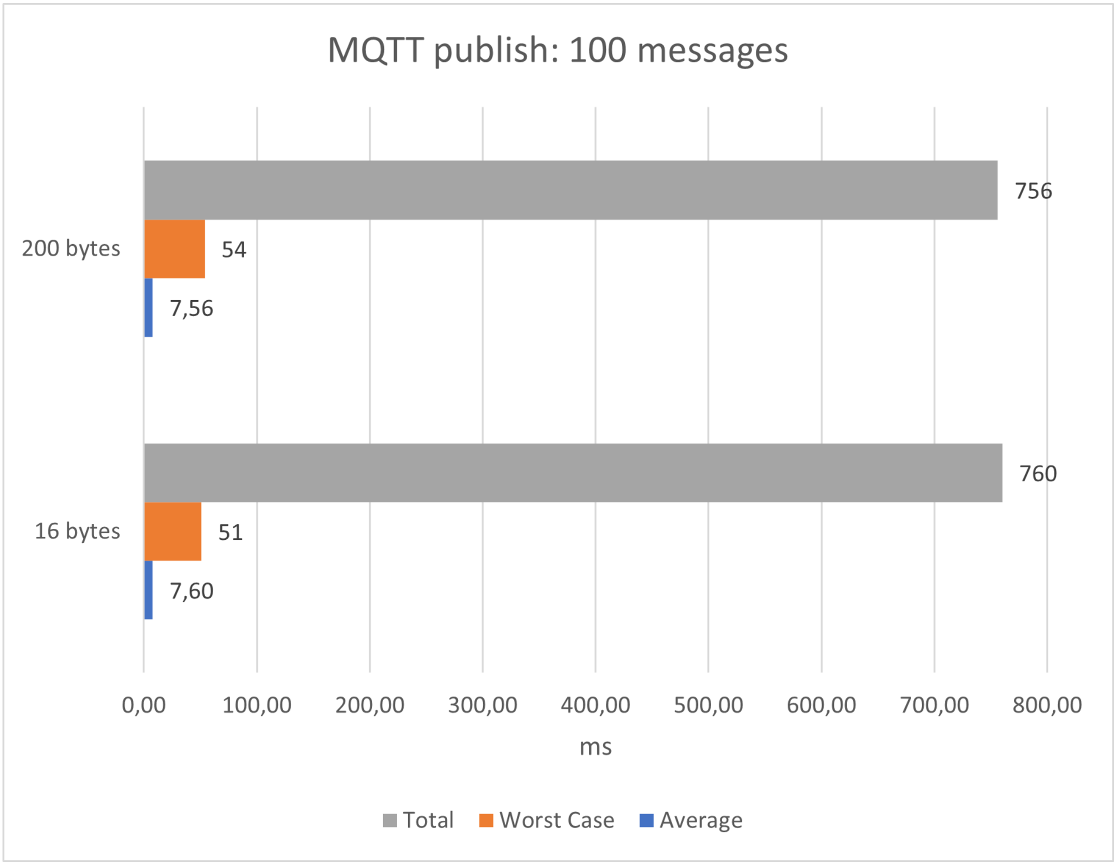


Figure 4.6: Sending many messages over MQTT

The tests were again conducted first by sending messages of 16 bytes initial dimension and then of 200 bytes, for the same reasons of the previous tests: we can see from the diagram in Figure 4.6 that the dimension of the message didn't matter at all as we got basically the same results.

The first message was the one that took the longest in both tries, as it included the time taken by the MQTT connection: the connection time was further tested, and it takes about 40-50 milliseconds. Other than the first message, most other messages took less than 4ms, but there were a few outliers (less than 8% of the messages) that took between 10 and 30 ms: this was later discovered to be because the client was set to probe the connection (by pinging the broker) every few messages. This could have probably been removed, but it didn't affect our results so we left them as they were.

Figure 4.6 shows that in the worst case it took 760ms to send 100 messages. In 760ms a car travelling at 150 km/h travels 31,66m, which is less than a third of the considered reference range.

Chapter 5

Conclusions

Communications between cars ECUs and external networks can have endless applications, and it won't be long until they will be used regularly without the driver even noticing. But, as we saw in these pages, designing an architecture for such communications is very challenging: not only there needs to be found a good compromise between performance and security, as in every other scenario, but the hardware that composes cars ECUs isn't as powerful and capable as to make this search for a compromise simple.

Nonetheless, the architecture just presented was proven to be worthy of such a task: the speed of the communication, thanks to the use of the fast and lightweight MQTT protocol, was more than enough to uphold the requisites of our use cases. And while it may not be up to the highest of security standards, for sure it is secure enough to protect short-term data in high-speed environments.

The use of an AUTOSAR transformer, moreover, accomplishes two things: it facilitates application and Runtime Environment developers to make a message ready for being transmitted and, because custom transformers are AUTOSAR Complex Device Drivers, different car manufacturers can just integrate it as a library; this way it is possible for cars from different makers to communicate over the same architecture, which is very important in use cases that benefit from complete coverage of the vehicles on the road.

We will now have a look at how the presented architecture could be improved, and we will see some other points of view for approaching the problem.

5.1 Possible improvements of the architecture

From a performances point of view there is little room for improvement given the technical limitations of the hardware on which the architecture was developed.

However, higher performances could be achieved by having the transformer run on a dedicated ECU suitable for cryptographic operations (e.g. with an hardware cryptographic accelerator).

An important improvement would be to use a different encryption mode, most notably Galois/Counter Mode (GCM, introduced in the first chapter) which is safer than ECB but it's able to be computed in parallel, unlike CBC. It is not supported by the Secure Hardware Extension, so it should be provided to the Crypto Stack by a separated hardware module, an external library or a dedicated driver. The same goes for other MAC calculation techniques.

Security-wise there's more margin for improvements. First of all, we are sending secure messages over an non-secure channel, which is acceptable in some cases but securing the channel could definitely be important for more security demanding situations (for example, the highway toll payment use case). Moreover, it could be implemented some system for client and/or server authentication, both for the connection with the broker and for the consumer clients (or anyway, if we are changing the communication protocol, with all the entities involve). These, and other possible improvements on the security, must of course come together with an improvement of the hardware, considering that SSL/TLS was too heavy for our board.

5.2 Further developments

This thesis presented a complete architecture for the communication between cars and external networks: from when a message is generated (i.e.: collected by a sensor) to its delivery to the consumer.

We also had a look at different use cases of how this architecture could be employed in the real world, and we saw that they are not mutually exclusive and can be part of the same infrastructure of connected vehicles. We saw an example of the entities that could be involved for the management of the road traffic (e.g. the gateway that delivers the MQTT topic), and what messages those entities could be exchanging.

In this direction, a further work would be to actually design in detail and develop such an architecture, building on top of the one here presented to expand to the other entities proposed. This presents a different domain of challenges compared to the ones dealt with by the architecture proposed by this thesis, and potentially contains different projects at different layers.

Another problem that we didn't deal with, as it must be addressed at a different level (possibly by the RTE or even the OS), is the handling of messages from the recipient vehicle point of view, particularly hard in cases of high-traffic areas. Having to elaborate a large quantity of data can also lead to the problem of the data being too old and so not relevant anymore (e.g.: the car moved).

his could be a problem of queueing the messages or of task allocation but could also be resolved by not having the external entities communicate to cars all the info from every single other vehicle and somehow grouping the intel. This of course depends on the actual design of the whole architecture, so it's successive to the previous point if dealt with in this way.

Lastly, the whole architecture could be moved to the AUTOSAR Adaptive Platform, which provides two potentially useful modules: the Network Management module and the Communication Management module. The Adaptive Platform isn't as widespread as the Classic Platform, in fact the board used only comes with Classic Platform modules, but it has several advantages: most notably it can be deployed on containers and hypervisors other than on actual microcontrollers, which makes testing a solution very much easier.

Bibliography

[1] Hiroaki Takada. *Introduction to Automotive Embedded Systems.* Nagoya University. June 2012. URL: `https://cse.buffalo.edu/~bina/cse321/fall2015/Automotive-embedded-systems.pdf` (cit. on pp. 1, 4, 38).

[2] *Electronic control unit.* Wikipedia. URL: `https://en.wikipedia.org/wiki/Electronic_control_unit` (cit. on p. 3).

[3] Robert N. Charette. «This Car Runs on Code». In: *IEEE Spectrum* (Feb. 2009). URL: `https://spectrum.ieee.org/this-car-runs-on-code` (cit. on p. 3).

[4] Christoph Hammerschmidt. «Number of automotive ECUs continues to rise». In: *eeNews Automotive* (Mar. 2019). URL: `https://www.eenewsautomotive.com/en/number-of-automotive-ecus-continues-to-rise/` (cit. on p. 3).

[5] Karim Nice. «How Car Computers Work». In: *HowStuffWorks* (2001). URL: `https://auto.howstuffworks.com/under-the-hood/trends-innovations/car-computer1.htm` (cit. on p. 4).

[6] *AUTOSAR Introduction - Part 1.* EXP. AUTOSAR. July 2021 (cit. on p. 5).

[7] Heiko Weber et al. *The 2019 Strategy& Digital Auto Report. Time to get real: opportunities in a transforming market.* Tech. rep. Strategy&, 2019. URL: `https://www.strategyand.pwc.com/de/en/industries/automotive/digital-auto-report-2019/digital-auto-report-2019.pdf` (cit. on p. 5).

[8] *AUTOSAR.* Wikipedia. URL: `https://en.wikipedia.org/wiki/AUTOSAR` (cit. on pp. 5, 6).

[9] *OSEK.* PiEmbSysTech. URL: `https://piembsystech.com/osek/` (cit. on p. 5).

[10] *AUTOSAR Introduction - Part 2.* EXP. AUTOSAR. July 2021 (cit. on p. 6).

[11] *Layered Software Architecture.* EXP. AUTOSAR. Oct. 2018 (cit. on pp. 6, 8, 16, 34).

[12] *Specification of RTE Software*. SWS. AUTOSAR. Nov. 2020 (cit. on pp. 9, 12).

[13] *Virtual Functional Bus*. EXP. AUTOSAR. Nov. 2018 (cit. on p. 10).

[14] *General Specification of Transformers*. ASWS. AUTOSAR. Nov. 2020 (cit. on pp. 10, 12, 39).

[15] *Automotive Cybersecurity Talk. Secure Boot - What You Need to Know*. Buzzsprout, EScrypt. Apr. 2021. URL: `https://automotivecybersecuritytalk.buzzsprout.com/1741159/8249753-secure-boot-what-you-need-to-know` (cit. on p. 14).

[16] *Specification of Secure Hardware Extensions*. TR. AUTOSAR. Nov. 2019 (cit. on p. 14).

[17] *Utilization of Crypto Services*. EXP. AUTOSAR. Oct. 2018 (cit. on p. 15).

[18] *Specification of SW-C End-to-End Communication Protection Library*. SWS. AUTOSAR. Dec. 2017 (cit. on p. 18).

[19] *E2E Protocol Specification*. PRS. AUTOSAR. Nov. 2020 (cit. on p. 19).

[20] *Specification of Module E2E Transformer*. SWS. AUTOSAR. Nov. 2019 (cit. on p. 19).

[21] *MQTT V3.1 Protocol Specification*. IBM, Eurotech. Aug. 2010. URL: `https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf` (cit. on p. 19).

[22] *ISO/IEC 20922:2016*. ISO/IEC. June 2016. URL: `https://www.iso.org/standard/69466.html` (cit. on p. 19).

[23] Steve Cope. «Beginners Guide To The MQTT Protocol». In: *Steve's Internet Guide* (Feb. 2021). URL: `http://www.steves-internet-guide.com/mqtt/` (cit. on p. 20).

[24] The HiveMQ Team. *MQTT & MQTT 5 Essentials. A comprehensive overview of MQTT facts and features for beginners and experts alike*. 2020. URL: `https://www.hivemq.com/download-mqtt-ebook/` (cit. on pp. 20, 21).

[25] The HiveMQ Team. *Getting Started with MQTT*. HiveMQ. Apr. 2020. URL: `https://www.hivemq.com/blog/how-to-get-started-with-mqtt/` (cit. on p. 21).

[26] *MQTT: The Standard for IoT Messaging*. OASIS, MQTT Technical Committee. URL: `https://mqtt.org/` (cit. on p. 21).

[27] Peter R. Egli. *MQTT MQ TELEMETRY TRANSPORT. AN INTRODUCTION TO MQTT, A PROTOCOL FOR M2M AND IoT APPLICATIONS*. Indigoo. 2016. URL: `https://www.indigoo.com/dox/wsmw/1_Middleware/MQTT.pdf` (cit. on p. 21).

[28] Steve Cope. «Understanding the MQTT Protocol Packet Structure». In: *Steve's Internet Guide* (Jan. 2021). URL: http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/ (cit. on p. 22).

[29] Steve Cope. «Introduction to MQTT-SN (MQTT for Sensor Networks)». In: *Steve's Internet Guide* (July 2021). URL: http://www.steves-internet-guide.com/mqtt-sn/ (cit. on p. 23).

[30] mqtt.org Community. *mqtt.org Wiki - libraries*. Github. URL: https://github.com/mqtt/mqtt.org/wiki/libraries (cit. on p. 23).

[31] *Eclipse Paho*. Eclipse Foundation. URL: https://www.eclipse.org/paho/ (cit. on p. 23).

[32] *Eclipse Mosquitto. An open source MQTT broker*. Eclipse Foundation. URL: https://mosquitto.org/ (cit. on p. 23).

[33] Erik Moqvist. *MQTT Tools. MQTT version 5.0 client and broker using asyncio*. Github. URL: https://github.com/eerimoq/mqttools (cit. on p. 23).

[34] *Comparison of MQTT implementations*. Wikipedia. URL: https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations (cit. on p. 23).

[35] *CSRS Glossary - encryption*. NIST. URL: https://csrc.nist.gov/glossary/term/encryption (cit. on p. 24).

[36] «What is encryption? Types of encryption». In: *Learning SSL* (). URL: https://www.cloudflare.com/it-it/learning/ssl/what-is-encryption/ (cit. on p. 24).

[37] Gary C Kessler. *An Overview of Cryptography*. Jan. 2022. URL: https://www.garykessler.net/library/crypto.html (cit. on p. 24).

[38] Christof Paar and Jan Pelzl. *Understanding Cryptography. A Textbook for Students and Practitioners*. 2010 (cit. on p. 25).

[39] *RC4*. Wikipedia. URL: https://it.wikipedia.org/wiki/RC4 (cit. on p. 25).

[40] *DATA ENCRYPTION STANDARD (DES)*. URL: https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf (cit. on p. 25).

[41] *Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard*. Jan. 1997. URL: https://csrc.nist.gov/news/1997/announcing-development-of-fips-for-advanced-encryp (cit. on p. 25).

[42] Philip Bulman. «Commerce Department Announces Winner of Global Information Security Competition». In: (Oct. 2000). URL: https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security (cit. on p. 25).

[43] John Schwartz. «TECHNOLOGY; U.S. Selects a New Encryption Technique». In: *The New York Times* (Oct. 2000). URL: https://www.nytimes.com/2000/10/03/business/technology-us-selects-a-new-encryption-technique.html (cit. on p. 25).

[44] Alfred J. Menezes et al. *Handbook of applied cryptography.* 1997 (cit. on p. 25).

[45] *ISO/IEC 10116:2006. Information technology — Security techniques — Modes of operation for an n-bit block cipher.* ISO/IEC. Feb. 2006. URL: https://www.iso.org/standard/38761.html (cit. on p. 25).

[46] Morris Dworkin. «Block Cipher Techniques». In: (Jan. 2017). URL: https://csrc.nist.gov/projects/block-cipher-techniques/bcm/modes-development (cit. on p. 25).

[47] *Block cipher mode of operation.* Wikipedia. URL: https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation (cit. on p. 26).

[48] David Wagner Helger Lipmaa Phillip Rogaway. *Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption.* URL: https://web.archive.org/web/20170126135324/http://csrc.nist.gov/groups/ST/toolkit/BCM//documents/proposedmodes/ctr/ctr-spec.pdf (cit. on p. 26).

[49] R. Housley. *RFC 5652. Cryptographic Message Syntax (CMS).* Sept. 2009. URL: https://datatracker.ietf.org/doc/html/rfc5652#section-6.3 (cit. on p. 26).

[50] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode.* Oct. 2010. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a-add.pdf (cit. on p. 26).

[51] *Message authentication.* Wikipedia. URL: https://en.wikipedia.org/wiki/Message_authentication (cit. on p. 27).

[52] «Data Origin Authentication». In: (July 2010). URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648434(v=pandp.10)?redirectedfrom=MSDN (cit. on p. 27).

[53] H. Krawczyk et al. *RFC 2104. HMAC: Keyed-Hashing for Message Authentication.* Feb. 1997. URL: https://datatracker.ietf.org/doc/html/rfc2104#section-2 (cit. on p. 27).

[54] Mihir Bellare. *AUTHENTICATED ENCRYPTION*. URL: `https://cseweb.ucsd.edu/~mihir/cse207/slides/s-ae.pdf` (cit. on p. 27).

[55] *ISO/IEC 19772:2009. Information technology — Security techniques — Authenticated encryption*. ISO/IEC. Feb. 2009. URL: `https://www.iso.org/standard/46345.html` (cit. on p. 27).

[56] *Authenticated encryption*. Wikipedia. URL: `https://en.wikipedia.org/wiki/Authenticated_encryption` (cit. on p. 27).

[57] E. Rescorla et al. *RFC 5746. Transport Layer Security (TLS) Renegotiation Indication Extension*. Feb. 2010. URL: `https://datatracker.ietf.org/doc/html/rfc5746` (cit. on p. 29).

[58] Domenico Junior Alessi. «Master's Degree thesis». MA thesis. Politecnico di Torino, 2022 (cit. on pp. 30, 32).

[59] Mansour Ahmadian et al. *MODEL BASED DESIGN AND SDR*. URL: `http://ftp2.sundance.com/Pub/documentation/pdf-files/DSPeRPaper.pdf` (cit. on p. 31).

[60] Michael Carone. *What Is Simulink?* MathWorks. URL: `https://uk.mathworks.com/videos/simulink-overview-61216.html` (cit. on p. 31).

[61] *AUTOSAR Blockset. Design and simulate AUTOSAR software*. MathWorks. URL: `https://uk.mathworks.com/products/autosar.html` (cit. on p. 31).

[62] Muhammad Ibrahim. «Research and mitigate the available open source solutions for the AUTOSAR standard with implementations of basic functionalities of the communication stack». MA thesis. Politecnico di Torino, 2022 (cit. on p. 32).

[63] *ISO 11898-1:2015. Road vehicles – Controller area network (CAN)*. ISO. Dec. 2015. URL: `https://web.archive.org/web/20170801153708/https:/www.iso.org/standard/63648.html` (cit. on p. 33).

[64] *CAN Wiki - Main Entry*. CAN Wiki. URL: `http://www.can-wiki.info/doku.php?id=start` (cit. on p. 33).

[65] Marco Guardigli. «Hacking Your Car». In: *Only Dead Fish Go With The Flow* (Oct. 2010). URL: `https://marco.guardigli.it/2010/10/hacking-your-car.html` (cit. on p. 33).

[66] *The CAN Bus Protocol*. Kvaser. URL: `https://www.kvaser.com/about-can/the-can-protocol/` (cit. on p. 33).

[67] *CAN (Controller Area Network) protocol*. Java T Point. URL: `https://www.javatpoint.com/can-protocol` (cit. on p. 33).

[68] *CAN FD - The basic idea*. CAN in Automation (CiA). URL: `https://www.can-cia.org/can-knowledge/can/can-fd/` (cit. on p. 33).

[69] *Requirements on BSW Modules for SAE J1939*. SRS. AUTOSAR. Nov. 2021 (cit. on p. 35).

[70] *Requirements on Secure Onboard Communication*. SRS. AUTOSAR. Nov. 2020 (cit. on p. 36).

[71] *Specification of Secure Onboard Communication*. SWS. AUTOSAR. Nov. 2020 (cit. on p. 36).

[72] Franco Orberti, Ernesto Sanchez, Alessandro Savino, Filippo Parisi, and Stefano di Carlo. *TAURUM P2T: Advanced Secure CAN-FD Architecture for Road Vehicle*. Control and Computer Eng. Dep. of Politecnico di Torino, PUNCH Torino S.p.A., July 2021 (cit. on p. 37).

[73] *Car2X in the new Golf: a technological milestone*. Volkswagen. Mar. 2020. URL: `https://www.volkswagen-newsroom.com/en/stories/car2x-in-the-new-golf-a-technological-milestone-5919` (cit. on pp. 38, 51).

[74] *Car2X & C-V2X – connected vehicle pilot projects from the US*. Audi. June 2021. URL: `https://www.audi.com/en/innovation/autonomous-driving/car-to-x.html` (cit. on p. 38).

[75] Darius Sveikauskas. *Security By Design Principles According To OWASP*. Patchstack. June 2021. URL: `https://patchstack.com/security-design-principles-owasp/` (cit. on p. 42).

[76] *S32K148-Q176 General Purpose Evaluation Board*. NXP. URL: `https://www.nxp.com/design/development-boards/automotive-development-platforms/s32k-mcu-platforms/s32k148-q176-general-purpose-evaluation-board:S32K148EVB` (cit. on p. 42).

[77] *ISO/SAE 21434:2021. Road vehicles — Cybersecurity engineering*. ISO. Aug. 2021. URL: `https://www.iso.org/standard/70918.html` (cit. on p. 42).

[78] *Classic AUTOSAR with EB tresos*. Elektrobit. URL: `https://www.elektrobit.com/products/ecu/eb-tresos/` (cit. on p. 42).

[79] *S32 Design Studio IDE*. NXP. URL: `https://www.nxp.com/design/software/development-software/s32-design-studio-ide:S32-DESIGN-STUDIO-IDE` (cit. on p. 42).

[80] *ISO 3779:2009. Road vehicles — Vehicle identification number (VIN) — Content and structure*. ISO. Oct. 2009. URL: `https://www.iso.org/standard/52200.html` (cit. on p. 44).

[81] *Arduino IDE 1.8.19*. Arduino. URL: `https://www.arduino.cc/en/software` (cit. on p. 48).

[82] *ESP8266 Series of Modules*. Espressif. URL: `https://www.espressif.com/en/products/modules/esp8266` (cit. on p. 48).

[83] Ivan Grokhotkov. *ESP8266WiFi library*. URL: `https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/readme.html` (cit. on p. 48).

[84] Monstrenyatko. *ArduinoMqtt*. URL: `https://github.com/monstrenyatko/ArduinoMqtt` (cit. on p. 48).

[85] *Swarm Intelligence: The "Car-to-X" principle*. Audi MediaCenter. Aug. 2019. URL: `https://www.audi-mediacenter.com/en/audi-at-the-2019-ces-11175/swarm-intelligence-the-car-to-x-principle-11182` (cit. on p. 50).

[86] *Autostrade: Pedaggio e Telepedaggio*. Telepass. URL: `https://www.telepass.com/it/privati/servizi/autostrada` (cit. on p. 52).

[87] *Cómo funciona - Via-t Sistema de Telepeaje*. Via-t. URL: `https://www.viat.es/funcionamiento/como-funciona` (cit. on p. 52).

[88] *Plus codes*. Google. URL: `https://maps.google.com/pluscodes/` (cit. on p. 52).

[89] *what3words*. URL: `https://what3words.com/about` (cit. on p. 52).

[90] Alessandro Di Vincenzo. «Analysis of MQTT protocol security and a method for key distribution through separation of knowledge». MA thesis. Politecnico di Torino, 2022 (cit. on p. 53).

[91] *Add your car key to Apple Wallet on your iPhone or Apple Watch*. Apple. Feb. 2022. URL: `https://support.apple.com/en-gb/HT211234` (cit. on p. 54).

[92] *5G Small Cell Testing*. LitePoint. URL: `https://www.litepoint.com/5g-small-cell/` (cit. on p. 63).

[93] *Art. 142. Limiti di velocità*. Codice della strada. Repubblica Italiana (cit. on p. 63).

Acknowledgements

First of all, I shall express my deepest gratitude to the three people that helped me the most during the writing of this thesis.

I would like to thank Alessandro Di Vincenzo because we shared our moments of joy and pain (mostly pain), he supported me in the hardest of times and the past months would have been very much tougher without him.

I thank Jacopo Federici because he has spent a lot of time for me, he guided me more than I could have ever wished for and he has been a lighthouse to look at when I wasn't sure of where to go.

Thanks to Giulio Muscarello, not only for the practical help but most importantly for supporting me, for understanding my despair and for having my back when I was overwhelmed by this project and I could not deal with other problems.

I would also like to extend my thanks to my supervisor, prof. Ernesto Sanchez, for being very nice and not intrusive and for having handed some precious advice when most needed.

I am also very grateful to all the people at Brain Technologies and the other interns who I worked with.

But this thesis is just the final piece of the puzzle that has been my journey throughout the last two years, which have been the most challenging of my life so far. So I feel it is appropriate to speak my appreciation for all the people that have been by my side during this time.

A huge thanks to all my friends for... being my friends: for helping me when I needed it, for cheering me up when I felt down, for making me forget my troubles even for just a second. I often take my friends for granted, but just knowing that they are there and that I can rely on them whenever I need makes my life much more lighter. For every day, every instant, every moment that I'm living, thanks a lot.

In particular I thank my friends from the Politecnico, as they have been essential for surviving during these two years of remote lessons and exams. And I thank my friends back in Napoli, that made me feel not as far from home as I was.

Thanks to my fellows whom I have fought with shoulder to shoulder, for giving my life a purpose that was not just about myself, and for making me remember that as big as my problems seem to me, there are always bigger problems out here.

Last but not least I thank to my family, without which any of this simply could not have been possible. They supported me and my choices, they let me go when the time was right, and most importantly they made me feel that no matter how hard the things could have been, I could always count on them and they always had my back. I very rarely show them gratitude and these few words could never be enough, but I am always deeply grateful to them even when I am not disclosing it.