# POLITECNICO DI TORINO

**Master's Degree in Mechatronic Engineering**


**Master's Degree Thesis**

# Implementation of a PKI-based security communication and Value Added Service for EV charging using ISO 15118 standard

**Supervisors**

**Prof. Marcello CHIABERGE**

**Eng. Alberto SEVEGA**

**Candidate**

**Filippo D'AGOSTINO**

**April 2023**

*This research paper ("Thesis") has been produced in the context of an internship program that Filippo D'Agostino (the "Intern") has attended within Accenture Spa. The Thesis is confidential and cannot be used other than by the Intern for the purpose of presenting his/her work. It must not be disclosed other than to Politecnico di Torino's professors on a confidential basis and to the extent required for Politecnico di Torino to carry out an evaluation and assess the Intern's final exam. In preparing this document, the Intern acknowledges that he/she has relied on confidential information provided by Accenture only for the purposes of producing the Thesis. The Intern shall defend, indemnify and hold harmless Accenture from and against any claim, demands, actions, judgments, awards, settlements, fees, liabilities, losses, damages, costs and expenses (including and without limitation attorneys' fees and court costs) ("Obligations") arising out of or relating to any allegation or claim that the Thesis or any use thereof infringes, misappropriates or otherwise misuses or violates the Intellectual Property or other rights of any Person. Accenture shall have the right, in its sole discretion, to participate in the defense of any such allegations or claims at its expense with a counsel of its choosing. The Intern shall not compromise or settle any such allegation or claim, or agree to binding arbitration thereof, in any manner without Accenture's prior written consent, unless such settlement is solely monetary in nature, and releases all Accenture Indemnified Parties from all Obligations with no admission of liability and has no adverse effect on any of them. This paper was written for my university thesis and is my personal work. The views expressed are my own and may not necessarily reflect those of Accenture.*

# Summary

Nowadays, the world is facing an energy transition phase towards full decarbonization and a 100 percent energetic sustainable situation. One of the big steps, that could be crucial for the ecological changeover, is involving the vehicle passage from ICE to electric vehicles. In this field, technological progress is moving towards the V2G technology, a method that enables the bidirectional electrical power flow at the charging station.

This thesis aims to implement the digital communication between an Electric Vehicle (EV) and a Charging Point as a Proof-of-Concept in a full-simulated charging process in order to lay the foundations for the V2G technology to work smoothly. In particular, it will focus on the digital security field; thus, it will take care of identification, protection and authentication of the data transfer.
The communication itself is ruled by the ISO 15118 standard which has been applied, in this case, using a Java stack protocol called Rise-V2G. The simulation environment has been built using 2 Raspberry Pi 4 on which are developed a series of Python scripts to manage the data flow. All the communication necessary data are shown on 2 touchscreen displays.
On the digital security side, it requires to integrate a PKI-based protocol that is normalized by the already quoted ISO standard and the VDE-AR-E 2802-100-1 guide. The latter provides the application of an asymmetric encryption and a 2-way digital certificate identity check for all Entities involved in the charging

process. Besides the generation of customized certificates, the 2 main objectives are the usage of the installation and update of new certificates. This represents a key role since it allows the Plug and Charge mode, which differs from the modern identification method, through RFID, by simply plugging in the connector.
A further implementation that has been developed in this project is the execution of a Value Added Service (VAS) during the charging process. This process has been developed in the stack Java script, as regarding the description and for its possible selection by the user, and in a Python script for its real execution.

# Acknowledgements

I would like to thank my supervisors Eng. Alberto Sevega and Prof. Marcello Chiaberge for the opportunity to collaborate with Accenture Spa and take part in this innovative project.
I also thank my former supervisor Ph.D. Massimo Reineri for supporting me during the first half of this experience.

*If at first you don't succeed, laugh until you do...*

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**BEV**

Battery Electric Vehicle

**BL**

Business Logic

**CCB**

Contract Certificate Bundle

**CCP**

Contract Certificate Pool

**CPO**

Charging Point Operator

**CPS**

Certificate Provisioning Service

**CSMS**

Charging Station Management System

**CSR**

Certificate Signing Request

**EIM**

External Identification Mode

**eMAID**

eMobility Authentication Identifier

**eMSP**

eMobility Service Provider

**EV**

Electric Vehicle

**EVCC**

Electric Vehicle Communication Controller

**EVSE**

Electric Vehicle Supply Equipment

**EXI**

Efficient XML Interchange

**GUI**

Graphical User Interface

**HEV**

Hybrid Electric Vehicle

**ICE**

Internal Combustion Engine

**MO**

Mobility Operator

**OEM**

Original Equipment Manufacturer

**OSI**

Open System Interconnection

**PCID**

Provisioning Certificate ID

**PCP**

Provisioning Certificate Pool

**PFS**

Power Flow Simulator

**PHEV**

Plug-in Hybrid Electric Vehicle

**PKI**

Public Key Infrastructure

**PLC**

Power Line Communication

**PnC**

Plug and Charge

**PoC**

Proof-of-Concept

**SA**

Secondary Actor

**SECC**

Supply Equipment Communication Controller

**SoC**

State of Charge

**TCP**

Transmission Control Protocol

**TLS**

Transport Layer Security

**UDP**

User Datagram Protocol

**V2B**

Vehicle-To-Building

**V2G**

Vehicle-To-Grid

**V2GTP**

Vehicle-To-Grid Transfer Protocol

**V2H**

Vehicle-To-Home

**VAS**

Value Added Service

**XML**

eXtensible Markup Language

# Chapter 1

# Introduction

## 1.1 Evolution of EV market

The modern world scene is ruled by the research and development of a possible solution to the impending problems of climate change and energy crisis, and this scenario brings the current global automotive landscape into a deep changing phase. This is witnessed by the recent European Parliament's approval of the European Commission's proposal to ban the production of endothermic cars from 2035. It represents a strong change of course which is translating into a discreet increase in the electric vehicles market.

### 1.1.1 Historical evolution of the EV

The first appearance of an electric vehicle on the market is registered in the first half of the XIX century. Its production slightly increases till the next century, when EVs' diffusion peaks. In 1900-1910, EVs were the main means for short trips around the cities. However, the availability of electricity was only guaranteed in the areas with a high-density population. Moreover, in the same years, Ford's gas-powered model-T changed completely the focus of cars; in fact, the cars based on gasoline as

a power source were cheaper, and affordable for a larger population and they have much longer autonomy. Another reason that affected the EV disappearance was the development of a better system of roads and interconnections between cities.
Over the next 30 years, electric vehicles entered a sort of dark age with little advancement in this technology. Cheap and plentiful gasoline and the continued improvement of the internal combustion engine have negatively impacted the demand for alternative fuel vehicles.
In the early 1970s, an accretion of oil prices and gasoline shortages determined a push in exploring options for alternative fuel vehicles, including electric cars. Nevertheless, the vehicles developed and produced in the 1970s still suffered from drawbacks compared to gasoline-powered cars. EVs had limited performance, usually topping at speeds of 72 km/h, and their typical range was limited to 64 km before needing to be recharged.
The real revival of the electric vehicle took place at the beginning of the XXI century when its market took off due to various factors. Initially, it was 2 companies that started this trend: the newly established Tesla Motors in Silicon Valley and Toyota in Japan. In addition, many environmental incentives were created and new technological steps were taken. Some of these represented major improvements in battery performance, such as in charging infrastructures and charging times.
As these core vehicle-related technologies improved, new challenges quickly took their place, such as how to manage the energy balance, how to distribute and locate charging points across countries, how to make the charging process user-friendly, etc. Questions like these guaranteed a bright future for electric vehicle development.

### 1.1.2 Economical evolution of the EV market

Considering the EVs evolution from the economical point of view and how quantitatively the automotive market changes, I will focus on analyzing the data collected in the last 10 years since this is the period in which has been recorded the biggest

variation in terms of how much the main product, in our case the vehicles, is changed in favor of a second one with a different power source.

In the early 10s, the stock of electric vehicles was about 17000 BEV units and 400 PHEV units. After a decade, these figures increased exponentially, reaching 11 million for BEVs and 5.2 million for PHEV.

The growing trend of charging points can be observed; they increased from the value of 3700 units of slow CPs and 310 units of fast ones to account for over 1.2 million for the slow units and 560000 units for the fast ones.

The charts below show in more detail these 2 tendencies.

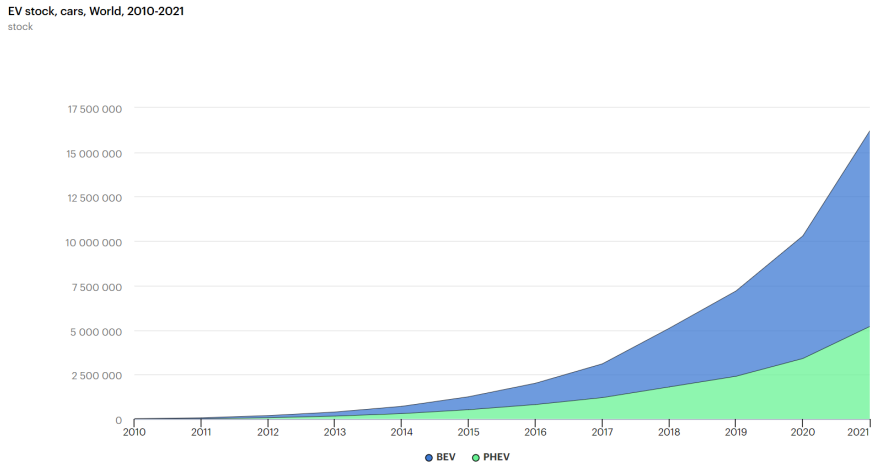


**Figure 1.1:** EVs stock global trend between 2010 and 2020.[1]

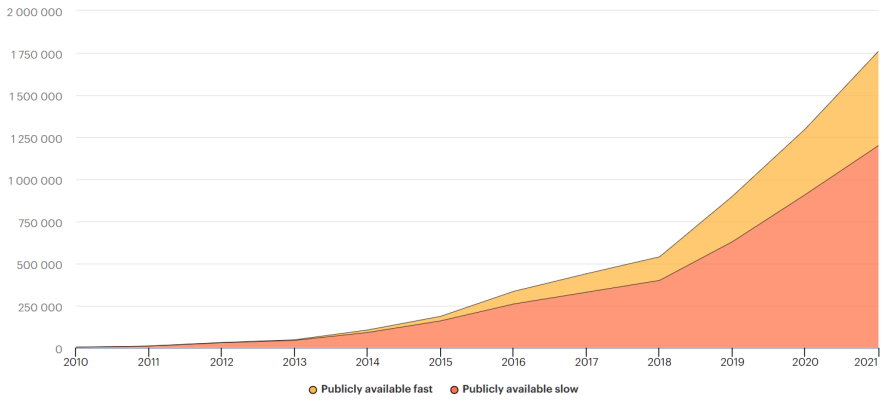


**Figure 1.2:** Charging points global trend between 2010 and 2020.[1]

At the same time, ICE vehicles have undergone an initially stronger increase than EVs one, but at the end of the decade, it almost stabilizes. The future of vehicles based on the combustion engine will not be characterized by a different trend; indeed, the predictions of ICE vehicles will fall by about 9.9 million units by 2030.
The following bar chart shows hypothetically how the ICE car global sales could change in the next 10 years and it compares these values to the ones registered in the last 5 years.

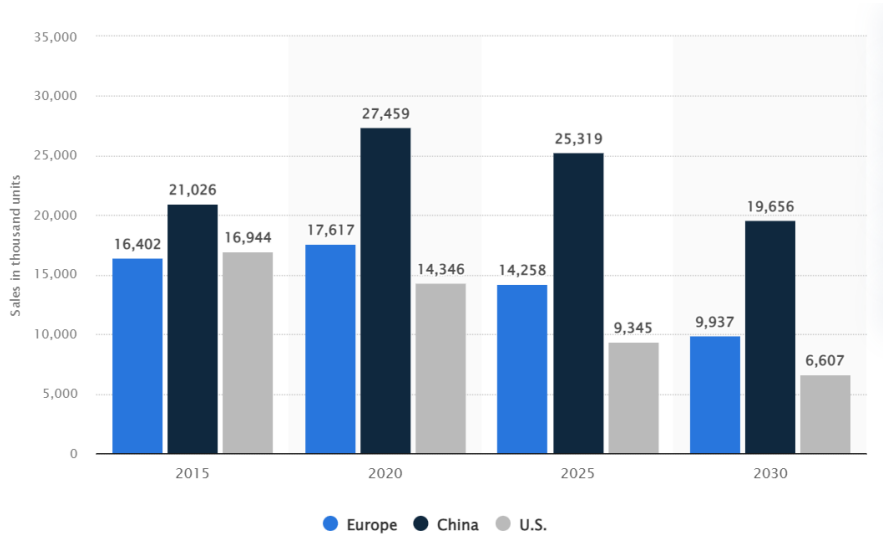


**Figure 1.3:** ICE vehicles global sales from 2015 and future predictions by 2030.

With regard to the predicted number of EV sales in the next 10 years, strong growth can be expected. In particular, the estimated value of EV sales by 2030 will be over 45 million units. This is a likely scenario based on the technological advancements achieved until now and the past and present vehicle market evolution. However, it should not be explained that electric motors and batteries will remain the leading power resource used in the transport field since the countries, hosting the world heading companies, are released incentives for the research.
Also, in this case, the IEA offers a chart accurately describing the future possible

progression of EV sales.

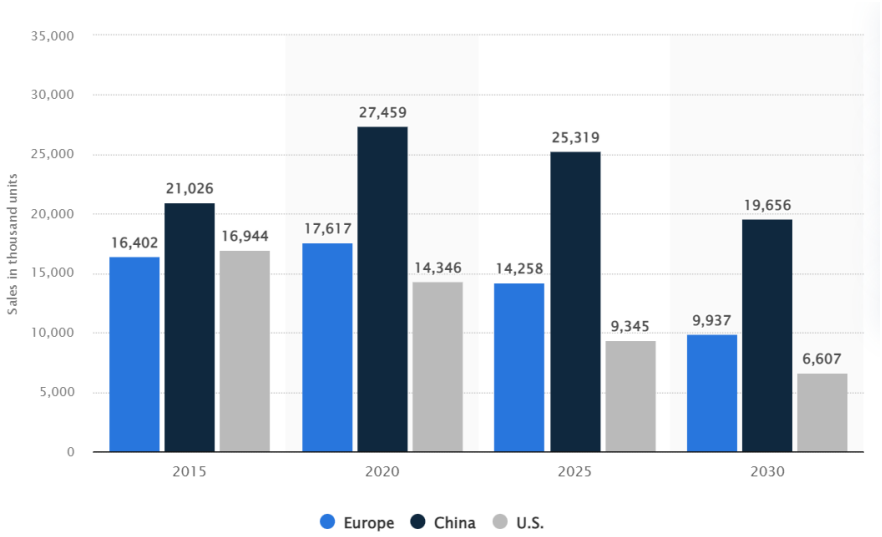


**Figure 1.4:** EV sales prediction by 2030.

## 1.2 EV world

As already mentioned, the world surrounding the electric vehicle is a technological front in constant evolution and it has not already reached its maximum potential. The modern EVs are spaced out a lot from their original prototype especially differentiating themselves under some specific structural or functional characteristics. This section aims to analyze and describe the common line that ties up all the categories and underline the different aspects for which they can be discriminated.

### 1.2.1 EV main families

There exist 4 macro categories [2] into which electric vehicles may be divided:

- **Battery Electric Vehicle (BEV)**:
  it is also called All-Electric Vehicle (AEV) and it relies exclusively on a battery and the electric drive train as power source and its transmission. It can operate in 2 modes:

  - direct mode, activated during the acceleration, simply provides the car's move using the energy from a big battery pack to one or multiple electric motors;
  - inverse mode, activated during deceleration when the brakes are pressed, the motor becomes an alternator converting the move into current, sending it back to recharge the battery.

- **Hybrid Electric Vehicle (HEV)**:
  it is also called a standard or parallel hybrid. Its power source is the cooperation between an ICE and an electric motor. Therefore, it has both a battery and a gasoline tank.
  The first main feature that distinguishes this type of e-vehicle is that the batteries cannot be charged through an external port using a power grid, but only through the motion of the wheels, through the ICE or a combination of both. The second distinguishing feature is the movement of the vehicle, which can be powered by its own 2 motors.

- **Plug-in Hybrid Electric Vehicle (PHEV)**:
  often called series hybrid, is a type of hybrid vehicle that has both ICE and electric motor. The difference with the HEV type set in the recharging process; the battery pack of the PHEV can be charged by an external port using a Charging Station.
  Typically PHEV can run in 2 modes:

  - all-electric mode, in which the motor and battery provide all the car's energy;
  - hybrid mode, in which both electricity and gasoline are employed.

  As an observation, we can notice that smaller engines can be mounted in the case of PHEV since the electric motor supplements the engine's power.

- **Fuel Cell Electric Vehicle (FCEV)**:
  also known as Fuel Cell Vehicle (FCV) or Zero Emission Vehicle, is characterized by the integration of a fuel cell to generate the electricity to run the vehicle.

### 1.2.2 Structure and main functional components of an EV

The main architectural components of an EV are dictated by the typology of the EV briefly explained in the previous subsection. However, the majority of the fundamental elements are actually the same for every modern EV. These machinery are listed and concisely described below.

- **Traction battery pack**:
  it is the most important and in some cases the only energy storage system in the form of direct current (DC).
  Actually, the battery pack is the macro element that provides energy, but it consists of smaller sub-elements. The basic unit is made up of individual batteries, also called cells, which are grouped into modules to save space so they can deliver a larger voltage. These modules complete the battery pack.
  Nowadays, many types of batteries can be used, and it is highly recommended to install a system to keep them under control; this type of system is called Battery Management System (BMS). It measures 3 essential parameters of the battery pack: current, voltage and temperature. It constantly compares these values with the safety limits and shuts down the load when they exceed the limits. Apart from safety purposes, the BMS is also used for some computational purposes; in particular, it measures 2 important values:

  - State of Charge (SOC)–>tells how far you can drive before recharge
  - State of Health (SOH)–>tells you when it's time to replace your batteries

  The other BMS aim is to maximize the life cycle of the battery pack.

- **Power Inverter**:
  electrical component with the function of modifying the current coming from the battery DC in AC so that it can drive the electric motor. In addition, it also has the function of converting the AC current into DC current during regenerative braking and then using it to charge the battery.

- **Controller**:
  essential element with the primary function of controlling energy flows and basic parameters useful for the correct car operation. As the brain of the vehicle, it controls the motor's speed and the DC voltage coming out of the battery pack. All its inputs come from the user like the throttle, breaks pressure, etc.

- **DC/DC Converter**:
  electronic element used to convert the battery pack voltage down to 12 volts such that the services, like infotainment, wipers, mirror controls, etc., can be made usable.

- **Charger**:
  allows charging processes from outside sources like grids. As a consequence of the different categories to which a car belongs, it is immediately noticed that not all vehicles own a charger. For example, all the HEV does not have it, but it is not valid the vice versa. The charger can be on-board if it is installed inside the vehicle, or off-board if it is installed in the Charging Station; this argument will be clarified in the next section.

- Many other components are shared in common between the EVs, but are less relevant for the scope of this thesis. Some of them are thermal systems (cooling), charge ports and mechanical transmission.

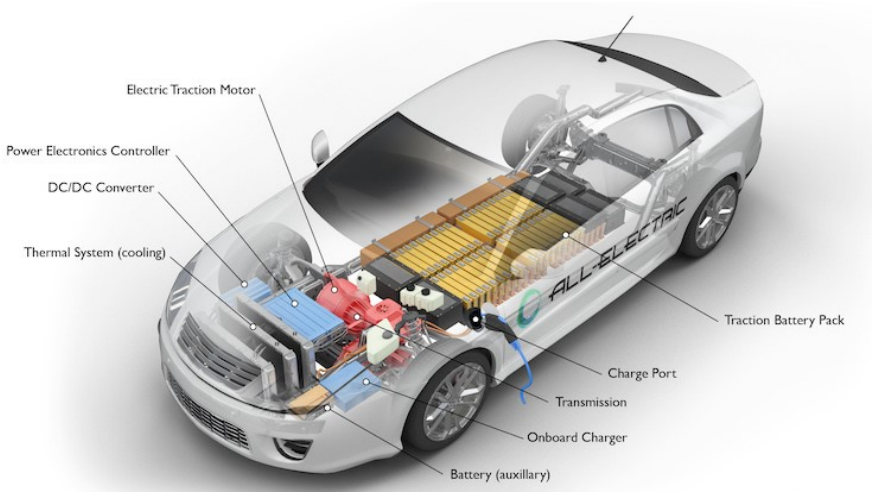


**Figure 1.5:** EV schema of the main components.

### 1.2.3 EVSE world

The charging process of an electric vehicle is performed by the Charging Station, also called Charging Point or Electric Vehicle Supply Equipment (EVSE). It is referred to the device that supplies the electrical energy required to charge the electric vehicle's battery and communicates with the vehicle to ensure an adequate and safe flow of electricity. Without going into too much negligible detail, a brief description of the principal components is provided below.

- The power electronics assembly is the heart of a charging station. Physically, it consists of wires, capacitors, transformers, and other electronic components. Functionally, it provides the power for the onboard or offboard charger for electric vehicles.

- The charge controller acts as the charge station's "street smarts". It oversees basic charging functions, such as turning the charger on/ off, measuring power consumption, and storing important real-time and event data.

- The network controller allows the station to communicate with its network via a built-in telecommunications device so managers can monitor it and review historical event data. It also controls user access to a charging station through a series of white (authorized) or black (unauthorized) lists.
- The cable and connector assembly plugs into the EV inlet using the EV coupler; this provides the physical interface EV-EVSE.

The development in the field of e-mobility not only led to a technological improvement of the vehicle itself, but also inevitably entailed the development of charging
On modern roads, the user of an electric vehicle has the possibility of charging the batteries by choosing between different modalities, which essentially depends on how much time he has available to charge the vehicle. These modalities are classified according to the amount of energy delivered to the vehicle. They are, of course, regulated and are designated differently depending on.

They are called Levels if they are defined by SAE J1772 and they count 3:

| | Type of Supply | Voltage(V) | Current(A) | Charging Time |
|---|---|---|---|---|
| Level 1<br>Slow Charging | Single-phase AC | 120 | 15-20 | 8-20 hours |
| Level 2<br>Quick Charging | Single/Three-phase AC | 208-240 | 20-50 | 3-8 hours |
| Level 3<br>Fast Charging | Single/Three-phase DC | 208-480 | <=125 | 15-30 mins |

**Table 1.1:** Charging levels illustrating the type of supply, voltage, current and the necessary charging time for each one.

Stage 1 has a charging load of 1.4- 1.9 kW and is most commonly used in the home, as it can be plugged into a standard household outlet.

Level 2 offers a charging power of up to 22 kW. It is mainly used in residential areas, parking zones, work and commercial areas and requires special household and public installation due to the high power supply.
Level 3 offers a charging load of 48 kW at 80 A and 400 kW at 400 A. This charging station is much larger than the others because it is designed to contain the AC /DC charger; in fact, it directly supplies DC power to the EV and is also therefore faster in the charging process. However, its production cost is also the highest of the 3. There is even a fourth neo-level (Level 4) which again works in DC and is only used for certain configurations and outlets like CHAdeMO.
In other words, we can distinguish furtherly the EV into 2 categories: AC-capable charging, which has the AC/DC charger on board, and DC-capable charging, which has it outside of the vehicle.
If they are defined by IEC 61851-1, they are called Modes counting to 4:

| | **Characteristics** |
|---|---|
| Mode 1 | AC charging, low current, no communication |
| Mode 2 | AC charging, for temporary solutions, using in cable control box on a standard household socket; communication by using Pulse Width Modulation (PWM) |
| Mode 3 | AC charging with additional features. Charging power can be controlled over high-level communication |
| Mode 4 | DC and AC charging with high-level communication. Off-board charger enables very fast charging speeds for DC charging |

**Table 1.2:** Charging modes illustrating each general characterization.

Despite the charging classes being called by different names, they can be seen as their own correspondents, as shown in the figure below.

| AC LEVEL 1 | | AC LEVEL 2 | DC LEVEL 3 (FAST CHARGER) |
|---|---|---|---|
| 120 AC, 1-phase, 12A or 16A max | | 208V-240V Ac, 1-phase, up to 80A max | 380V-600V Ac, 3-phase input DC output |
| **Mode 1 (AC)** | **Mode 2 (AC)** | **Mode 3 (AC)** | **Mode 4 (DC)** |
| • 250V AC, 1-phase, 16A max or 480V AC, 3-phase, 16A max<br>• Cord with no pilot auxiliary connections | • 250V AC, 1-phase, 32A max or 480V AC, 3-phase, 32A max<br>• Cord with control pilot & shock protection | • 250V AC, 1-phase, 32A max or 480V AC, 3-phase, 3-phase, 32A max<br>• Permanently connected to AC supply with control pilot & shock protection | • AC or DC input supply, cord or permanently connected, with control pilot & shock protection |
| • Delivers AC power from the wall socket to OBC<br>• Typically takes 8-12 hours to charge fully depleted battery | | • Delivers AC power from the electrical supply to the OBC<br>• Typically takes 4-6 hours to charge fully depleted battery | • Delivers DC power, bypassing the OBC<br>• Typically provides 80% charge of fully depleted battery within 30 minutes |

**Figure 1.6:** comparison between Charging Levels and Charging Modes.

EVSE software is designed to manage charging stations and their networks and should not be confused with EV applications designed to monitor the vehicle. EVSE network software promotes rapid setup and configuration of EV charging stations and facilitates a bi-directional data flow between the charging station and the cloud-based network control center. This functionality enables operators to remotely configure, manage and update charging point software, set up and control driver access to charging, set pricing, manage billing and generate usage reports. The software applications also allow drivers to easily find and reserve available charging stations. The software tools can also be configured to send notifications to operators (hardware/software issues) and e-drivers ("charging complete" and "charging station available")

The last EVSE component that needs a description is the plug. It changes form depending on the geographic region and the charging level.


**Figure 1.7:** plug types based on the region and the charging mode.

The 2 main connectors families are classified according to the charging level [3]:

- **AC Charging**:
  - AC Type 1–>designed for single-phase AC from 6 to 32 A and thus allows charging capacities of up to 7.4 kW.
  - AC Type 2–>developed for 3-phase AC, for charging battery EVs at 3–50 kW. A 1-phase charging process can also be carried out via the Type-2 connector.
- **DC Charging**:
  - DC GB/T–>mainly used in China, is based on the communication protocol defined by GB/T 27930.
  - CHAdeMO–>mainly used in Japan, uses CAN for the DC charging communication.

- Combined Charging System (CCS)–>extension for the AC Type-1 and Type-2 plug for high DC charging capacities, which additionally has 2 large power contacts.
  For high-level communication for smart charging, a powerline communication (PLC) is overlaid on the CP pin.
- Combo 1 (North America and Japan) and Combo 2 (Europe) connectors to provide power at up to 350 kW.

### 1.2.4 Smart charging

In recent years, technological progress in the electric vehicle ecosystem has moved strongly in a specific direction, such as batteries and charging systems. The latter is evolving in favor of the user. In other words, companies are trying to develop new services to make charging the vehicle faster and easier, and to meet customer needs with different services.
Thus, one of the latest and most important steps in the field of charging options is the so-called smart grid. It can basically be defined as an intelligent electrical network that combines electrical systems and smart digital communication technology. It is a self-sufficient electrical network system based on digital automation technology for monitoring, control, and analysis within the supply chain.
Essentially, for a grid to be called smart, it must provide 2-way communication between electric utilities and consumers. A smart grid is capable of controlling electrical energy at any single point on a power system and from multiple and widely distributed generation sources. This means that a smart grid must respond digitally to users' changing demand for electricity.
The consequence of this feature is a more balanced electrical power supply; in fact, smart grids allow for better efficiency in power transmission. Normally, problems with the power grid occur when there is an unbalanced factor. The smart grid helps to avoid these disturbances and keep the ratio
between supply and demand almost constant In addition, one of the biggest positive

effects of smart charging in today's world is helping to integrate renewable energy systems into the grid and manage them.

In addition to the benefits that smart charging brings to the energy network, it also affects the way operators can manage the digital network and maintain control over the parameters of the charging stations; thanks to the dense digital network, they can monitor the number of EVSE in operation, their current status and their individual events. They can also monitor the billing and payment process without having to intervene because the companies' goal is to achieve a higher level of system automation that minimizes both the operator's and the user's actions.

### 1.2.5 Overview on newest standard ruling the eMobility charging process communication

In the context of this eMobility panorama, the need arose to unify and, above all, regulate all the actors involved in the charging process of an electric vehicle. In particular, the communication between the Electric Vehicle Communication Controller (EVCC) and the EVSE controller, called Supply Equipment Communication Controller (SECC), has been regulated in a new document, along with all other standards: ISO 15118.
It is flanked by a second guide called VDE-AR-E 2802-100-1, which defines the roles of all the actors involved in the eMobility panorama and the measures for the specific area regarding the security protocols applied in the communication.

# Chapter 2

# Digital security guide for ISO 15118 implementation

## 2.1 Overview on Public key Infrastructure (PKI)

Public Key Infrastructure (PKI) [4] is a system of processes, technologies, and policies that allows you to encrypt, decrypt and sign data. It creates a secure connection for public web pages, private systems and other services that support MFA (Multi-Factor Authentication), a type of authentication method for which the user shall provide 2 or more verification factors to gain access to a resource. The majority of use cases in which PKI is applied are: web page security, files and emails/messages encryption, authentication and identification of VPN connections. However, its usage shall not be thought strictly related to web security, but it has great potential for the IoT sector.

The explanation of the broad topic of PKI results is more understandable once its main features and working points are clarified. It is based on 2 digital tools: **certificates** and **keys**. The first ones can be thought of as trusted licenses which identify the owner and authorize him to make some active and/or passive actions

like interacting with or receiving documents.
The other pillar upon the PKI stands is represented by the encryption keys. They are usually numbers with the purpose of being encrypted and/or decrypted once associated with a mathematical formula.
The high-level working actors are 2: **Certificate Authority (CA)** and Registration Authority (RA). The latter is the entity providing digital certificates to users on a case-by-case basis. All of the certificates that are requested, received, and revoked by both the Certificate Authority and the Registration Authority are stored in an encrypted certificate database while certificate history and information are kept in a certificate store, which is usually grounded on a specific computer and acts as a storage space for all memory relevant to the certificate history. The release of a new certificate by the RA shall be approved by the CA.
The main characteristic the highest authority in PKI must guarantee is trustworthiness. It deals with the issuance of digital certificates and acts as a guarantor of the owner's identity.
Hence, the following key points are recognizable by what has just been said until now:

- **Customizability**: digital certificates can size up or down to accommodate any type of device.

- **Scalability**: PKI easily scales so you can manage high volumes of certificates effectively.

- **Competitivity**: IoT certificates are cost-effective and priced for high-volume.

- **High-security level**: PKI shows an extremely valuable identification encryption/decryption method both in terms of the micro-number of possibilities to break through either compared to the time quantity dedicated for the identification.

### 2.1.1 Security concepts

The 2 biggest problems to deal with, when we are talking about digital security, are the protection of sensible and confidential data and the verification of the parties' identity.
In order to make as hard as possible the purpose of breaking one of these security cores, encryption plays a crucial role in guaranteeing the following goals:

- **Confidentiality**:
  service keeping the content of information secret from all except those authorized to have it

- **Data Integrity**:
  service addressing the unauthorized alteration of data. To ensure data integrity, then it must include the ability to detect data manipulation by unauthorized parties

- **Authenticity**:
  service applying to both entities and information itself. This aspect usually is divided into 2 major classes: entity authentication and data integrity. It is strictly related to the identification.

- **Reliability/Availability**:
  Property of service of being available and working reliably

- **Non-repudiation/Accountability**:
  service preventing an entity from denying previous commitments and actions.

- **Privacy**:
  service protecting personal data, where for personal data we mean any information relating to an identified or identifiable natural person, so one who can be identified, directly or indirectly,

### 2.1.2 Digital certificates and keys

Public Key Infrastructure founds its ability to authenticate on digital certificates. These are digital tools acting as an identity profile. In PKI, their structure is dictated by the X.509 standard [5] setting specific mandatory and optional fields and their "physical" characteristics. These features are described better in the VDE-AR-E 2802-100-1 guide which will be analyzed in more detail later. For now, it is basic to observe that every X.509 certificate includes a public key, digital signature, and information about both the identity associated with the certificate and its issuing certificate authority (CA).
In particular, the public key is a part of the other tool that works synergistically with digital certificates which is the key pair.
In general, a key pair, also called cryptographic key pair, is a couple of data strings used to lock and unlock cryptographic functions like authorization, authentication and encryption/decryption. The key pair can be divided into 2 categories corresponding to 2 encryption methods:

- **symmetric-key algorithm**: 2 equal keys are used to encrypt and decrypt a message or a signature. This method works with a high computational speed, but the security level is quite low.

- **asymmetric-key algorithm**: 2 different keys are used; the private key, secretly stored by the owner is used to encrypt, while the public one is used by the external actor to decrypt the object and vice versa if the communication flow goes reversely.

Focusing on the second encryption method, adopted in the PKI environment, the private key can only be accessed by the owner of a digital certificate, and they can choose to whom the public will be distributed. Then, a certificate is essentially a way of handing out that public key to users that the owner wants to have it.
The asymmetric key pairs ensure a higher level of security given the fact that files encrypted by the private key can only be decrypted by the public key and vice

versa. If the public key can only decrypt a specific file that has been encrypted by the private key, it means that the file assures the intended receiver and sender took part in the informational transaction. Most often used for one-way communication, asymmetric encryption utilizes separate keys that are mathematically connected; whatever is encrypted in the public key can only be decrypted by its corresponding private key and vice versa.

### 2.1.3 Signature generation

A fundamental tool of digital security is the concept of Signature. It applies not only to a certificate but also to a message, as will be shown later. A digital signature is an alphanumeric series that attests to the authentication of an object and guarantees that it has not been tampered.
A further distinction must be done before proceeding. There exist 2 certificate types:

- **Self-signed Certificate**: the certificate owner can sign the certificate itself without relying on a CA.
- **Signed Certificate**: the certificate owner shall send a Certificate Signing Request (CSR) to a CA or Sub-CA to sign its own certificate and allow the signature generation.

In this case, let's consider an entity able to self-sign its own digital certificate using an asymmetric-key method. The signature is simply derived by taking the certificate in a human-reading format and applying sequentially 2 encryption algorithms. The first is a hash algorithm (SHA-256, ...). It is a widespread mathematical function that takes as input an object, whatever it is either readable or already encrypted, and converts it into an alphanumeric series. This process is irreversible.
The second encryption method sees the use of the private key to encrypt the hashed certificate. The inverse process is possible only using the public key. What the

owner obtains, is the certificate signature.
It demonstrates the authenticity of the certificate since, when attached to the original certificate and received by a second party, this can apply the corresponding public key and return it to the hashed certificate.
The final check is done by hashing the original certificate with the same algorithm and comparing the 2 obtained hash series.
A signed certificate is created in the same way with the difference that the key pair and the signature are produced by the CA receiving the CSR.

## 2.2 VDE-AR-E 2802-100-1

The VDE application guide [6] establishes the framework for creating a certificate policy and/or Certification Practice Statement document in the context of the Plug&Charge process described in DIN EN ISO 15118 (all parts), which specifies an authentication, authorization, and billing process. Moreover, it applies to the use of certificates in the expanded scope of this standard including the so-called Secondary Actors (SAs), described in section 2.2.1.
Additionally, the leading core is to put forward possible technical alternatives for the installation and the update of a contract certificate in the vehicle.

### 2.2.1 eMobility Panorama

The concept of Public Key Infrastructure [7] is applied to the modern eMobility market. It is referred to as a PKI ecosystem, an environment hosting all the players involved in the eMobility panorama. A PKI ecosystem includes single PKIs, each with a specific structure and a definite number of SAs, ruled by ISO15118 standard.

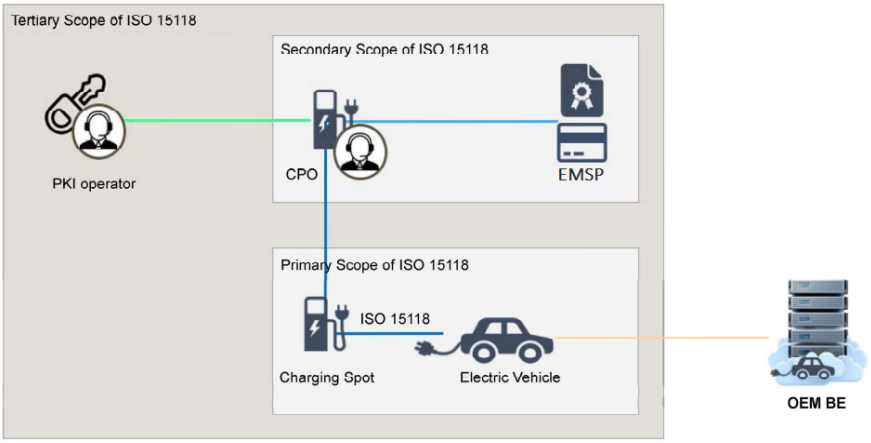


**Figure 2.1:** Single PKI structure.

The diagram above describe simply which are the main roles inside a single PKI. Going into details:

- **End customer**:
  the one who is going to use the EVSE. He has access to a charging and billing process with the greatest possible degree of automation, so installation and replacement of any necessary charging contracts in the form of a digital certificate in the EV (contract certificate).
  The user may wish to have the option of using more than one charging contract in his EV and selecting the appropriate charging contract and associated sales tariff to suit his purposes.

- **Vehicle Manufacturer (OEM)**:
  the company releasing the Electric Vehicle. It shall enable end customers to conduct automated charging and billing processes.

- **E-Mobility Service Provider (eMSP)**:
  Company in charge of offering EV charging services to the end users. They provide access to the eMSPs network making the customers sign one or more charging contracts with which they can conduct automated charging and billing

processes using the most comprehensive charging infrastructure possible.
Providers of mobility services are the contract parties with which end customers issue digital X.509 contract certificates. Through the use of contract certificates, end customers can authenticate and authorize themselves via their EVs at Charging Points. An eMSP may be also called MSP or MO, and it has the possibility to act as a Trust Anchor for its certificate chain.

- **Charging Point Operator (CPO)**:
  company acting in 2 roles:
  - Commercial CPO: purchases electricity for the Charging Points and provides their infrastructure. Its objective is to grant access to its charging framework to as many customers as possible.
  - Technical CPO: provides IT system for the Charging Station and installs Trust Anchors (V2G Root CA and MO root CA certificates), CPO certificates and the associated certificate chains. The IT network back from the EVSE is called "backend".

- **Certificate Provisioning Service (CPS) operator**:
  provides an IT system to receive the Contract Certificate Bundle from MOs, process the data and create a Signed Contract Certificate Bundle, then made it available to the CCP.
  Required to ensure that the Contract Certificate Bundle received from MOs is valid.

- **Contract Certificate Pool (CCP) operator**:
  handles the messages for installing and updating new contract certificates; they are called *CertificateInstallationRes* and *CertificateUpdateRes*.
  The CCP operator provides an IT system to receive precompiled messages (XML format) from the CPS as part of the Signed Contract Certificate Bundle and has the responsibility to store this data and deliver the corresponding

EXI-encoded and Base64-encoded *CertificateInstallationRes* and *CertificateUpdateRes* messages to the requesting CPO backend or OEM backend.

- **OEM Provisioning Certificate Pool (OEM PCP) operator**:
  provides an IT system to transfer Vehicle Data to an authenticated MO upon request.

- **V2G Root CA operator**:
  the lead of the PKI pyramid, it is a company in charge of issuing and revoking digital certificates, acting as trust anchors compulsory for CPO and optionally for MO and OEM.
  The V2G Root CA performs the following tasks:

  – issuing and revoking V2G Root CA certificates.

  – providing a root certificate pool for centrally retrieving and filing the root CA certificates and implementing an authentic connection between the 2.

  – providing a central message broker to exchange status messages of the processed data entities such as contract certificates or OEM provisioning certificates. The status values exchanged through this message broker reflect the processing state of the sent data.

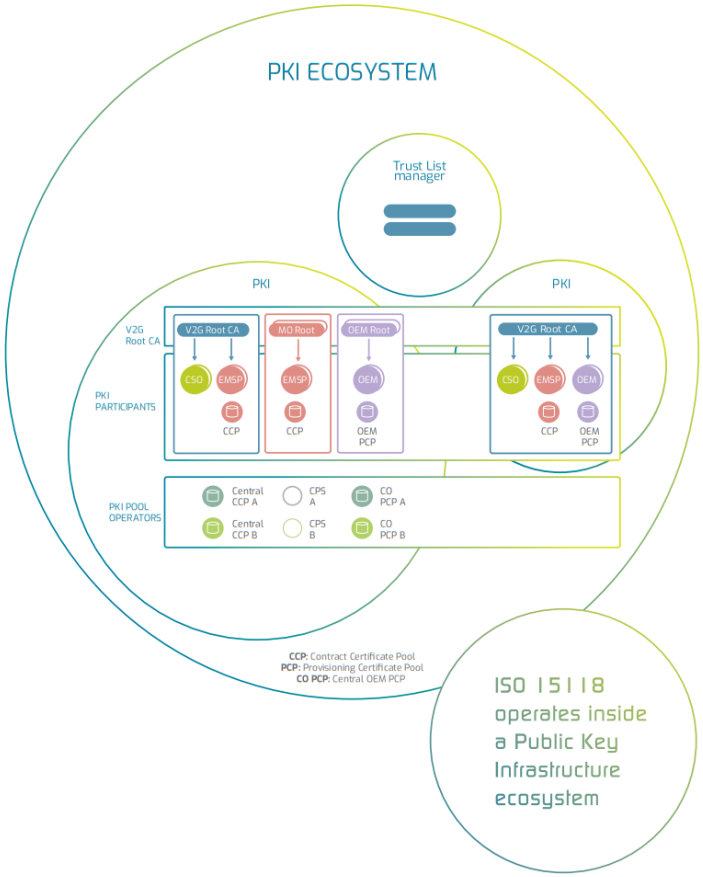


**Figure 2.2:** Overview of PKI ecosystem. [8]

### 2.2.2 Certificates types and main features

Digital certificates play a key role in realizing recognition and mutual trustworthiness. To this end, there is not only one certificate for each party, but a chain structure with different "trust levels" such that if even one link in a chain loses its trust or has been compromised, all certificates would be revoked in cascade from that point.

It should be noted, however, that the chain can not be too long to avoid memory shortage problems, i.e., each chain must have no more than 4 levels. The certificates at the top are called trust anchors. In the middle there are 2 Sub-CA certificates,

while at the end there is the leaf certificate. It is the end of the chain and it is important to emphasize that its associated private key shall not be used to sign anything.
A last clarification shall be done before focusing on each certificate. Not for all the chains the trust anchors has to be the V2G Root CA. EMSPs and OEMs may act as CAs as well. All the possible certificates involved in the e-mobility ecosystem are briefly described below:

- **V2G Root CA certificate**:
  issued by V2G Root CA is used as the basis for verifying all other derived certificates to ensure that they are genuine and trustworthy. It may be used as a top-level trust certificate for any chain, but is only binding for the CPO and CPS.
  The corresponding private key is held by the respective certificate authority (root CA).
  OEMs shall ensure that the vehicle's memory contains at least the V2G Root CA certificates that cover the vehicle purchaser's primary driving areas, including roaming to neighboring countries; in the situation where the vehicle does not have a suitable V2G Root CA certificate stored, an error must be reported and the vehicle manufacturer or user shall make an effort to verify and store the V2G Root CA certificate.

- **MO root CA certificate**:
  EMSP uses this certificate as the MO's top-most trust anchor.

- **OEM root CA certificate**:
  OEM uses this certificate as the OEM's top-most trust anchor.

- **Contract certificate**:
  it is associated with a charging contract. It shall be stored inside the EV, together with its private key, in the interest of proving to the charging point the existence of a valid contract with the EMSP.

It is the certificate core allowing Plug and Charge.
Contract certificate is issued, possibly, by a MO/V2G Sub-CA 2.

- **Charging Point Certificate (SECC certificate)**:
  provides the authentication of the charging station to the EV.
  It shall be derived from a V2G Root CA and it shall be stored in the EVSE.

- **OEM Provisioning certificate**:
  issued individually for and saved in each EV and it is used to verify the identity of the EV when provisioning a contract certificate. It shall be possible to renew the provisioning certificate in the vehicle if it's revoked. This process can be done by a workshop or by means of an online process using the OEM backend and telematics link of the EV. It is derived from an OEM/V2G Root CA.

- **Certificate Provisioning Service (CPS) leaf certificate**:
  uses the private key belonging to the leaf certificate to sign parts of the Contract Certificate Bundle received from the MO (including the contract certificate and the associated private key). Based on this signature, the EV determines the authenticity of the contract certificate and associated private key to be installed in the vehicle.

- **Private operator root CA certificate**:
  it is stored by a private operator in its charging infrastructure (no V2G root CA) located in a private environment (PE). The corresponding private key associated with the private operator root CA certificate is held by the respective private operator root CA.
  No Sub-CA certificates are used in a PE. This means that the Private operator root CA directly issues and signs charge point certificates for a privately used charging infrastructure.

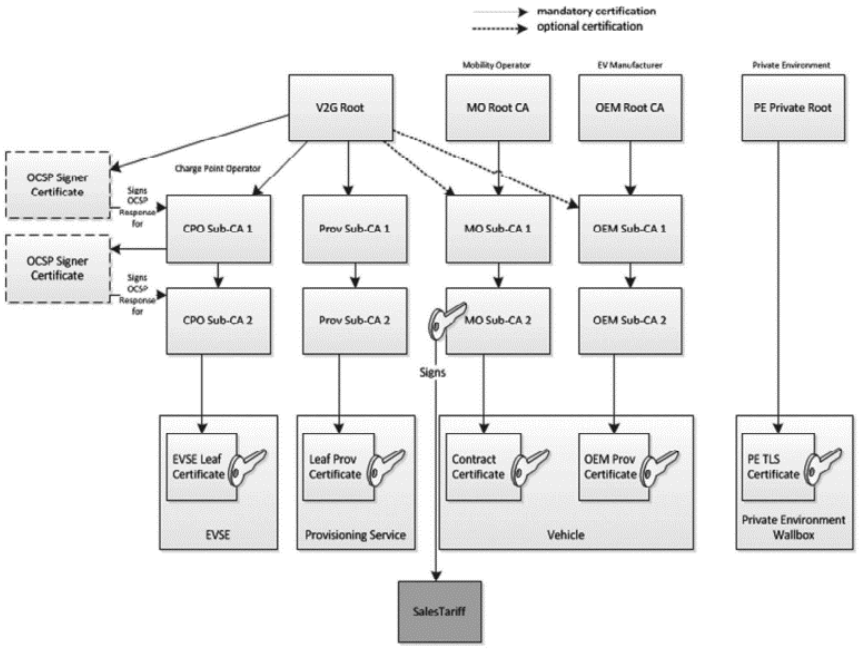


**Figure 2.3:** Schema of every certificate chain involved in PKI.

In this PKI scenario, the only self-signed certificates are the Root ones. All the other Subs and Leaf certificates and signed using a CSR by their upper-level entity. Besides the certificate types, some other objects need to be defined a priori in sight of the installation/update process description.

- **Provisioning Certificate ID (PCID)**: it is a unique alphanumeric identifier associated to a specific EV.
- **E-Mobility Authentication Identifier (EMAID)**: it is a unique alphanumeric identifier that associates the vehicle's PCID with an eMSP Charging Contract.
- **Contract Certificate Bundle (CCB)**: contains data elements the EMSP sends to the CPS to create a signed *CertificateInstallationRes* and *CertificateUpdateRes* message. Once the CPS sends it to the CCP, then it becomes **SignedCCB**.

- **Vehicle Data**: contains data elements that the OEM provides to the OEM PCP.

- **XSD schema version**: it is the unique namespace URI of an XSD schema that is related to the DIN EN ISO 15118 (all parts) version used in the EV in which the contract certificate and private key need to be installed or updated.

The "physical" certificate values must be set for the ISO 15118 application. Their dimensions are ruled by the X.509v3 format according to which the certificate's size shall not exceed 800 bytes. The number of V2G Root CA stored in the EV shall be between 1 and 5, while the maximum number rises to 10 for the EVSE.

### 2.2.3 Certificate validation

The term certificate validation indicates the action required by an entity to verify the trustworthiness of the certificate and of the CA and Sub-CAs behind it.
The leaf certificate is the last link in the chain and is directly involved in the validation process as a starting point. It can be trusted if several conditions are met.
The first states that its content shall not be altered after issuance; this check can be performed to confirm the signature up to the trust anchor level and thus the integrity of the signed content. Of course, the expiration date shall not have expired and all fields of the X.509v3 format are not empty and valid. The last item that needs to be valid and acceptable is the subject ID.

The policy for checking validity is referred to as a validity model. Usually, there exist 3 different types of validation models:

- Shell model:
  a signature for a leaf certificate is valid if the higher-level CA certificates in

the certificate path up to the root certificate are valid at the time of the check (the current time).

- Chain model

- Shell-chain hybrid approach

In the VDE and ISO 15118 context, the shell model is applied. The main advantage of its use is getting simpler the certificate check since it does not require that the signature for a leaf certificate shall not be valid for any longer than its higher-level CA certificates. However, the other coin side is represented by the unavailability of the certificate infrastructure if its CA has been revoked; then, a revocation "en masse" is the result.
More precisely in ISO 15118, the X.509v3 certificate format has been adopted and it has been accepted the use of "Basic Path Validation", where The algorithm checks a certificate in respect of the current date and the current time (shell model). PKIX certificate and CRL profile (PKIX profile) have been listed in ISO 15118 standard for normalizing all the certificate types.

An important difference between all the certificates involved in the process of validation is the expiration date. It may result differently and usually shortens its time interval in the leaf certificate direction.
The longest expiration date is owned by the V2G Root which must be renewed as rarely as possible and it is set to 40 years. For the CPO/CPS Sub-CA1s the validation period is 4 years and half of it for CPO/CPS Sub-CA 2s. MO and OEM are in charge of their own certificates and of their Sub-CAs.
The expiration dates of the 2 remaining leaf certificates are: 3 months for the SECC certificate and from a maximum of 4 years and a minimum amount of time of 4 weeks for the contract certificate.

### 2.2.4 Certificate Installation and Update

The process of certificate installation and update is a complex trial that involves a lot of actors. Considering a single PKI ecosystem, the full certificate generation is clearly described with the following schema:

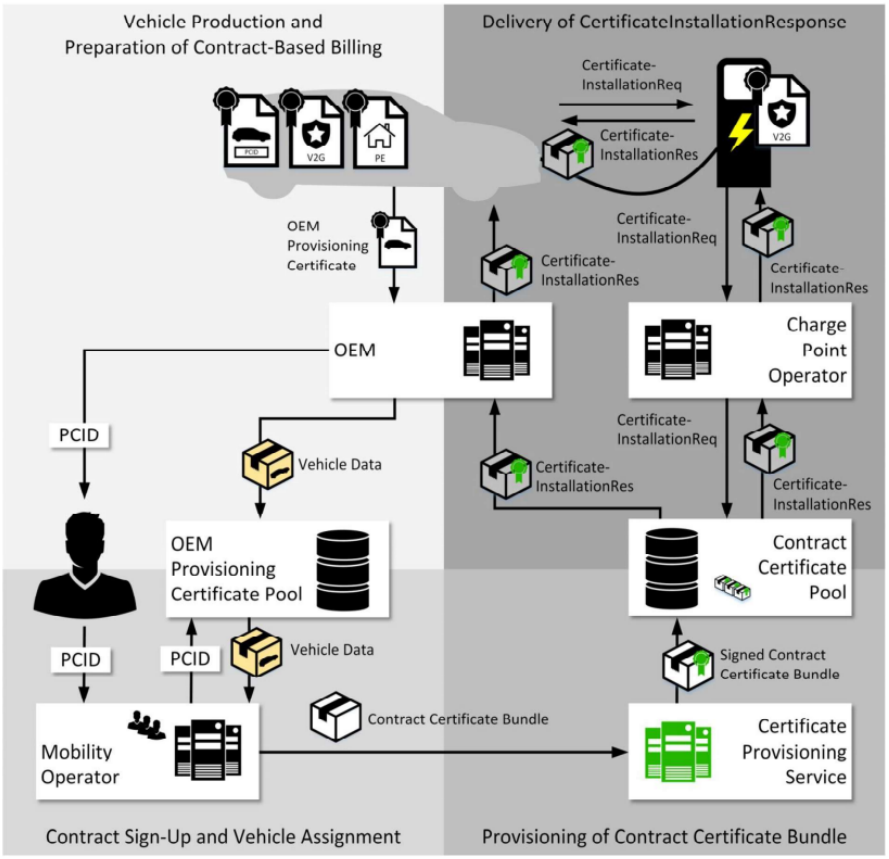


**Figure 2.4:** Overall system approach to facilitate the Plug and Charge process.

This high-level flow chart is the main and more understandable diagram to follow the path for the generation of a contract certificate between the user's EV and the eMSP (MO).

The VDE guide is written in order to face all the certificates' journeys from the EV creation to their installation. Thus, a brief overview of the context before the actual certificate installation and update in the EV is going to precede a more

detailed final part, which is the focus of this project and it is the key point that makes the Plug&Charge work.
As highlighted in Figure 2.4, it is useful to divide the full certificate generation process into sub-processes:

- Preparing contract-based public charging and billing:
  1. Providing root certificates for public charging and contract-based billing
  2. Producing the vehicle and signing the contract
  3. Assigning the vehicle to a contract
- Providing a contract certificate for automated billing:
  1. Providing a contract certificate
  2. Delivering a contract certificate upon request

### Vehicle Production, Contract sign-up and Vehicle assignment

The entire process starts with all the SAs creating and depositing their identity, under the form of Root certificate, in the RCP.
In the meantime, the user has to complete 2 tasks: order his new EV and sign a contract with an eMSP entity.

- **EV production** the OEM has to carry out the generation of the PCID and the registration of the OEM Provisioning Certificate in the PCP, in addition to producing and delivering the required vehicle.
- **submit of a Charging Contract** the user has to choose an eMSP and issue a contract with it, generating a ContractID.

The PCID and the ContractID are collected by the user and sent to the eMSP which is going to validate them. Then, the PCID is sent to the PCP to create a data pack called Vehicle Data. It contains the OEM certificate chain that the OEM shall provide to the PCP.

The Vehicle Data is sent back to the eMSP such as the Provisioning Certificate can be validated and the new vehicle can be assigned to the user's contract. In this step, the eMSP adds another important element for the PnC gear; it generates the eMAID.

### Provisioning of Contract Certificate Bundle

The second step involves the creation of a Contract Certificate Bundle (CCB) by the eMSP using the eMAID and some other data derived from previous processes. At this point, it is kicked in the CPS where it is signed and sent it to the CCP. This signing process implies the draft of the 2 messages required by the ISO15118-2 standard, called*CertificateInstallationRes* or *CertificateUpdateRes*, besides the contract certificate chain validity check and the CCB decryption. The VDE application guide specifies also the exact procedure to create the signature of these 2 messages, better defined in Chapter 3.
Once the data pack has been received, the CCP thins out the data storing only the PCID, the eMAID, the XSDSchemaVersion, useful for further decryption in the EV-EVSE communication, and the *CertificateInstallationRes* or *CertificateUpdateRes* message.

### Delivery of *CertificateInstallationRes/CertificateUpdateRes*

The ISO 15118-2 standard imposes restrictive timing requirements on the time with which the Contract Certificate shall be installed or updated. Therefore, the role of the CCP is critical to ensure prompt delivery of one of these response messages when the Charging Point will require them.

The last step takes place when the EV arrives at the Charging Point and a Certificate Installation or Update session is required. The request is forwarded by the Charging Station to the Charging Station Management System (CSMS). In particular, the CPO must receive the request from the vehicle to issue or update

the contract certificate in the form of an EXI-encoded *CertificateInstallationReq* or *CertificateUpdateReq* message. The Charging Station shall then forward this request to the CPO, using a Base64 encoding format. The forwarded request shall also include the XSD schema version used by the EV so that the CCP is able to properly decode the EXI message. The CPO shall then forward the EXI-encoded *CertificateInstallationReq* or *CertificateUpdateReq* message and the XSD schema version to the CCP using the Base64 encoding format. This means that there is no need for the CPO to further process the message itself, it simply acts as a forwarding gateway to the CCP.
Decoding the Request message using the XSDSchemaVersion allows the CCP to get the Response message, encrypt it and send it so that it gets back to the EV.
The actual installation or update of the contract certificate and especially the composition of the *CertificateInstallation* and *CertificateUpdate* request and response messages are better explained in the next chapter.

# Chapter 3

# DIN EN ISO 15118 standard: Vehicle-To-Grid Communication Interface

With the arrival of EV technology and its spread all over the world, came out the need to unify and regularize the components of this newborn business area. In other words, it was necessary to introduce rules on digital communication networks and on physical connectors, from the electrical components inside the charging points to the digital messages interchanging between EV and EVSE.
In particular, the set of norms dealing with the bidirectional communication between the Electric Vehicle Communication Controller (EVCC) and the EVSE controller called Supply Equipment Communication Controller (SECC) is called ISO 15118 standard.

## 3.1 DIN EN ISO 15118 standard: Vehicle-To-Grid Communication Interface

The ISO15118 standard protocol imposes all the use cases, listed in detail in its first part [9], and the requirements that need to be satisfied to implement digital communication between an electric vehicle and a Charging Point [10]. The standard defined them at all levels and follows the basic template of the Open Systems Interconnection Model (OSI Model).
The latter describes 7 layers that computer systems use to communicate over a network.
In the following there is a brief description of every layer:

1. **Physical layer** –> lowest layer, responsible for the physical connection between devices. The physical layer contains information in the form of bits. It is responsible for transmitting raw bit stream over the physical medium.

2. **Data layer** –> responsible for the node-to-node delivery of the message. The main function of this layer is to make sure data transfer is error-free from one node to another, over the physical layer.

3. **Network layer** –> works for the transmission of data from one host to the other located in different networks. It also takes care of packet routing. The sender and receiver's IP addresses are placed in the header by the network layer.

4. **Transport Layer** –> provides services to the application layer and takes services from the network layer. It is responsible for the End to End Delivery of the complete message and provides the acknowledgment of the successful data transmission and re-transmits the data if an error is found.

5. **Session layer** –> responsible for the establishment of connection, maintenance of sessions, authentication, and also ensures security.

6. **Presentation layer** –> also called Translation layer, extracts data from the application layer and manipulates them as per the required format to transmit over the network.

7. **Application layer** –> top layer, implemented by the network applications. Data produced by them has to be transferred over the network. This layer also serves as a window for the application services to access the network and for displaying the received information to the user.

The ISO 15118 is based on the OSI model and to cover all the levels (Figure 3.1), it is divided into 7 parts and a new section which is the update of the second section:

- **ISO 15118-1**[9] General information and use cases definition:
  It is the basis for the other parts of the ISO 15118 series and specifies terms and definitions, general requirements and use cases for conductive and wireless High-Level Communication (HLC) between the EVCC and the SECC. It is applicable in both cases V2G provides.

- **ISO 15118-2**[10] Network and application protocol requirements:
  The core of the entire international standard: it defines all the messages and related technical requirements that are necessary to implement the use cases defined in ISO 15118-1. IP-based communication between EVCC and SECC. It defines messages, data model, XML/EXI-based data representation format, and usage of V2GTP, TLS, TCP, and IPv6.

- **ISO 15118-3** Physical and Data link layer requirements:
  It specifies the requirements of the physical and data link layer for high-level communication, directly between EV and modes 3 and 4 of EVSEs, based on a wired communication technology and the fixed electrical charging installation used in addition to the basic signaling.
  It covers the overall information exchange between all actors involved in the electrical energy exchange.

It addresses how powerline communication (PLC) is used to modulate the digital information specified in ISO 15118-2 onto the Control Pilot pin inside the connectors. Part 3 also describes a mechanism called SLAC (Signal Level Attenuation Characterization), which is used by the EV and the Charging Station to establish the data link between each other.

- **ISO 15118-4** Network and application protocol conformance tests:
  It specifies conformance tests in the form of an Abstract Test Suite (ATS) for a System Under Test (SUT) implementing an EVCC or SECC according to ISO 15118-2.

- **ISO 15118-5** Physical and Data link layer conformance tests:
  It specifies conformance tests in the form of an ATS for a SUT implementing an EVCC or SECC with support for PLC-based High-Level Communication and Basic Signaling according to ISO 15118-3.

- **ISO 15118-8** Physical and Data link layer requirements for wireless communication:
  It specifies the requirements of the physical and data link layer of a wireless HLC between EV and EVSE. Wireless communication technology is used as an alternative to wired communication technology as defined in ISO 15118-3. It is applicable for conductive charging as well as Wireless Power Transfer (WPT).

- **ISO 15118-9** Physical and Data link layer conformance tests for wireless communication:
  This specification provides conformance tests for the use cases in part 8 and completes the current list of required conformance tests for both wired and wireless communication.

- **ISO 15118-20**[11]: second generation network and application protocol requirements:

It is an updated version of ISO 15118-2, with additional features like wireless charging, bidirectional energy transfer (V2G and V2H), and charging buses via pantographs. This standard will be deepened in the last chapter.

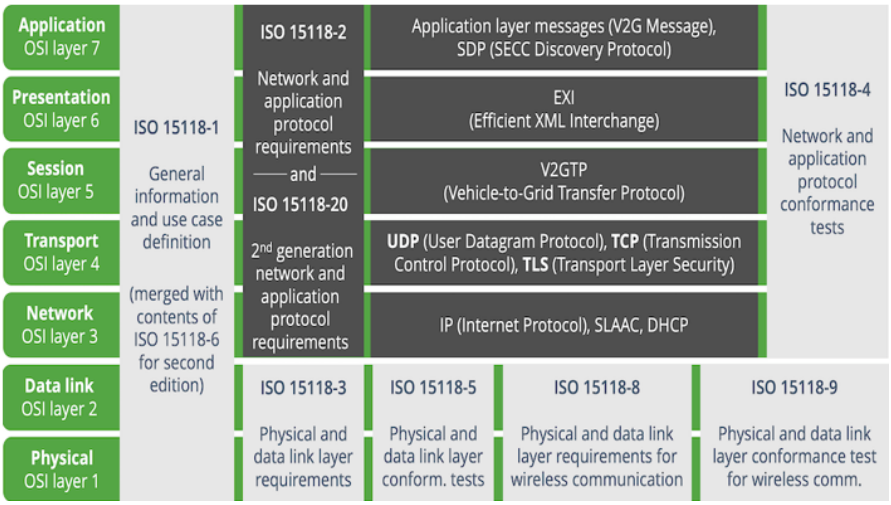


**Figure 3.1:** ISO 15118 standard implementation on the OSI Model.

## 3.2 ISO 15118-2 standard

This section is focused on analyzing the structure and on what are requirements the EVCC-SECC system communication needs to satisfy so that it can be considered valid. The most important features and services that this section implements, and that will be explained in more detail, are the already quoted smart charging, the Plug and Charge (PnC), the data security system PKI-based, Renegotiation during charging processes and the Value Added Services (VAS).

### 3.2.1 Overview and objectives

Quoting literally what the standard says: "This part of ISO 15118 specifies the communication between battery electric vehicles (BEV) or plug-in hybrid electric vehicles (PHEV) and the Electric Vehicle Supply Equipment. The application layer message set defined in this part of ISO 15118 is designed to support the

energy transfer from an EVSE to an EV. ISO 15118-1 contains additional use case elements (Part 1 Use Case Element IDs: F4 and F5) describing the bidirectional energy transfer. The implementation of these use cases requires enhancements of the application layer message set defined herein. The definitions of these additional requirements will be subject of the next revision of this International Standard"[10]. The purpose of this part is to describe in detail the communication between an EV and an EVSE. Aspects are specified to detect a vehicle in a communication network and enable an Internet Protocol (IP) based communication between EVCC and SECC.


**Figure 3.2:** Macro scheme of the ISO 15118-2 objective.

### 3.2.2 Structure of the standard

The standard is divided into 8 chapters in which the first 4 clarify what are the terms and the symbols used, introduce some definitions and conventions and state the scope of the document. The fifth and the sixth chapter define some other basic acknowledgments as definitions of OSI-based service, notations for XML schema diagrams and a brief document overview.

The most interesting chapters are the last ones since they go into the merits of what shall be effective and what shall be the digital structure of the EVCC-SECC communication.

For this reason, these 2 are also the longest and the most complex. Thus, to explain them as clearly as possible, it is possible to ideally group subsections based on the arguments they handle.

Chapter 7 faces 3 big fundamental features: security, the main state flow of the

communication and the structure of the messages.

A little premise is necessary to clarify how the communication is organized according to the standard. The Charging Station (SECC) always acts as a server while the Electric vehicle (EVCC) always behaves as a client. These 2 communicate using different protocols that will be described later, but it is important to keep in mind that their communication is based on a TLS handshake. As the last thing, the 2 parties' communication is based on a well-determined messages exchange; during this process, the server and the client pass through individual "phases", each with defined characteristics, called states.

The figure below shows an example of the flow model.

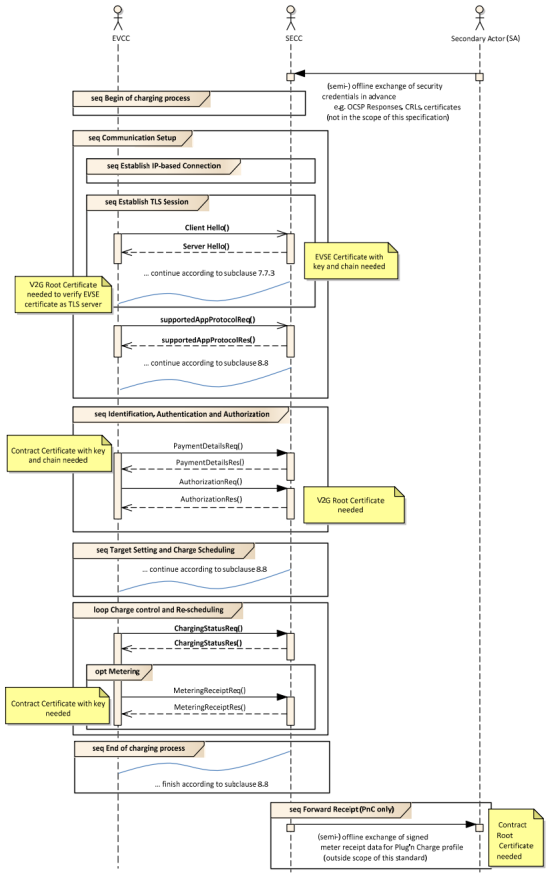


**Figure 3.3:** Example of a SECC-EVCC communication sequence.

As described in Chapter 7, the first argument introduced is the **digital security**.

The security concept provides a basic transport-based protection mechanism. The usage of Transport Layer Security (TLS) always may be applied, but it is not mandated for all scenarios. The expected version of the TLS is 1.2 or later.
The main notion of security in ISO 15118-2 standard is digital certificate-based or PKI-based security. The authentication and the protection of data inside messages are done by the verification of these certificates, and in some cases of their signatures, to different levels. This argument will be faced with a very high level of detail at the end of this chapter. Thus, the only information needed to understand the following process is the basic characteristics of the certificates.
There exist different types of the certificate, but for all of them it is mandatory the structure of X.509v3-certificate: The other requirements the standard impose

| **Certificate field** | **Description** |
|---|---|
| Version | Version of certificate (for 15118 shall be v3 = 0x2) |
| Serial Number | Unique certificate number (within the domain of the issuer) |
| Signature Algorithm | Used signature algorithm |
| Issuer | Entity, who has issued and signed the certificate |
| Validity Period | Time period, in which the certificate is valid |
| Subject | Entity, to which the certificate is issued |
| Public Key | Public key corresponding to a private key |
| Issuer UID | Optional issuer unique identifier |
| Subject UID | Optional subject unique identifier |
| Extensions | Optional fields |
| Signature | Signature of the certificate generated by the issuer |

**Table 3.1:** Digital certificate fields according to X.509v3-certificate.

are the dimension of the certificates, but, as it has already been said, a separate chapter will be dedicated to it.

Chapter 7 goes on to impose in which cases the TLS handshake is mandatory and when it results optional. The only case in which the latter case happens is when an External Identification Mode (EIM) is required by the user.

The term EIM refers to one between QR code or RFID identification mode; it continues with the classical smart charging process. The other identification modality is called Plug and Charge (PnC) and is the main innovation point that the standard introduces in the SECC-EVCC communication together with V2G connection.

It is essentially referred to as an identification process that takes place without the user's action and the only requirement that it needs is the connector plugged in. The following macro-argument is the **sequence of states**. Actually, it shows only the general communication states of the V2G communication from an EVCC perspective (Figure 3.4).

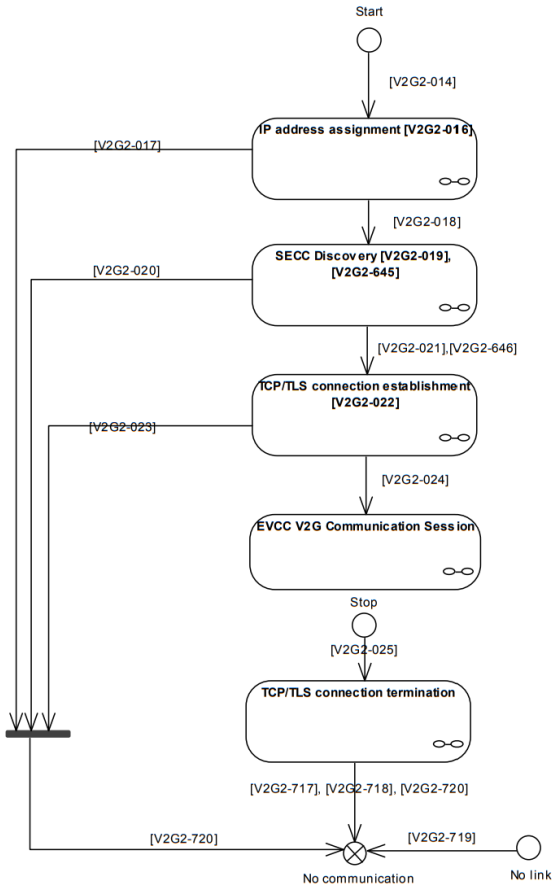


**Figure 3.4:** Scheme of general communication states from an EVCC perspective.

For complete comprehension, it results necessary to underline that the code on

the line in the figure named 'V2G2-XXX' refers to the requirements in the document.

At last, it presents the explanation of how the OSI Layer is covered by the standard starting with the description of **communication protocols** implementation. The OSI Transport Layer sees the implementation and the requirements of the following 3 protocols:

- **Transmission Control Protocol (TCP)**:
  - allows applications of V2G entities to establish a reliable data connection to other entities,
  - exchanges reliable and proper delivery of data between senders and receivers,
  - provides flow and congestion control.
- **User Datagram Protocol (UDP)**:
  connectionless protocol which does not provide the reliability and ordering already guaranteed by TCP. Packets may arrive out of order or may be lost without notification from the sender or receiver. However, UDP is faster and more efficient for many lightweight or time-sensitive purposes.
- **Transport Layer Security (TLS)**:
  provides the "real" data security system. This allows to establish an authenticated and encrypted (ensures integrity protection and confidentiality protection) channel between the EVCC and the SECC. For ISO 15118 security it was agreed to use unilateral authentication where SECC acts always as server (the EVCC authenticates the SECC).

A further protocol called **V2G Transfer Protocol (V2GTP)** dictates the structure of a transmitted message. It is a compact communication protocol to transfer V2G messages between 2 V2GTP entities, which consists of SECC and EVCC in this case. Its Protocol Data Unit (PDU) mainly consists of a header

and payload definition (Figure ) that allows the separation and process of V2G messages efficiently.
V2GTP is based on TLS+TCP which uses a pair of IP addresses (source address and destination address) and a pair of port numbers (source port and destination port) to establish and identify a connection for the bidirectional exchange of byte streams.

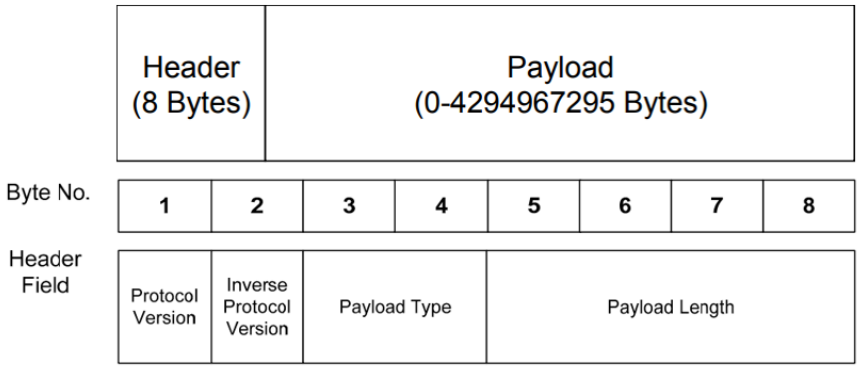


**Figure 3.5:** V2GTP Message structure.

Therefore, a message shall be subject to a sub-distinction of its single fields based on simple and complex types, which will be described in Chapter 8. To describe the V2G Message Set the Presentation Layer uses the widely adopted XML data representation accordingly the document defines messages (i.e. data structures and data types) based on XML Schema which allows the type-aware use of XML and enables simplified validity estimation of exchanged messages.
The Efficient XML Interchange (EXI) format allows using and process of XML-based messages on a binary level. Thus, the EXI format increases the processing speed of XML-based data as well as reduces memory usage. Basically, EXI is a W3C recommendation. The EXI format uses a relatively more simple grammar-driven approach, than the basic XML format, which achieves very efficient encoding for a broad range of use cases. To meet the demands in ISO 15118 in terms of efficient processing, fewer resource usage, message size, and message extensibility schema-aware settings should be selected.

### 3.2.3 Message Signature

The Request-Response messages contain sensitive information about the user, its charging contract, charging parameters and services data. They shall be kept private and confidential and do not have to be altered by third parties.
Message security is provided by W3C-recommended XML Signature, which satisfies the authenticity requirements of some data fragments of XML-based V2G. XML Signature defines a mechanism by which messages and message parts can be digitally signed to ensure integrity, that the data has not been tampered with, and to verify the identity of the message originator.
The message signature works exactly as the certificate one analyzed in Chapter 2, but, in this case, it uses different encryption algorithms.
The encoding procedure sees 2 main steps: **reference generation** and **signature generation**. Reference generation includes all the steps required to add some XML data to be signed to the SignedInfo XML element. The signature generation then calculates the signature value. Each internal step is explained as follows:

- **add the Reference XML element**: the URI attributes are set to the message ID and the data is EXI encoded using the EXI schema based on V2G_CI_MsgDef schema.
- **hash and encryption of the EXI stream**: use the SHA-256 algorithm to hash the EXI-encoded message. This value is then Base64 encoded and added to the SignedInfo as DigestValue.

Once the reference generation has been completed, the signature generation shall be performed. For this final step, only the SignedInfo data is needed.

- **EXI encoding of the SignedInfo element**: the SignedInfo (URI+reference digest) is EXI encoded based on the XMLdsig schema.
- **hash and encryption of the EXI stream**: the EXI stream is hashed using the SHA-256 algorithm and then used as input for the ECDSA signature

algorithm with secp256r1. The signature generation is done by means of the private key.

- **final encoding and message completion**: the encrypted value is Base64 encoded and finally added to the Signature XML data as SignatureValue.

The following message shows what a final message looks like with an XML representation.


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns6:V2G_Message
    xmlns:ns6="urn:iso:15118:2:2013:MsgDef"
    xmlns:ns5="http://www.w3.org/2000/09/xmldsig#"
    xmlns:ns7="urn:iso:15118:2:2013:MsgBody"
    xmlns:ns2="urn:iso:15118:2:2010:AppProtocol"
    xmlns:ns4="urn:iso:15118:2:2013:MsgDataTypes"
    xmlns:ns3="urn:iso:15118:2:2013:MsgHeader">
    <ns6:Header>
        <ns3:SessionID>EE3EAB8B81FD9D54</ns3:SessionID>
        <ns5:Signature>
            <ns5:SignedInfo>
                <ns5:CanonicalizationMethod Algorithm="http://www.w3.org/TR/canonical-exi/"/>
                <ns5:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256"/>
                <ns5:Reference URI="#ID1">
                    <ns5:Transforms>
                        <ns5:Transform Algorithm="http://www.w3.org/TR/canonical-exi/"/>
                    </ns5:Transforms>
                    <ns5:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
                    <ns5:DigestValue>m4zPveFtBkFTOcdZeh9nnuxA+Z5cqa2DWYsy5zNDces=</ns5:DigestValue>
                </ns5:Reference>
            </ns5:SignedInfo>
            <ns5:SignatureValue>ANWySSIb2UeYfXdGsNonGJoyAt3Dx45yv4S7nG/LJ+D7qsluvvuBuObVWlkiU8R/8g3iM6yvCAOhUvR/wy6AEg==</ns5:SignatureValue>
        </ns5:Signature>
    </ns6:Header>
    <ns6:Body>
        <ns7:AuthorizationReq ns7:Id="ID1">
            <ns7:GenChallenge>Td4iACdm2zeTJ60sggohXA==</ns7:GenChallenge>
        </ns7:AuthorizationReq>
    </ns6:Body>
</ns6:V2G_Message>
```


**Figure 3.6:** XML representation of the AuthorizationReq.

### 3.2.4 State, messages and types

The core of the standard is defined in its eighth chapter where is specified in more detail what has been already anticipated about the syntax of the communication itself.
Its organization is based on 3 macro-categories: **states**, **messages** and **types**.

- **State**:
  is a condition/status that characterizes at which point of the communication cycle the EVCC or the SECC are placed. Although the sequence of states is not linear, it can show some ramifications, and its general structure is strictly mandatory. Figures 3.6 and 3.7 may clarify this point.

The first difference can be made between the 2 communication controllers. The SECC always acts as server and the EVCC as client. There is one more distinction, consisting of some different message types and it depends on whether the charging mode is AC or DC.

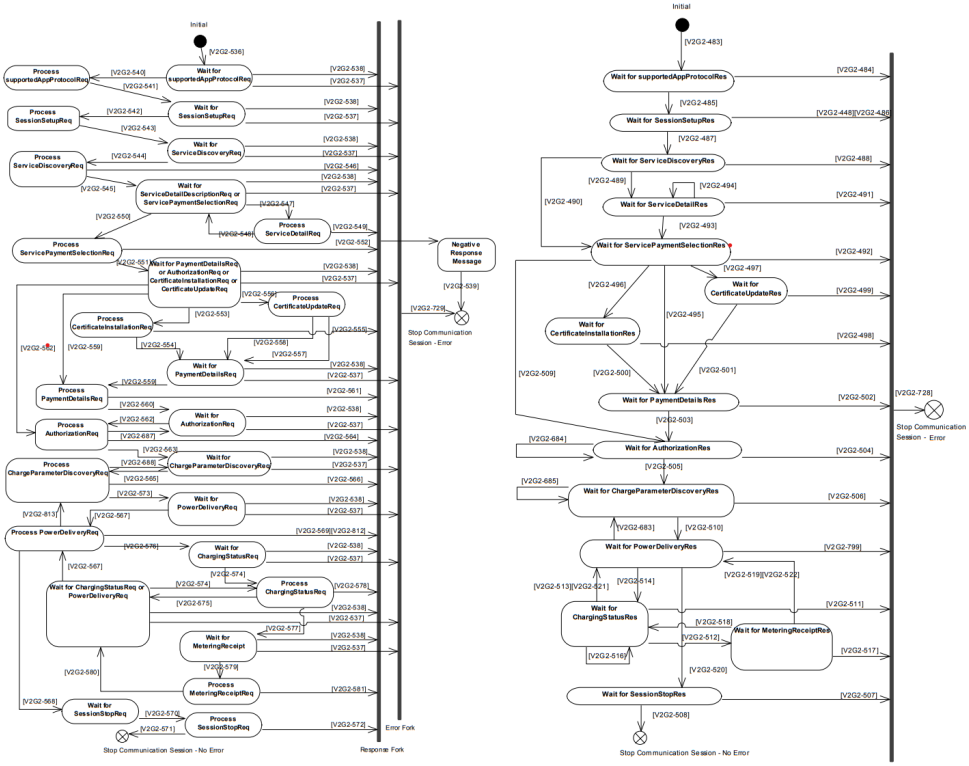


**Figure 3.7:** Sequence communication states for AC V2G messaging (left SECC, right EVCC).

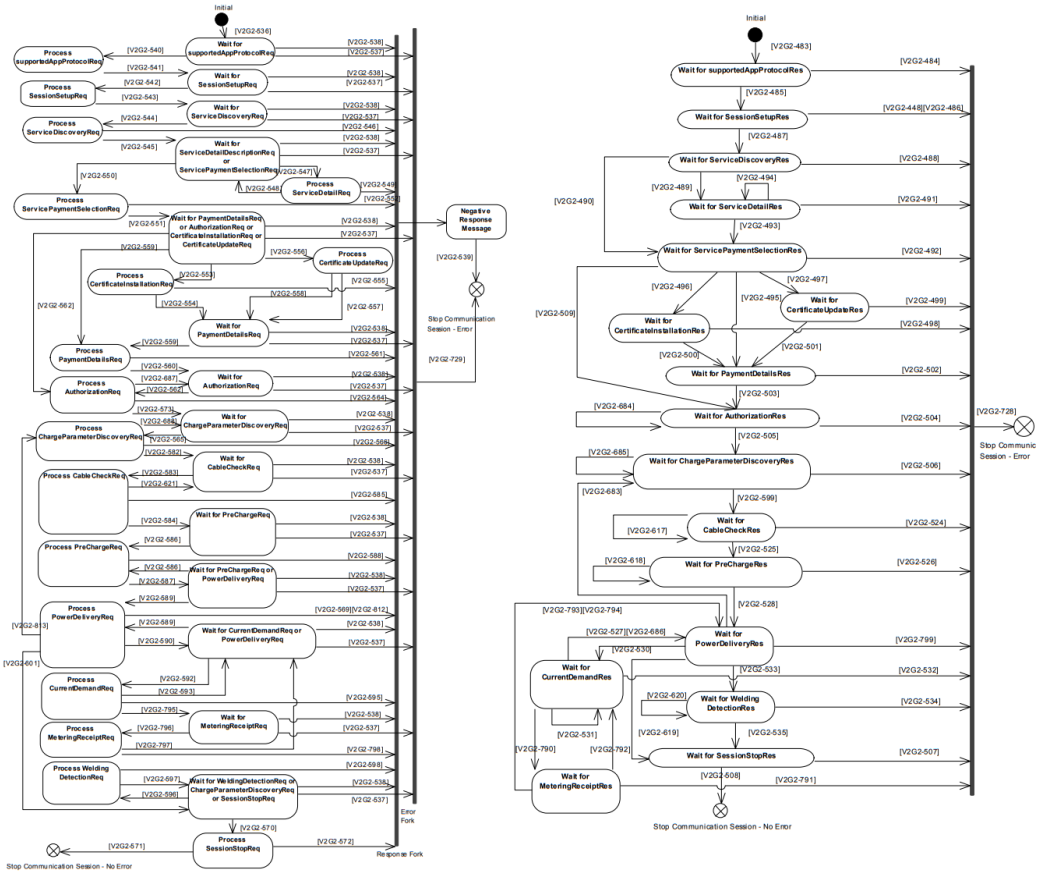


**Figure 3.8:** Sequence communication states for DC V2G messaging (left SECC, right EVCC).

- **Message**:
  is the object interested in transmitting the data and parameters between EVCC and SECC bidirectionally.
  There exist 2 kinds of messages:

  – Request message "-Req":
    sent by the EVCC to the SECC, contains EV information or the customer's chosen parameters. It is welcomed and read by the corresponding "WaitFor-" state;

  - Response message "-Res":
    sen by the SECC to the EVCC, contains EVSE-offered services or details about the charging process. It is received and read by the corresponding "WaitFor-" state.

ISO 15118-2 defines a common structure for all messages. It is composed by:

  - Header:
    includes generic information for protocol flow and is not directly related to the semantics of each particular message.
  - Body:
    provides the actual semantics of each message. In other words, it contains all the actual values and parameters that shall be transmitted to the other side.

The body structure is organized into sub-fields which contain the information necessary to be transmitted. These elements are obviously different for every message and some of them are encoded before being sent. The encoding applied is the Base64 and the EXI scheme. Only in some specific cases, the messages are also encoded, but generally, they are simply exchanged in XML format.

- **Type**:
  is the most low-level element that composes the communication syntax. Everything above, except for the states, is based on types; it means that every message and each sub-field of it belongs to a well-defined type.
  There exist 2 kinds:

  - **Simple Type**:
    it is a basic object in informatics, like Integer, String, Long, Byte, etc.

– **Complex Type**:
  it is a composition of 2 or more simple and/or complex types.

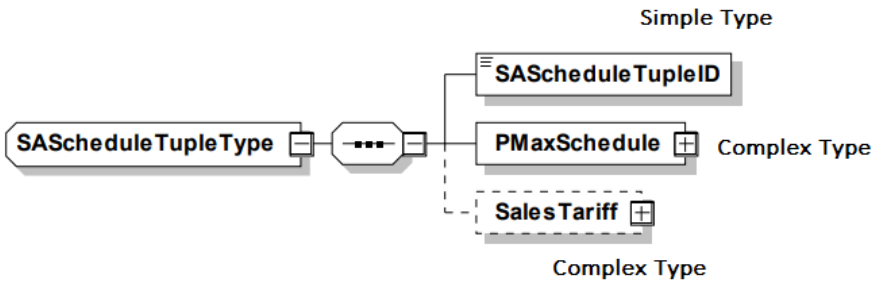


**Figure 3.9:** Example of a complex type and its sub-components.

Combining these 3 elements together, the ISO 15118-2 document founds the basis for a solid and compact communication syntax and flow. It results as much clear, readable, but especially 'implementation-friendly' as possible also with the purpose of avoiding misunderstandings and errors in the use cases application.

All the services described in this document present comparable behaviors and their semantics is quite similar. However, there is one particular service, called **Value Added Service (VAS)** which is barely touched but never deepened.
It refers to the possibility to create and include an additional set of services. What actually has been said by ISO 15118 regarding the VAS is how it can be implemented inside the messages, but a separate state is not provided by the standard. The messages providing information and the selection of VAS are *ServiceDiscoveryRes*, *ServiceDetailReq*, *ServiceDetailRes* and *ServicePaymentSelectionReq*. The VAS implementation requirements are the following: TLS 1.2 (or higher) shall be used, as a cryptographic communication protocol, and the parameterSetID, with which the service is identified, shall stay between 60001 and 65535.
The Value Added Service is more developed in terms of communication syntax in the ISO 15118-20 standard.

There is one last feature extremely important to understand how the standard works. It is the **Identification Mode**. The distinction has been made between 2 modalities:

- **External Identification Mode (EIM)**:
  is the modality implemented by all the installed charging station. The customer identifies himself using an RFID card or a QR code at the charging point.
- **Plug and Charge (PnC)**:
  this brand new mode requires the user to plug simply the cable into the Charging Station and the identification will happen automatically through the digital certificates cross-check between EV and EVSE. The novelty is the digital info passage inside the charging cable, which was not possible before.

Moreover, inside this 2 macro-category, shall be made a further classification between AC and DC charging modes. Thus, the final result is 4 possible sets of communication strategies.

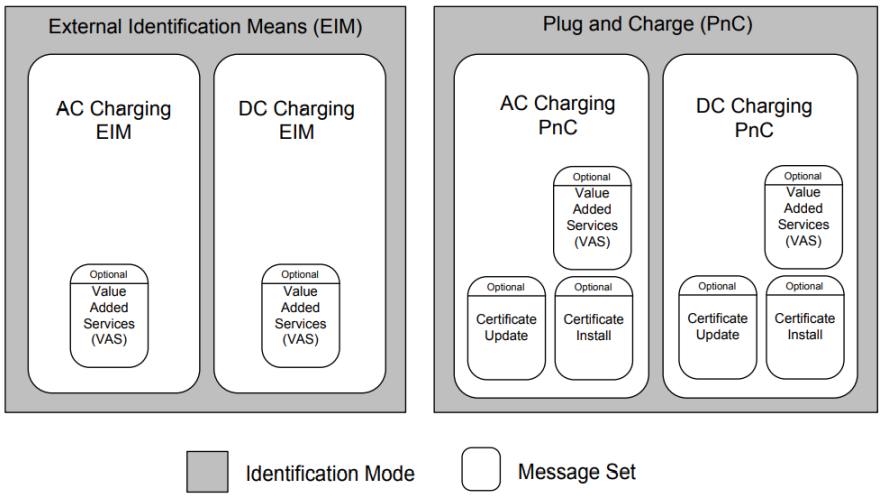


**Figure 3.10:** Overview on Identification Modes and Message Sets.

The AC/DC Charging mode is mandatory by the EV type, the choice is not up to the driver unlike the use of PnC or EIM which is a user's free choice.
The message Sets expected for each of the 4 situations are mostly the same with

some exceptions. The interest of this project falls on 3 of them: *CertificateInstallationReq/Res*, CertificateUpdateReq/Res, *AuthorizationReq/Res*.
The VAS is excluded since in this protocol version it has no message or its own Message Set.

## 3.3 Base of Plug and Charge

Before talking in detail about the Message Sets for the Plug%Charge, shall be done an overview of what this new feature offers and how it works from the user's point of view.

### 3.3.1 What does PnC usage entail

Compared with the modern charging operation, PnC impacts at different levels, experiencing benefits in several processes of the charging session:

- identification:
  the EV driver shall only plug the cable into the Charging Station without showing any type of card/code. The personal credentials for the recognition and permission to start the charging session are taken directly from the vehicle using a cross-check of the respective certificates.
  From the user's point of view, this identification modality offers a faster identification phase and makes him to avoid the handle of a further card. Moreover, it prevents digital bugs from happening during the operation.

- billing handle:
  as happens for the first step-phase, also the billing handle is carried over autonomously, affecting the user with the previously quoted benefits.

- data security:
  confidential data are kept private and out of reach of external parties. No uninvited reader and no data manipulation are allowed thanks to the cryptographic

tools of the PKI-based security protocol required by ISO 15118.

- smart grid implementation:
  PnC fully integrates the smart grid technology features; so it can boast also all the advantages that came out with smart charging.

To work properly, Plug&Charge needs the execution of 2 Messages well-defined in the ISO 15118-2 document. They are referred to as *CertificateInstallationReq/Res* and *CertificateUpdateReq/Res*, handling respectively the installation and the update of the contract certificate.

### 3.3.2 Certificate Installation

The installation of a new contract certificate results necessary if, for example, the old certificate is expired, revoked or simply there is none stored inside the EV.
The SECC may have to request the certificate from a SA and they may have to be created by this SA. After the installation of this certificate, the charging process at the EVSE may start.
As it applies to every message, a request and a response message can be distinguished and they will be analyzed separately.

#### *CertificateInstallationReq*

By sending the *CertificateInstallationReq* the vehicle requests the SECC to deliver the certificate that belongs to the currently valid contract of the vehicle.
The Body message contains 2 fields:

- **OEMProvisioningCert**:
  contains an EV-specific certificate that was earlier installed in the EVCC typically by an OEM. It is used to identify the currently valid contract of the EV. It is transmitted in Base64 to encrypt the message later on.

- **ListOfRootCertificateIDs**:

contains a list of Certificate IDs belonging to all the Root Certificates currently installed in the EV.

The *CertificateInstallationReq* shall be signed by its own OEM Provisioning Certificate and then sent to the SECC.

#### *CertificateInstallationRes*

After receiving the Request, the SECC sends the *CertificateInstallationRes* including the requested certificate. Then, the EVCC installs this certificate. There is only one certificate delivered to the EVCC: the one for signing messages. It belongs to the currently valid contract of the EV.
The elements contained in the body message are the following:

- **SAProvisioningCertificateChain**:
  the certificate chain used by the EVCC to verify the signature in the message header.

- **ContractSignatureCertChain**:
  new certificate chain for signature purposes that has to be installed in the EVCC.

- **ContractSignatureEncryptedPrivateKey**:
  the private key that belongs to the new certificate for signature purposes. It has to be installed in the EVCC as well.
  Since it is secret data, it shall be encrypted by the element identified using the provided OEM Provisioning Certificate of the EVCC and using the calculated DH secret for encryption.

- **DHpublickey**:
  Diffie Hellman public key from the SA for generating the session key at the EVCC to encrypt the Contract Signature Private Key at the SA and decrypt it at the EVCC.

- **eMAID**:
  also called ContractID, is the element identifying the charging contract.

The CPS shall sign the 4 last body elements using the Provisioning Service Certificate. The EVCC shall verify the signature of the *CertificateInstallationRes* using the signer certificate chain SAProvisioningCertificateChain, validate said chain, and ensure that it traces back to a valid V2G Root Certificate.
The ContractSignatureCertChain received in the Response Message shall be stored persistently in such a way, that it can be applied later on to verify tariff tables.

### 3.3.3 Certificate Update

Updating the certificate to the EVCC is required when the certificate is still valid, but is about to expire. In this use case, the EVCC requests a new contract certificate from the SECC to be installed into the EVCC.
The request and the response message shall be implemented in the update sequence.

#### *CertificateUpdateReq*

In this message, the vehicle requests the SECC to deliver a new contract certificate that belongs to the currently valid contract of the vehicle.
The elements of the message are the following.

- **ContractSignatureCertChain**:
  contains the currently available signature certificate including the certificate chain in the EVCC. The SECC uses this certificate chain to check the message signature included in the header of the message.

- **eMAID**

- **ListOfRootCertificateIDs**

### *CertificateUpdateRes*

After receiving the *CertificateUpdateReq*, the SECC retrieves the requested certificate from the SA. Therefore, it needs to establish an online connection. Then the SECC sends the *CertificateUpdateRes* including the new contract certificate to the EVCC. Finally, the EVCC installs this certificate.
The body message encloses the following elements; they are the same as in *CertificateInsallationRes*.

- **SAProvisioningCertificateChain**
- **ContractSignatureCertChain**
- **ContractSignatureEncryptedPrivateKey**
- **DHpublickey**
- **eMAID**

The procedure is the same as in the Certificate Update case, so it is the encryption by CPS and the signature verification and storing action that needs to be done by the EVCC after it receives the message.

# Chapter 4

# ISO15118-PoC: hardware and software implementation

In this chapter, the high-level software architecture will be analyzed with a brief overview of each unit. The hardware components used to make the simulation between the EV and EVSE will be also reported. The project consists of a Proof-of-Concept (PoC) of a full charging sequence ruled by ISO15118-2 standard focused on the Plug and Charge feature and with particular attention to the certificate handling.

## 4.1 General Demo organization

In its final aspect, the resulting PoC has been implemented as a Demo of how effectively works a charging session for real, or better how could work if PnC would be implemented. The charging process has been developed under both points of view of the end customer (EV) and the Charging Station (EVSE).

### 4.1.1 High-Level Architecture

The PnC-Demo consists of 2 firmware implemented on 2 hardware sets, one simulating the Electric Vehicle infotainment and the second one simulating the Charging Point.
The 2 firmware are mirrored if analyzed with a high point of view. In fact, each of them is based on a central script, working as a hub for sorting and handling data, and 4 satellite scripts, each equipped with specific functionalities (Figure 4.1).

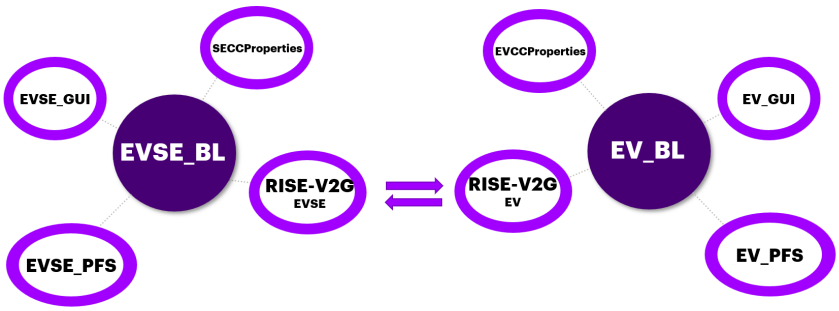


**Figure 4.1:** Demo software general structure.

Each code unit is assigned to play a specific role:

- **EV/EVSE-BL (Business Logic)**:
  acts as the brain of the whole system. It is responsible for handling data coming in and out of the system, coordinating and managing the satellite units.

- **EV/EVSE-GUI (Graphical User Interface)**:
  makes the Demo usable and interactive with the human tester. It is tasked with taking the input commands and values and showing the output information to the user.

- **EV/EVSE-PFS (Power Flow Simulator)**:
  handles all the data about involved energy values, parameters defining the generation of the charging profiles and their expression on a Cartesian diagram.

- **EVCC/SECC-Properties**:
  contains all the global variables used by the python units.

- **RISE-V2G (EV/EVSE)**:
  Java open-source stack protocol ruled by ISO 15118-2 developed by the Switch Company. It handles the actual communication and message exchange between SECC and EVCC.

### 4.1.2 Hardware components

The software execution has been tested on the laptop at first and then it has been carried on hardware supports such that the simulation may have a more realistic footprint.
For this scope, each software pack is integrated on a Raspberry Pi 4, one simulating the EVCC and the other one the SECC. In the same way, the Graphical User Interface is displayed to the tester through a couple of display touchscreen mounted on the RPIs assembled with a Raspberry case.
In order to implement the External Identification Mode, as an alternative for the PnC, on the EVSE side, a Raspberry Camera module must be integrated such that a simulated RFID or QR code can be visualized and recognized by the SECC.
Finally, the usual cable connecting the electric vehicle to the Charging Point is simulated using an Ethernet cable that must be plugged into the RPIs at the beginning of the charging process.

**Figure 4.2:** Hardware components assembled.

## 4.2 Java software components

The Java open-source stack protocol, called RISE-V2G, si a runnable program that emulates the communication session between an EV and a Charging Point according to ISO15118-2 standard protocol. It works as a starting point to build the entire simulation. In fact, every Python unit has been added afterward reasoning mainly on the working principles of the stack protocol, although in the final result, it shall be read as a simple satellite unit.

Inside the firmware, RISE-V2G covers the role of the communication unit. It applies the ISO15118-2 protocol setting up the communication server-client (SECC-EVCC), handling all the expected parameter values, messages and their signatures.

### 4.2.1 RISE-V2G: original...

RISE-V2G has been programmed with the 2 "main" scripts called "*StartSECC*" and "*StartEVCC*" each of which is encapsulated inside its corresponding folder, described more in detail below. This program has been designed also for running

separately and without using any IDEs. It is possible to run the EVCC and the SECC in a real demonstration of messages exchange since RISE-V2G has been created as Maven project, and it allows the generation of .jar files to be executed. As RISE-V2G was thought by their creators, it provides 5 leading folders. They are the following:

- **RISE-V2G-Certificates**
- **RISE-V2G-EVCC**
- **RISE-V2G-PARENT**
- **RISE-V2G-SECC**
- **RISE-V2G-Shared**

The structure of each folder is similar for all. This statement is especially true for the EVCC and SECC folders that are almost mirrored from a high-level point of view and it makes exceptions Certificates folder.

### RISE-V2G-Certificates

RISE-V2G-Certificates has been created with the scope of simulating the certificate generation and distribution described in the VDE-AR-E 2802-100-1 application guide; obviously, the majority of the flow chart branches, in which the process is described, are cut-off from this emulation. This shortage happens because all the different Companies, other than a Roaming platform and a CSMS, would be needed if a full and literal implementation is made. Thus, the RISE-V2G-Certificates folder generates all the certificate chains and the respective key pairs.
It presents a completely different range of files if compared with the other folders:

- **passphrase.txt**:
  contains a password to open and view the certificate chain once it is generated.

- **configs** sub-folder:
  contains a list of .cnf files for every necessary certificate that shall be generated. They contain all the information the certificate must contain.

- **generateCertificates.bat** and **generateCertificates.sh**:
  batch and shell file to be used on their respective OS for generating all the certificate chains and the cryptographic key pairs.
  At the beginning of the file, the validity period of each certificate is set.

- **copyNewCertsAndKeys.bat** and **copyNewCertsAndKeys.sh**:
  batch and shell file to be used on their respective OS to copy the generated certificate chains and their keys and paste them into the appropriate directory.

The operation of generateCertificates.bat, or generateCertificates.sh, is the focus of the RISE-V2G-Certificate folder. The output of this executing file is the following sub-folders:

- csrs:
  contains all the certificates taken from the "configs" folder in a .csr format.

- certs:
  contains all the certificates in .der, .pem and .p12 format. The reason behind the necessity to have all these format files is due to a conversion issue; the ISO15118-2 protocol requires its certificates to be stored as .der files.

- keystores:
  contains the OEM and CPO certificate chains in .p12 format, and the containers into which the certificate chains and the keys are stored in the EVCC and SECC. They are called *evccKeystore.jks*, *evccTruststore.jks*, *seccKeystore.jks* and *seccTruststore.jks*

- privateKeys:
  contains the private keys associated to each certificate.

RISE-V2G-Certificates makes an exception for what concerns a specific file contained in all the other folders. This file is called ***pom.xml***. It includes all the dependencies and information for generating the .jar files.

### RISE-V2G-PARENT

RISE-V2G-PARENT creates all the necessary .jar files and settings needed for the IDEless execution. It contains the "pom.xml" file that allows generating the target folder respectively in RISE-V2G-EVCC and RISE-V2G-SECC only using a single command. This folder works like a shortcut that avoids the user to browse through folders and generate the target folder in loco.

### RISE-V2G-Shared

The remaining 3 folders are the containers of the actual simulation program. All the 3 of them collect 227 java files. Actually, the majority of these scripts are stored in the RISE-V2G-Shared folder. Analyzing more in detail this latter one includes all the classes, methods and schemas that are used by either EVCC and SECC. In fact, the "Shared" name gets already an idea of what is inside it. It can be assumed that this folder stores all the scripts defining classes and methods like "raw materials" such that they can be adapted to the context.
Due to a large number of scripts and especially the complexity of the code, there will be analyzed only the java files that allows an understanding of the program's working flow of the program.

Watching directly inside RISE-V2G-Shared, "pom.xml", "LICENSE.txt" are located, which are already explained, together with **src** folder. Inside it starts a directory common between the "Shared", "EVCC" and "SECC" folders; it is the working directory where all the Java scripts are actually stored. The working directory is the following one: "**\src\main\java\com\v2gclarity\risev2g\X**".

The "X" sub-folder takes its name from the top-level folder; then it could be "evcc", "secc" or "shared", in this case.
The directory described above contains the following sub-folders:

- **enumerations**:
  group and define within itself all the enum types that will e used in the program. they have defined in the following scripts:
  - CPStates.java
  - GlobalTypes.java
  - **GlobalValues.java**: defines the alias for the certificate chain and for the stores previously characterized.
  - MessageSets.java
  - PKI.java
  - **V2GMessages.java**: defines the alias for every name of the messages exchanged between EVCC and SECC, whether it is a Request and Response.
- ExiCodec:
  contains all the methods and classes used for encrypting and decrypting EXI messages.
- messageHandling:
  defines some actions that can affect the state of the session, like a Pause, a Termination, etc.
- **misc**:
  stores the basic JAVA files used for the instances regarding sessions, states and messages structure. They are:
  - **State.java**: set the State class and its basic methods. In particular, the following are the most interests for our case:

```
1       import ...
2
3       public abstract class State {
4
5       private Logger logger = LogManager.getLogger(this.
    getClass().getSimpleName())
6       private State nextState = null;
7       private MessageHandler messageHandler;
8       private V2GCommunicationSessiocommSessionContext;
9       private HashMap<String, byte[]>
    xmlSignatureRefElements;
10      private ECPrivateKey signaturePrivateKey;
11
12      public State(V2GCommunicationSession
    commSessionContext) {
13      setCommSessionContext(commSessionContext);
14      setMessageHandler(commSessionContext.
    getMessageHandler());
15      setXMLSignatureRefElements(new HashMap<String, byte
    []>());
16      }
17
18      public abstract ReactionToIncomingMessage
    processIncomingMessage(Object message);
19
20      public SendMessage getSendMessage(
21              BodyBaseType message,
22              V2GMessages nextExpectedMessage) {
23      int timeout = getTimeout(message, nextExpectedMessage
    );
24      return getSendMessage(message, nextExpectedMessage, "
    ", timeout);
25      }
26
```

```
27          public SendMessage getSendMessage(
28              BodyBaseType message,
29              V2GMessages nextExpectedMessage,
30              String optionalLoggerInfo) {
31          int timeout = getTimeout(message, nextExpectedMessage
    );
32          return getSendMessage(message, nextExpectedMessage,
    optionalLoggerInfo, timeout);
33      }
34
35          ...
36
37
```

– TimeRestrictions.java

– **V2GCommunicationSession.java**: set the communication Session class and its methods, like the logger, etc.

```
1           import ...
2
3           public abstract class V2GCommunicationSession extends
     Observable {
4
5       private Logger logger = LogManager.getLogger(this.
    getClass().getSimpleName());
6       private HashMap<V2GMessages, State> states;
7       private State currentState;
8       private State startState;
9       private MessageHandler messageHandler;
10      private byte[] sessionID;
11      private V2GTPMessage v2gTpMessage;
12      private V2GMessage v2gMessage;
13      private boolean tlsConnection;
14
```

```
15      public V2GCommunicationSession() {
16          setStates(new HashMap<V2GMessages, State>());
17          setMessageHandler(MessageHandler.getInstance());
18          setSessionID(null);
19          setV2gTpMessage(null);
20      }
21          ...
22
```

The abstract class V2GCommunicationSession takes advantage of the Observable library in order to be sure the session is notified when a certain event triggers it.

- V2GImplementationFactory.java
- V2GTPMessage

- **utils**:
  includes 4 Java scripts that do not have actually anything in common, but they become very useful at a lot of points along the code.
  - ByteUtils.java: set all the functions that deal with the conversion from or to byte type.
  - **MiscUtils.java**: one of the most important scripts in the whole RISE-V2G pack. It is responsible for handling all the parameters involved in the EV-EVSE charging simulation required by ISO15118. It accomplishes its task by building a *get()* method exploiting the java.util.Properties library.

```
1       import ...
2
3       public static Object getPropertyValue(String
    propertyKey) {
4       Object returnValue = null;
5       String propertyValue = "";
```

```
6
7           try {
8               propertyValue = getProperties().getProperty(
    propertyKey).replaceAll("\\s", "");
9           } catch (NullPointerException e) {
10              getLogger().warn("No entry found in the
    properties file for property '" + propertyKey + "'", e);
11              return null;
12          }
13          switch (propertyKey) {
14          case "network.interface": // EV + EVSE property
15              returnValue = propertyValue;
16              break;
17          case "session.id": // EV property
18              returnValue = propertyValue;  // a hexadecimal
    string representing a byte array
19              break;
20          ...
21          default:
22              getLogger().error("No property with name '" +
    propertyKey + "' found");
23          }
24          return returnValue;
25          }
26          public static Properties getProperties() {
27          return properties;
28      }
29
```

This Java script also owns other secondary methods developed for reading a file storing the simulation parameters, storing the parameter into a file and determining the link-local IPv6 address, set the port for UDP, TCP and TLS connection.

– **SecurityUtils.java**: the most important file-storing tools concerning

digital certificates, encryption keys and signatures. The main actions it has to cover are the following:

  * **get** every type of certificate, certificate chain, certificate attributes (eMAID, ...) public or private key and Signature
  * **save** the contract certificate chain
  * **generate** digests, signatures and encryption keys.
  * **encrypt** and **decrypt**
  * **verify** the certificates' validity, signatures and if a certificate installation or update is needed.

- SleepUtils.java

- **v2gMessages**:
  it contains 2 files called "SECCDiscoveryReq.java" and "SECCDiscoveryRes.java" which initialize some variable attributes of the messages read and sent by the SECC. Moreover, it includes 2 additional sub-folders. One is called apProtocol, but "msgDef", the second one, is what really matters for this analysis.
  Inside this sub-folder are set all the simple and complex types described in the ISO15118-2 protocol which is the basis of every state, message and message field. Thus, it builds their general structure by setting protected attributes, defining them also as XML elements, and defining get and set methods. The CertificateInstallationType is reported as an example:

```
1  import ...
2
3  @XmlAccessorType(XmlAccessType.FIELD)
4  @XmlType(name = "CertificateChainType", propOrder = {
5  "certificate",
6  "subCertificates"
7  })
8
```

```
9     public class CertificateChainType {
10    @XmlElement(name = "Certificate", required = true)
11    protected byte[] certificate;
12    @XmlElement(name = "SubCertificates")
13    protected SubCertificatesType subCertificates;
14    @XmlAttribute(name = "Id", namespace = "urn:iso:15118:2:2013:
      MsgDataTypes")
15    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
16    @XmlID
17    @XmlSchemaType(name = "ID")
18    protected String id;
19
20    public byte[] getCertificate() {
21        return certificate;
22    }
23    public void setCertificate(byte[] value) {
24        this.certificate = value;
25    }
26    public SubCertificatesType getSubCertificates() {
27        return subCertificates;
28    }
29    public void setSubCertificates(SubCertificatesType value) {
30        this.subCertificates = value;
31    }
32    public String getId() {
33        return id;
34    }
35    public void setId(String value) {
36        this.id = value;
37    }
38    }
39
```

The files developed inside the RISE-V2G-Shared are purposely designed without characterization since they shall work as a template, a workframe able to be adapted

to the EVCC and SECC needs. Intuitively, implementing RISE-V2G-SECC and RISE-V2G-EVCC is as similar as possible such that the characterization of the classes and methods in the Shared folder result is more feasible.

### RISE-V2G-SECC and RISE-V2G-EVCC

At the first level of SECC and EVCC, the files are basically the same, without considering the values inside. They contain:

- pom.xml
- LICENSE.txt
- **SECCConfig.properties/EVCCConfig.properties**: .properties file containing global parameters of SECC and EVCC respectively. They work like a key-value list to which Miscutils.java allows access to get and set the values.
- "**`\src\main\java\com\v2gclarity\risev2g\X`**": The "X" sub-folder is referred to "secc" for the SECC and "evcc" for the EVCC. It is the folder where the code parts characterizing the 2 controllers are stored.

The structure of the sub-folder containing the scripts is again almost the same. A single analysis can be conducted, without treating them separately, making some exceptions.
The following sub-folders are organized in the following folder-java files structure:

- **evseController/evController**: contains files aimed at managing controller parameters.

| SECC | EVCC |
|---|---|
| DummyACEVSEController.java | DummyEVController.java |
| DummyDCEVSEController.java | |
| IACEVSEController.java | IACEVController.java |

| SECC | EVCC |
|---|---|
| IDCEVSEController.java | IDCEVController.java |
| IEVSEController.java | IEVController.java |

**Table 4.1:** evseController and evController folder structure and content.

The "Dummy" files create the class structure and its attributes while the "I" files deal with the set() and get() methods for them.

- **main**: contains the runnable file that allows starting the charging session.

| SECC | EVCC |
|---|---|
| StartSECC.java | StartEVCC.java |

**Table 4.2:** main folder structure and content.

They follow 3 steps. At first, initialize the logger manager. Then, a load function is called on the MiscUtils.GlobalValues.java file in order to set the correct path for the Config.properties files. Finally, they start the UDP, TCP, TLS server and clients as Threads creating a new Communication Session.

- **misc**:

| SECC | EVCC |
|---|---|
| SECCImplementationFactory.java | EVCCImplementationFactory.java |

**Table 4.3:** misc folder structure and content.

They aim to create the BackendInterface, the respective "I" Controller for SECC and EVCC applied the current Communication Session.

- **session**: contains files aimed at defining the Communication Session specific to each controller and all of its attributes and methods.

| **SECC** | **EVCC** |
|---|---|
| V2GCommunicationSession-HandlerSECC.java | V2GCommunciationSession-HandlerEVCC.java |
| V2GCommunicationSessionSECC.java | V2GCommunicationSessionEVCC.java |

**Table 4.4:** session folder structure and content.

They are the Java scripts fit to set up the specific Communication Session for the specific controller and create the appropriate methods() needed by the Session to handle its data.

- **states**:

| **SECC** | **EVCC** |
|---|---|
| ForkState.java | / |
| ServerState.java | ClientState.java |
| WaitForAuthorizationReq.java | WaitForAuthorizationRes.java |
| WaitForCableCheckReq.java | WaitForCableCheckRes.java |
| WaitForCableCheckReq.java | WaitForCableCheckRes.java |
| WaitForCertificateInstallationReq.java | WaitCertificateInstallationRes.java |
| WaitForCertificateUpdateReq.java | WaitForCertificateUpdateRes.java |
| WaitForChargeParameterDiscoveryReq.java | WaitForChargeParameterDiscoveryRes.java |
| WaitForChargingStatusReq.java | WaitForChargingStatusRes.java |
| WaitForCurrentDemandReq.java | WaitForCurrentDemandRes.java |
| WaitForMeteringReceiptReq.java | WaitForMeteringReceiptRes.java |
| WaitForPaymentDetailsReq.java | WaitForPaymentDetailsRes.java |
| WaitForPaymentServiceSelectionReq.java | WaitForPaymentServiceSelectionRes.java |
| WaitForPowerDeliveryReq.java | WaitForPowerDeliveryRes.java |
| WaitForPreChargeReq.java | WaitForPreChargeRes.java |
| WaitForServiceDetailReq.java | WaitForServiceDetailRes.java |

| SECC | EVCC |
|---|---|
| WaitForServiceDiscoveryReq.java | WaitForServiceDiscoveryRes.java |
| WaitForSessionSetupReq.java | WaitForSessionSetupRes.java |
| WaitForSessionStopReq.java | WaitForSessionStopRes.java |
| WaitForSupportedAppProtocolReq.java | WaitForSupportedAppProtocolRes.java |
| WaitForWeldingDetectionReq.java | WaitForWeldingDetectionRes.java |

**Table 4.5:** states folder structure and content.

They define all the possible States the Communication Session may join. For each state, the content and the operation code are different.

- **transportLayer**: contains files aimed at managing controller parameters.

| SECC | EVCC |
|---|---|
| ConnectionHandler.java | / |
| StatefulLayerServer.java | StatefulLayerClient.java |
| TCPServer.java | TCPClien.java |
| TLSServer.java | TLSClient.java |
| UDPServer.java | UDPClient.java |

**Table 4.6:** transportLayer folder structure and content.

They are concerned with message handling. They work like receivers and senders of messages. Moreover, inside the transportLayer sub-folder, the UDP, TCP and TLS server and client are defined.

In addition to the folders described above, RISE-V2G-SECC contains a further sub-folder called "backend". It contains 2 files:

- **DummyBackendInterface**: load Secondary Actor (SA) Schedule List containing parameters of all the simulated charging profiles and the simulated certificate chains.

- **IBackendInterface**: load get() and set() methods applied on the simulated charging profiles and certificate chains.

The "backend" sub-folder works like a really simple simulation environment of CPO, eMSP and Roaming Platform, more in general, every actor upstream of the Charging Point. In fact, the simulated backend plays 2 main roles: handles the certificate chains and their movements behind the EVSE and loads the parameters needed to build the charging profiles.

### 4.2.2 RISE-V2G: ...and developments

RISE-V2G taken as a "raw" stack protocol works in terms of the messages exchanged and all the functioning machines that the ISO15118-2 requires to be applied. However, it remains a pure simulation with some deficiencies. First of all, it does not give the possibility for the user to interact with it, but the charging process is carried out automatically. Moreover, this automatization implies the inability of choosing and changing parameters during the process, whether it would be in the authorization, pre-charging and charging phases.
In order to overcome these problems some code developments have to be made. In particular, following the high-level architecture already explained, what is missing in the Java program is a local communication interface with the ability to exchange real-time information with the Python central unit such that RISE-V2G is decentralized and used only as the communication interface between SECC and EVCC.
The solution which better fixes the issue is the creation of a socket able to send and receive data.
The socket in question is implemented in the Java part and is recognized as the client. Its role is to connect to the Python socket Server sending data about the current state and the useful parameter's value and reading the returning data about the updated parameters value.
The Java socket client [12] is initialized with the following lines:

```
1
2     public static void Client(String CurrentState) {
3         Socket rpiSocket = null;
4         DataInputStream in = null;
5         PrintStream out = null;
6         String hostName = "localhost";
7         int portNumber = 5560;
8
9         try {
10            rpiSocket = new Socket(hostName,portNumber);
11            out = new PrintStream(rpiSocket.getOutputStream());
12            in = new DataInputStream(new BufferedInputStream(rpiSocket
    .getInputStream()));
13        } catch (UnknownHostException e) {
14            System.err.println("Don't know about host: hostname");
15        } catch (IOException e) {
16            System.err.println("Couldn't get I/O for the connection to
    : hostname");
17        }
```

The host port number is set arbitrarily to 5560 for the SECC client and 5561 for the EVCC client, but otherwise, the rest of the code is exactly the same.
The input information consists of a simple String type. Client class takes the State in which the Controller is currently in and it is the first parameter that will be passed every time to the Python server.
The next step is to define under which structure the data are gathered in the message. The JSON format is chosen since it is compact and easy to write. For this purpose, the "gson" library by Google is used. The JSON object is used only for generating the message and reading its response fields, but what is actually exchanged in the server-client communication is a String representing the JSON. The code setting up the JSON object is the following:

```
    // create the JSON object
    JsonObject obj = new JsonObject();
    obj.addProperty("key_1", "value_1");
    obj.addProperty("key_2", "value_2");
    obj.addProperty("key_3", "value_3");

    // convert the JSON object to a string and send it to the server
    Gson gson = new Gson();
    String jsonStr = gson.toJson(obj);
    out.println(jsonStr);
```

The key-value pairs are taken from the respective Config.properties file on which additional parameters are added to be able to set them and modify their values in a more comfortable and flexible way. Consequently also the MiscUtils method "getpropertyValue(...)", previously described, must be integrated in order to take the new parameter values.
The first key-value pair is always represented by the CurrentState and its name is taken from the homonymous parameter. The successive JSON fields change either in terms of key-value and in terms of quantity. It depends on the current State. For this reason, it is mandatory to set an if-else if condition list which works as a sorter such as different key-value pairs can be set for each message sent.

```
switch(CurrentState){
    case "WaitForPaymentServiceSelectionRes":
        obj.addProperty("CurrentState", CurrentState);
        obj.addProperty("key_2", (String) getPropertyValue("key_2"));
        obj.addProperty("key_3", (String) getPropertyValue("key_3"));

        // convert the JSON object to a string and send it to the
    server
        Gson gson = new Gson();
```

```
        String jsonStr = gson.toJson(obj);
        out.println(jsonStr);

    ...
```

The second Java client's duty is to read the response coming from the Python server. As it was done before, again in this case, the necessity to set an if-else condition list is required to read correctly the JSON content. Inside every if(), the String response is converted back into a JSON Object and the response parameter is updated in the Config.properties file from the key-value pairs.

```
    byte[] bytes = new byte[1024];
    in.read(bytes);
    String reply = new String(bytes, "UTF-8");

    Gson gson = new Gson();
    JsonObject jsonObject = gson.fromJson(reply.trim(), JsonObject.
    class);

    if (jsonObject.get("CurrentState").getAsString().equalsTo("
    WaitForPaymentServiceSelectionReq")){
    getProperties().setProperty("Config_key_2", jsonObject.get("key_2
    ").getAsString());
    } else if(...){
    ...
```

The Client class is set up in the MiscUtils.java file and it is called every time a new message is received by the controller. In this way, the parameters contained in the Config.properties file are updated at their newest version, in case the upcoming message modifies them, and, on the other hand, their values can still be taken and changed externally by the user through the socket before sending a response message.

This logic mechanism is applied both for SECC and EVCC states, but there is an

exception. In the EVCC state called "WaitForChargeParameterDiscoveryRes", the received corresponding message contains the charging profiles generated by the EVCC, but they are in form of an XML and they are too long to be passed using a String anyway. In this case, the XML is saved as an XML file and analyzed in the EV_PFS using a Python library.

## 4.3 Python software components

The high-level architecture of the Python units contains the homonymous Python files. Then, the resulting code files show in the following structure:
**EVSE/EV_GUI –> EVSE/EV_GUI.py**
**EVSE/EV_PFS –> EVSE/EV_PFS.py**
**EVSE/EV_BL –> EVSE/EV_BL.py**
**SECC/EVCC_Properties –> SECC/EVCC_Properties.py**

The Python code structure is the same for both the SECC and the EVCC. Thus, the code will be analyzed only one time in general for both the controllers and, in case exceptions exist, they are deepened apart, as it has been done for the RISE-V2G.

### 4.3.1 Graphical User Interface (GUI)

The Python unit responsible for the Graphical User Interface is the EVSE/EV_GUI.py. It stimulates the infotainment on the EV and the information display at the Charging Point. It plays 3 roles:

- makes the progress of the charging session viewable and easy-understandable to the user.

- gathers all the inputs from the user in the decision-making charging steps and communicates them to the Central unit (BL).

- keeps the user up to date on the background action and the exchanged charging parameters of the current charging session.

The Python library used for achieving these objectives is called customtkinter. It is the upgraded version of the widespread Tkinter library. It adds new features and improves the already existing ones in addition to making the interface more visually appreciable. Furthermore, the Image and ImageTk packets of the pillow library have been imported in order to manage pictures.

The code structure has been thought of such that every page has its own class with its own methods and attributes. However, the GUI is required to constantly display a page without interruptions during page transitions.
To achieve this purpose, a "main" frame is set up and then passed to the page class that needs to be opened:

```
class main_page(customtkinter.CTk):
    WIDTH = 800
    HEIGHT = 400
    customtkinter.set_appearance_mode("dark")
    customtkinter.set_default_color_theme("blue")

def __init__(self):
    super().__init__()
    self.title('EVCC')
    self.geometry((f"{main_page.WIDTH}x{main_page.HEIGHT}"))
    zero_page(self)
```

The "main_page" owns default attributes that every page must have as page dimensions, background color and word colors. In its ___init___(self) method, title and page geometry are fixed and a default page is called passing as a parameter the "main_page" itself.
Besides the structure, each page class expects an argument to be passed when it is

called. This is the initially created "main_page" frame.

The page structure follows the logic dictated by the customtkinter library. It plans to take the default main page received as an argument and divide it into frames, so creating a kind of grid table. Then, every frame is further divided into sub-frames. The result is a template page consisting of a grid table with X frames and each frame is sub-divided into a grid with different numbers of rows and columns. The possibility to create an uneven table gives a high level of freedom from a design point of view since it allows inserting widgets in different positions along the page. Below is reported the example of the page that will open if PnC as payment option method is chosen:

```
    class PnC_page():
    def __init__(self, window):
        self.PnC_page(window)

    def PnC_page(self, window):
        # CREATION OF FRAMES 2-left,1-right (2x2) #
        window.grid_rowconfigure(0, weight=1)
        window.grid_columnconfigure(1, weight=1)

        window.frame1 = customtkinter.CTkFrame(master=window,
corner_radius=15)
        window.frame1.grid(row=0, column=0, columnspan=2, padx=10,
pady=10, sticky="nswe")

        # frame #
        window.frame1.grid_rowconfigure(12, weight=1)
        window.frame1.grid_columnconfigure(0, weight=1)

        # empty rows with minsize as spacing
        window.frame1.grid_rowconfigure(0, minsize=10)
        window.frame1.grid_rowconfigure(3, minsize=5)
        window.frame1.grid_rowconfigure(4, minsize=20)
```

```
21          window.frame1.grid_rowconfigure(5, minsize=20)
22          window.frame1.grid_rowconfigure(7, minsize=10)
23          window.frame1.grid_rowconfigure(8, minsize=20)
24          window.frame1.grid_rowconfigure(9, minsize=10)
25          window.frame1.grid_rowconfigure(10, minsize=10)
26          window.frame1.grid_rowconfigure(11, minsize=10)
27
28          ...
```

The noteworthy widgets mostly used in the GUI script are the CTkLabel and CTkButton. The first one is widespread along the code since it allows appending tags either if they are passive and images. The second widget allows the interaction between the controller and the user. In fact, it accepts one or more functions that will be activated when the button will be pressed by the user. A further interactive widget is the CTkComboBox. Unlike what happens to push the button, it offers a selection.

```
1     ...
2     def button_1(self):
3         EVCC_Properties.Chosen_SA_schedule = 1
4         EVCC_Properties.RiseContinue = True
5     ...
6     def charging_profile(self, window):
7         ...
8         PATH1 = 'C:/Users/'+str(EVCC_Properties.UserDir)+'/Demo-PnC/
    workspace/DEMO_EVCC/ISO15118-Demo/ChargingProfile1.png'
9         PATH2 = 'C:/Users/'+str(EVCC_Properties.UserDir)+'/Demo-PnC/
    workspace/DEMO_EVCC/ISO15118-Demo/ChargingProfile2.png'
10        PATH3 = 'C:/Users/'+str(EVCC_Properties.UserDir)+'/Demo-PnC/
    workspace/DEMO_EVCC/ISO15118-Demo/ChargingProfile3.png'
11        profile1=ImageTk.PhotoImage(Image.open(PATH1).resize
    ((260,240)))
```

```
12          profile2=ImageTk.PhotoImage(Image.open(PATH2).resize
    ((260,240)))
13          profile3=ImageTk.PhotoImage(Image.open(PATH3).resize
    ((260,240)))
14
15          window.button1 = customtkinter.CTkButton(master=window.frame1
    , image=profile1, text="", text_font=("Roboto Medium", -15),
    corner_radius=50, fg_color="black", text_color="green4",
    border_width=1, border_color="green4", width=20, height=40,
    command=lambda: [self.button_1()])
16          window.button1.grid(row=2, column=1, padx=20, pady=10)
17
18          window.label1 = customtkinter.CTkLabel(master=window.frame1,
    text="Profile 1", fg_color="black", text_color="green4", text_font
    =("Roboto Medium", -30))
19          window.label1.grid(row=3, column=1, padx=10, pady=10)
20          ...
```

The last observation about the GUI script comes from some non-constant values. There are some labels, like the EV SoC, that must be updated constantly since they will change during the charging session. In the meantime, this requirement needs to be compliant with the real-time update requirement of the GUI. This is possible for the nature of how the customtkinter library has been thought of.
In general, the current page of a Graphical User Interface is built in such a way that it remains in an infinite loop such that the code structure to generate it will be executed continuously. It has a regeneration frequency rate so high that the page will result in static to an external point of view, even if it is constantly re-compiled. In this implementation, the Python library requires the "main_page", built using the "customtkinter.CTk", has to be initialized in the main loop of the runnable script.
This process will be described later when the Business Logic unit flow is analyzed. In terms of how the changing values have to be shown, this objective can be

achieved by self-calling the function suitable for showing these values.

```
1    def EV_ssoc(self, window):
2        window.ssoc = customtkinter.CTkLabel(master=window.frame1,
     text="Battery state of charge: " + str(EVCC_Properties.evresssoc)
     + "%", text_color="white", text_font=("Roboto Medium", -30))
3        window.ssoc.grid(row=10, columnspan=3, padx=10, pady=0,
     sticky="we")
4        window.ssoc.after(1000, lambda: [self.EV_ssoc(window)])
```

The only other Python script imported by the Python GUI unit is the Properties.py file. It is essential because every user action implies a modification of a global variable such that the edit can be communicated to the Central unit.

### 4.3.2 Power Flow Simulator (PFS)

The Power Flow Simulator (PFS) unit manages the energy parameters and the charging profiles. Unlike what happens with the GUI unit, the EVSE_PFS is structurally different from the EV one. In general, these units consist of a single class containing several methods according to the controller's specific needs.

#### EVSE_PFS

It covers 2 functions:

- **set the EVSE characteristic energy values**: they comprehend the nominal voltage, the current ripple, the maximum and minimum voltage, current and power values.
  These values are set in the SECC_Properties.py script as global variables.

- **generate the charging profiles**: the ISO15118 standard set some requirements that shall be satisfied in the creation of a charging profile. It is represented by a 2D Cartesian diagram in which the x-axis corresponds to the

time and the y-axis matches an ePriceLevel. The latter is an integer value that corresponds to a timely-variable energy price. The standard protocol does not allow real price values in the EVCC-SECC communication.
Each ePriceLevel is linked to other 3 parameters: relativePricePercentage, RenewableGenerationPercentage and CarbonDioxideEmission. Thus, each point provides 4 charging data at every time instant.
These values should be provided by the CPO backend which in turn should be generated by the eMSP based on the energy market. However, they are randomly simulated in the EVSE_PFS following a Gaussian curve.

```
        def gen_EPriceLevels(self):
        EPriceLevelsList = [1, 2, 3, 4]
        # random.choices(ePriceList, weights=(10,30,20,10),k=4)
        value = random.sample(EPriceLevelsList, k=4)
        EPriceLevels = value
        return EPriceLevels
    def gen_Renewables(self, mean, variance):
        VecRenewables = [0, 0, 0, 0]
        for i in range(0, 4):
            value = round(random.gauss(mean, variance))
            VecRenewables[i] = value
        return VecRenewables
    def gen_Carbon(self, mean, variance):
        VecCarbon = [0, 0, 0, 0]
        for i in range(0, 4):
            value = round(random.gauss(mean, variance))
            VecCarbon[i] = value
        return VecCarbon

```

One last, method is implemented to actually call the previous methods and

store the simulated values in the respective global variables to be passed to the EVCC.

The ISO15118 standard limits the number of charging profiles that can be proposed to the user to a maximum of 3 schedules.

```
1     def generate(self):
2         timeIntervals=round(int(SECC_Properties.departure_time)
      /4)
3         SECC_Properties.EVSE_VecTimes1=[0,timeIntervals*1,
      timeIntervals*2,timeIntervals*3]
4         SECC_Properties.EVSE_VecTimes2=[0,timeIntervals*1,
      timeIntervals*2,timeIntervals*3]
5         SECC_Properties.EVSE_VecTimes3=[0,timeIntervals*1,
      timeIntervals*2,timeIntervals*3]
6
7         # Charging Profile1
8         SECC_Properties.EVSE_VecEPriceLevels1=self.
      gen_EPriceLevels()
9         SECC_Properties.EVSE_VecRenewables1=self.gen_Renewables
      (50,10)
10        SECC_Properties.EVSE_VecCarbon1=self.gen_Carbon(5,1)
11
12        # Charging Profile2
13        SECC_Properties.EVSE_VecEPriceLevels2=self.
      gen_EPriceLevels()
14        SECC_Properties.EVSE_VecRenewables2=self.gen_Renewables
      (30,5)
15        SECC_Properties.EVSE_VecCarbon2=self.gen_Carbon(3,10)
16
17
18        # Charging Profile3
19        SECC_Properties.EVSE_VecEPriceLevels3=self.
      gen_EPriceLevels()
```

```
20          SECC_Properties.EVSE_VecRenewables3=self.gen_Renewables
    (70,10)
21          SECC_Properties.EVSE_VecCarbon3=self.gen_Carbon(4,5)
22
```

### EV_PFS

The number of functions that must be covered in the case of the EV Power Flow Simulator is a little higher than the previous one. They are again defined as methods in a single class. They are described as follows:

- **set the EV characteristic energy values**: they comprehend the charging mode, maximum, limit and target values of voltage, current and power, and the State of Charge. These values are set in the EVCC_Properties.py script as global variables. The State of Charge of the EV battery is simulated by simply taking a random Gaussian value when the EV arrives at the Charging Point.
- **read and store values of the charging profiles**: when the ChargeParameterDiscoveryRes is received by the EVCC, it is saved on the current repository as an XML file. It contains the 3 offered to charge schedules.

```
1     def readXML(self, SalesTariff):
2          VecTimes = [0,0,0,0]
3          VecEPriceLevels = [0,0,0,0]
4          VecPricePercentage = [0,0,0,0]
5          VecRenewables = [0,0,0,0]
6          VecCarbon = [0,0,0,0]
7          tree=et.parse('C:/Users/'+str(EVCC_Properties.UserDir)+'/
    Demo-PnC/workspace/DEMO_EVCC/ISO15118-Demo/'+str(SalesTariff))
8          root=tree.getroot()
```

```
9           SalesTariffID = root.find('{urn:iso:15118:2:2013:
    MsgDataTypes}SalesTariffID')
10          index=0
11          for SalesTariffEntry in root.findall('{urn:iso
    :15118:2:2013:MsgDataTypes}SalesTariffEntry'):
12
13              RelativeTimeInterval = SalesTariffEntry.find('{urn:
    iso:15118:2:2013:MsgDataTypes}RelativeTimeInterval')
14              start = RelativeTimeInterval.find('{urn:iso
    :15118:2:2013:MsgDataTypes}start')
15              VecTimes[index]=int(start.text)
16
17              EPriceLevel = SalesTariffEntry.find('{urn:iso
    :15118:2:2013:MsgDataTypes}EPriceLevel')
18              VecEPriceLevels[index]=int(EPriceLevel.text)
19
20              ConsumptionCost = SalesTariffEntry.find('{urn:iso
    :15118:2:2013:MsgDataTypes}ConsumptionCost')
21              startValue = ConsumptionCost.find('{urn:iso
    :15118:2:2013:MsgDataTypes}startValue')
22              Multiplier = startValue.find('{urn:iso:15118:2:2013:
    MsgDataTypes}Multiplier')
23              Unit = startValue.find('{urn:iso:15118:2:2013:
    MsgDataTypes}Unit')
24              Value = startValue.find('{urn:iso:15118:2:2013:
    MsgDataTypes}Value')
25
26              Cost = ConsumptionCost.find('{urn:iso:15118:2:2013:
    MsgDataTypes}Cost')
27              costKind=Cost.find('{urn:iso:15118:2:2013:
    MsgDataTypes}costKind')
28              amount=Cost.find('{urn:iso:15118:2:2013:MsgDataTypes}
    amount')
29              if (costKind.text == 'relativePricePercentage'):
30                  VecPricePercentage[index]=int(amount.text)
```

```
31              elif (costKind.text == 'RenewableGenerationPercentage
    '):
32                  VecRenewables[index] = int(amount.text)
33              elif (costKind.text == 'CarbonDioxideEmission'):
34                  VecCarbon[index] = int(amount.text)
35                  index += 1
36          self.memo_SalesTariffValues(SalesTariffID.text,VecTimes,
    VecEPriceLevels,VecPricePercentage,VecRenewables,VecCarbon)
37
38      def memo_SalesTariffValues(self, SalesTariffID, VecTimes,
    VecEPriceLevels, VecPricePercentage, VecRenewables, VecCarbon)
    :
39
40          if (SalesTariffID == '1'):
41              EVCC_Properties.EVSE_VecTimes1 = VecTimes
42              EVCC_Properties.EVSE_VecEPriceLevels1 =
    VecEPriceLevels
43              EVCC_Properties.EVSE_VecPercentages1 =
    VecPricePercentage
44              EVCC_Properties.EVSE_VecRenewables1 = VecRenewables
45              EVCC_Properties.EVSE_VecCarbon1 = VecCarbon
46
47          elif (SalesTariffID == '2'):
48              EVCC_Properties.EVSE_VecTimes2 = VecTimes
49              EVCC_Properties.EVSE_VecEPriceLevels2 =
    VecEPriceLevels
50              EVCC_Properties.EVSE_VecPercentages2 =
    VecPricePercentage
51              EVCC_Properties.EVSE_VecRenewables2 = VecRenewables
52              EVCC_Properties.EVSE_VecCarbon2 = VecCarbon
53
54          elif (SalesTariffID == '3'):
55              EVCC_Properties.EVSE_VecTimes3 = VecTimes
56              EVCC_Properties.EVSE_VecEPriceLevels3 =
    VecEPriceLevels
```

```
57              EVCC_Properties.EVSE_VecPercentages3 =
    VecPricePercentage
58              EVCC_Properties.EVSE_VecRenewables3 = VecRenewables
59              EVCC_Properties.EVSE_VecCarbon3 = VecCarbon
60
```

The xml.etree.ElementTree is imported. It allows to easily go back to take the child elements in the XML structure. For a clean and compact solution, the process of storing them in local variables is coded in a for loop. Once it is finished, a simple storing method is called. It works by operating on passing the charging schedule ID and the local variables in order to store them in the global parameters.

- **generate the piecewise functions**: it is needed a method for each schedule that may correlate the time instance to the correct value, returning it.

```
1     def f1(self, x):
2         if (x>=EVCC_Properties.EVSE_VecTimes1[0] and x<=
    EVCC_Properties.EVSE_VecTimes1[1]):
3             return float(EVCC_Properties.EVSE_VecEPriceLevels1
    [0])
4         if (x>EVCC_Properties.EVSE_VecTimes1[1] and x<=
    EVCC_Properties.EVSE_VecTimes1[2]):
5             return float(EVCC_Properties.EVSE_VecEPriceLevels1
    [1])
6         if (x>EVCC_Properties.EVSE_VecTimes1[2] and x<=
    EVCC_Properties.EVSE_VecTimes1[3]):
7             return float(EVCC_Properties.EVSE_VecEPriceLevels1
    [2])
8         if (x>EVCC_Properties.EVSE_VecTimes1[3] and x<=
    EVCC_Properties.departure_time):
9             return float(EVCC_Properties.EVSE_VecEPriceLevels1
    [3])
```

```
10
```

- **generate Cartesian graphs of the charging profiles**:

```
1     def show_profiles(self, PATH_1, PATH_2, PATH_3):
2         #Charging Profile 1
3         x1=numpy.linspace(0,EVCC_Properties.departure_time,1000)
4         y1=numpy.vectorize(self.f1)(x1)
5         matplotlib.use('agg')
6         fig_1 = pyplot.figure(1)
7         pyplot.plot(x1, y1)
8         pyplot.xlabel("Time")
9         pyplot.ylabel("EPriceLevels")
10        pyplot.text(0, EVCC_Properties.EVSE_VecEPriceLevels1
    [0]-0.1, "Renewables:"+str(EVCC_Properties.EVSE_VecRenewables1
    [0])+"\n"+"Carbon:"+str(EVCC_Properties.EVSE_VecCarbon1[0]))
11        pyplot.text(EVCC_Properties.EVSE_VecTimes1[1],
    EVCC_Properties.EVSE_VecEPriceLevels1[1]-0.1, "Renewables:"+
    str(EVCC_Properties.EVSE_VecRenewables1[1])+"\n"+"Carbon:"+str
    (EVCC_Properties.EVSE_VecCarbon1[1]))
12        pyplot.text(EVCC_Properties.EVSE_VecTimes1[2],
    EVCC_Properties.EVSE_VecEPriceLevels1[2]-0.1, "Renewables:"+
    str(EVCC_Properties.EVSE_VecRenewables1[2])+"\n"+"Carbon:"+str
    (EVCC_Properties.EVSE_VecCarbon1[2]))
13        pyplot.text(EVCC_Properties.EVSE_VecTimes1[3],
    EVCC_Properties.EVSE_VecEPriceLevels1[3]-0.1, "Renewables:"+
    str(EVCC_Properties.EVSE_VecRenewables1[3])+"\n"+"Carbon:"+str
    (EVCC_Properties.EVSE_VecCarbon1[3]))
14        pyplot.savefig(PATH_1)
15        pyplot.close(fig_1)
16        ...
17
```

The above process generating the first schedule diagram is repeated with the

same logic also for the other 2 profiles.
The operations carried forward in this Python unit are implemented using "numpy" and "matplotlib" libraries.

- **simulate the SoC**: in order to make the simulation as realistic as possible it is not enough to emulate the backend interface, but also the EV frontend. Then, a method used for mimicking the battery charging process is needed. It results are necessary also for technical purposes. RISE-V2G provides a message called "CurrentDemandReq" containing a mandatory parameter linked to the SoC value (EVRESSSOC). The method of implementing this functionality is structured as follows:

```
1    def start_charging():
2    while(EVCC_Properties.evresssoc < 100 and (EVCC_Properties.
     Risev2g_State=="WaitForChargingStatusRes" or EVCC_Properties.
     Risev2g_State == 'WaitForCurrentDemandRes')):
3            time.sleep(1)
4            EVCC_Properties.evresssoc += 3
5            if EVCC_Properties.evresssoc>100:
6                EVCC_Properties.evresssoc=100
7
```

### 4.3.3 Business Logic (BL) and Property file

The Business Logic (BL.py) Unit is so-called given its duties. It covers 3 fundamental required actions:

- **manage and coordinate the other satellite unit calls**,
- **handle the socket server communication**,
- **run the full stack code**.

The last objective is covered in the same script calling the "__main__" to the interpreter:

```
if __name__ == '__main__':
    GUI = EVSE_GUI.main_page()
    BL(GUI)
    GUI.mainloop()
```

Firstly, the GUI "main_page" is generated such that can be passed as an argument to the BL class and the GUI loop can be called.
The "customtkinter" library claims compulsorily the page is in the main loop.

The logic with which the BL class has been designed is the same either for the EVCC and SECC; so, its methods will be analyzed singularly in different sub-sections due to the complexity of the BL.py.
The BL() class accepts one argument to be passed called "window". It is the already created "main_page" of the GUI unit.
When initialized, it launches 4 mainThreads [13]. The Central Python unit consists of a single class initialized by launching 4 Threads.

```
class BL():
    def __init__(self, window):
        Thread_PlugConnection = threading.Thread(target=self.
    plug_connection)
        Thread_BL\_loop = threading.Thread(target=self.BL_loop, args
    =[window])
        Thread_socket = threading.Thread(target=self.sockett)
        Thread_rise = threading.Thread(target=self.start_rise)

        Thread_PlugConnection.daemon = True
        Thread_BL_loop.daemon = True
        Thread_socket.daemon = True
```

```
11          Thread_rise.daemon = True
12
13          Thread_PlugConnection.start()
14          Thread_BL_loop.start()
15          Thread_socket.start()
16          Thread_rise.start()
```

The 4 Threads roles are briefly introduced in the table below and described more in detail in the following sub-section.

| Thread | Semantics |
|---|---|
| PlugConnection | check continuously if the cable is plugged in or not |
| BL_loop | call the appropriate action depending on which is the communication current state |
| socket | handle the socket server communication |
| rise | run the JAR execution and restart it every time a charging sequence will end |

**Table 4.7:** Business Logic initialized Threads.

### Plug Connection Thread

It exploits the "psutil" library. The method simply consists of taking a list of booleans associated with different available connections, checking the one referring to the Ethernet cable and verifying it.

```
1 def plug_connection(self):
2     flag_first = True
3     while(True):
4         netStats=psutil.net_if_stats()
5         listName=list(netStats.keys())
6         ETH=listName[0]
7         if(netStats[ETH].isup == False):
```

```
8                SECC_Properties.Plug_connected = True
9            elif(netStats[ETH].isup == True):
10               SECC_Properties.Plug_connected = False
11           elif(flag_first == True and netStats[ETH].isup == False):
12               flag_first = False
13               self.open_page(('waiting_for_setup_page'))
14               EVCC_Properties.RiseContinue = True
```

It is relevant to highlight the global flag on the last line called "RiseContinue". It is used to pause the execution of the communication between the BL unit and RISE-V2G when it is set to False and make it restart once it changes to True. In particular, it affects the functioning of the socket server. Its specific operating logic of it will be explained in the next sub-section.

### Socket Server Thread [14]

It imports "socket" and "json" Python libraries. The structure of the socket server is the same as the one described for the client, but with the reading and sending process inverted in the flow. However, some differences exist.
The server shall be always available for the client to connect; in order to do so, an infinite while loop is initialized.

```
1    def sockett(self):
2        host = ''
3        port = 5561
4
5        socksize = 1024
6        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7        s.bind((host, port))
8        print("Server started on port: %s" % port)
9        s.listen(1)
10       while True:
11           flag_socket = True
```

```
12              print("Now listening...\n")
13              conn, addr = s.accept()
14
15              print('New connection from %s:%d' % (addr[0], addr[1]))
16              data = conn.recv(socksize)
17              data = data.decode('utf-8')
18              data = data.strip()
19              json_dict = json.loads(data)
20
21              EVCC_Properties.RiseContinue = False
22              ...
```

When the client connects to the server and it receives the message, the "loads()" method deserializes the incoming data to Python Object in order to be read by the socket. It distinguished the message by taking the CurrentState in the first field and checking it using a switch list.

```
1
2     match json_dict['CurrentState']:
3               ...
4               case 'WaitForCurrentDemandReq':
5                   SECC_Properties.Risev2g_State = json_dict['
    CurrentState']
6                   SECC_Properties.EVRESSSOC = json_dict['EVRESSSOC']
7                   SECC_Properties.EVSENotification = json_dict['
    EVSENotification']
8                   flag_socket = True
9               ...
```

In the second section, a second while loop is set, but at this time, the "flag_socket" provides a way out if it is changed to False. This operation can be done only once the parallel operation on the "BL_loop", the GUI unit and PFS are finished and the global flag "RiseContinue" is set again to True.

```
while flag_socket == True:
    if EVCC_Properties.RiseContinue == True:
        EVCC_Properties.RiseContinue = False
        flag_socket = False
        match EVCC_Properties.Risev2g_State:
            ...
            case "WaitForCurrentDemandReq":
                SECC_Properties.EVRESSSOC = json_dict['EVRESSSOC']
                SECC_Properties.EVSE_Notification = json_dict['
    EVSENotification']
                message = {'CurrentState':SECC_Properties.
    Risev2g_State', EVRESSSOC': SECC_Properties.EVRESSSOC, '
    EVSENotification': SECC_Properties.EVSENotification}
                reply = json.dumps(message)
            ...

        conn.sendall(bytes(reply, 'utf-8'))
```

The socket server ends up composing the reply message and sending it back to the Java client.

### Business Logic Loop Thread

If the socket Thread is the connection between the Business Logic and RISE-V2G, the BL_loop method is what connects the Central system to the GUI and PFS units.

It presents a simple code design. An infinite while loop is needed to keep the controllers awake. A second while loop using the "Plug_connected" flag is built in order to keep checking in which state the charging session is and, consequently, which operation shall be executed.

```
def BL_loop(self, window):
    self.open_page('first_page', window)
    while(True):
        while(SECC_Properties.Plug_connected == True):
            if(SECC_Properties.Risev2g_State == '
    WaitForServiceDiscoveryReq' and SECC_Properties.
    flag_ServiceDiscoveryReq == 0):
            self.set_value(SECC_Properties.Risev2g_State)
            self.open_page('payment_selection_page',window)
            ...
```

The method represented above includes some additional methods to hold some functionality more compactly.

- **set_value(self, Risev2g_State) method**: for each current state owns associated parameters that have to be set by default or taken from the GUI unit.

```
    def set_value(self, Risev2g_State):
        match Risev2g_State:
            case 'WaitForServiceDiscoveryReq':
                SECC_Properties.flag_ServiceDiscoveryReq = 1
            case 'WaitForPaymentServiceSelectioReq':
                SECC_Properties.flag_PaymentServiceSelectionReq =
    1
            ...

```

- **open_page("page_name", window) method**: some states require to open its corresponding GUI page. This method uses the window argument taken from the BL_loop Thread in order to open the single pages having the main one.

```
    def open_page(self, page_name, window):
        if (page_name == 'zero_page'):
            EVSE_GUI.zero_page(window)
        elif (page_name == 'first_page'):
            EVSE_GUI.first_page(window)
        ...

```

- **charging_option_selection(self, option_selected)**: the method used if Renegotiation, Pause or Stop is selected. It reset the appropriate flags depending on the charging option selected.
  The Renegotiation, Pause and Stop options involve a turning back in the charging flow re-proposing some states.

```
    def charging_option_selection(self, option_selected):
        match option_selected:
            case 'ReNegotiation':
                SECC_Properties.flag_ChargeParameterDiscoveryReq
= 0
                ...
            case 'Pause':
                ...
            case 'Stop':
                ...

```

- **reset_flags(self) method**: this method is used when the charging session ends. All the flags will be reset in order to be prepared for a new charging session.

```
    def reset_flags(self):
        SECC_Properties.flag_ServiceDiscoveryReq = 0
        SECC_Properties.flag_ServiceDetailReq = 0
        ...

```

- **contractCertChangeself method**: the method used when a CertificateUpdate message is required. It deletes the current Contract Certificate stored in the SECC and substitutes it with the new one such that it can be sent to the EVCC by RISE-V2G unit.

```
    def contractCertChange(self):
        oldPath = 'C:/Users/'+str(SECC_Properties.UserDir)+'/Demo-PnC/workspace/DEMO_SECC/rise-v2g-1.2.6/RISE-V2G-SECC/target/moCertChain.p12'
        newPath = 'C:/Users/'+str(SECC_Properties.UserDir)+'/Demo-PnC/workspace/DEMO_SECC/rise-v2g-1.2.6/RISE-V2G-SECC/target/moCertChain_New.p12'

        os.remove(oldPath)
        os.rename(newPath, oldPath)

```

### JAR execution Thread

The last Thread is involved in the execution of the RISE-V2G unit. It shall be executed using the JAR files respectively of the RISE-V2G-SECC and RISE-V2G-EVCC. It shall be launched as the last Thread in order to already have all the units ready for the EVCC to start the charging session. The JAR file is executed only when the cable is plugged in.

```
def start_rise(self):
    while(True):
        if(SECC_Properties.Plug_connected == True):
            SECC_Properties.Plug_connected = False
            Pathh = 'C:/Users/'+str(SECC_Properties.UserDir)+'/Demo-
    PnC/workspace/DEMO_SECC/rise-v2g-1.2.6/RISE-V2G-SECC/target'
            os.chdir(Pathh)
            os.system("java -jar rise-v2g-secc-1.2.6.jar")
```

### 4.3.4 Additional secondary files

Some procedure steps slow down the process of creating the new Contract Certificate, setting up the simulated backend and frontend certificate chains and generating the executable RISE-V2G EVCC and SECC JAR files.
Making the charging session flow faster and simpler to run is an important point to reach to make the demonstration as smooth as possible.
Thus, before running the EVSE_BL.py and EV_BL.py a compact file that will accomplish the previous functions is needed. The most suitable is a batch file, for a Windows simulation, and a shell file, for a Linux environment.
These will contain the same commands in different languages; so, only the first one will be analyzed.

```
    @echo off
    cd %UserProfile%\Demo-PnC\workspace\DEMO_SECC\rise-v2g-1.2.6\RISE
    -V2G-Certificates
    call generateCertificates.bat
    call copyNewCertsAndKeys.bat
    call generateNewContractCertificate.bat

```

```
7     cd %UserProfile%\Demo-PnC\workspace\DEMO_SECC\rise-v2g-1.2.6\RISE
      -V2G-PARENT
8     call mvn clean install
9
10    cd %UserProfile%\Demo-PnC\workspace\DEMO_EVCC\rise-v2g-1.2.6\RISE
      -V2G-PARENT
11    call mvn clean install
12
13    cd %UserProfile%\Demo-PnC
```

In the script, 3 sections are present. In the first one, the batch files involving the certificate generation are called. They generate all the necessary certificate chains gathering them in the Keystores and Truststores. Then, they are moved to the appropriate SECC and EVCC repositories. Finally, the new Contract Certificate is created and stored in the SECC directory.
The second and third sections act to generate RISE-V2G-SECC and RISE-V2G-EVCC JAR files. The last code line returns to the working directory.
The "%UserProfile%" keyword is used for generalizing the script and making it applicable to every working machine.

An additional batch file is built to end all the Java and Python background processes to be sure there will not be conflicts with a new run.

```
1     taskkill /IM java.exe /F
2     taskkill /IM python.exe /F
```

It kills forcefully the Java and Python open processes.

# Chapter 5

# Testing results

A set of different use cases are simulated in order to verify the whole program works as planned and its design results are suitable for a PoC implementation.
The following test cases will not take into account the EIM payment option through a QR code. It will be proposed in the actual simulation and the code is set up to cover that use case, but for interest grounds, it will not be analyzed.
As regards the certificates handling, the scenario involving the revocation of the contract certificate using the Certificate Revocation List (CRL) has been considered as Out-of-Scope.
Further consideration shall be made about the difference of how the PoC simulation is built and how actually should work with a real customer. In the following test cases, the user has the maximum choice capability, especially for parameters that will be set by default in a real implementation. In a real scenario, the only choice given to the driver will be the possibility to stop the charging session. A custom user experience can be thought of such that the user can select the charging profile and the renegotiation as a charging option to change his mind.

## 5.1 Test case: AC mode standard charging session simulation using Plug&Charge payment option

In the standard Plug&Charge scenario, the SECC will stay in listening mode for the EVCC to plug in the Ethernet cable. In the meanwhile, the EVSE backend simulator is loaded together with its certificate chains. When the EVCC connects to the SECC, the following procedure will be proposed:

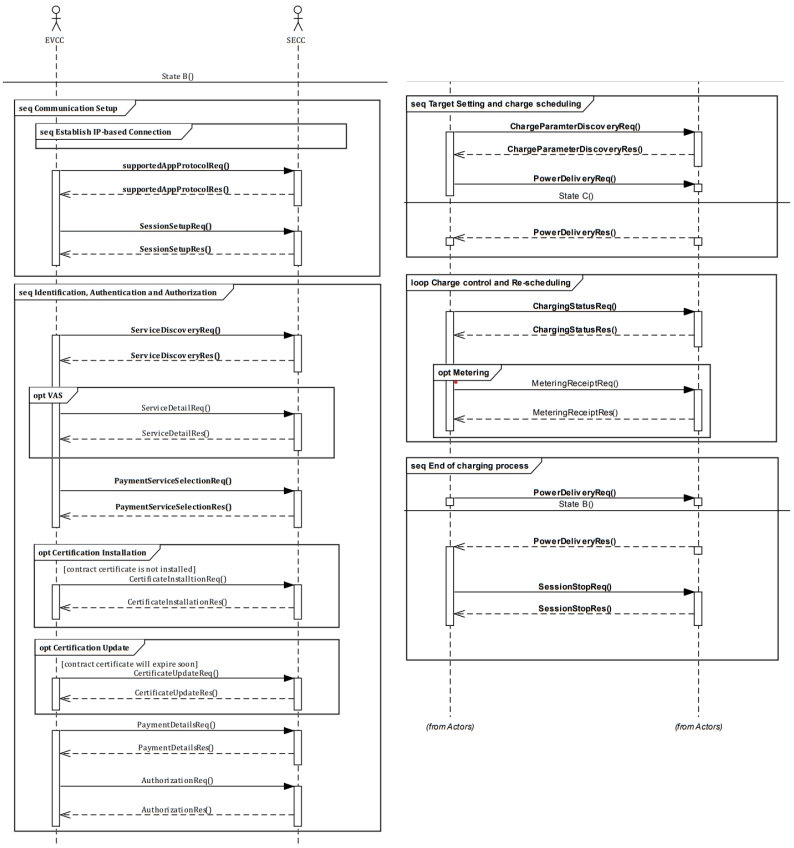


**Figure 5.1:** AC Request-Response Message Sequence PnC payment option.

The first 2 sets of messages contained in the Communication Setup section run in the background without the intervention of the driver.

The next message sequence provides the Identification, Authentication and Authorization of the EV by the SECC. It is the core of the PnC feature technological revolution.
In this phase, the driver is required to select the Payment Option and if he wants to approve further actions on the contract certificates if needed.

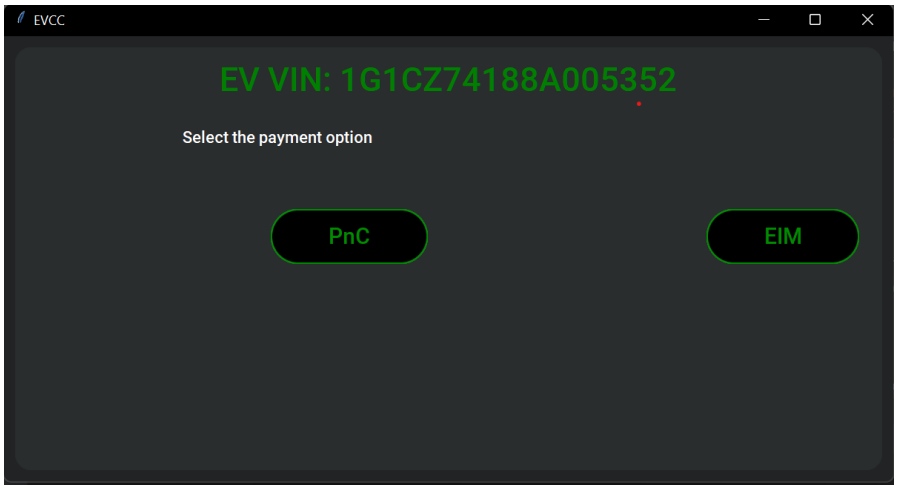


**Figure 5.2:** EV GUI page of the selection of the payment option.

Once the PnC has been chosen during the *PaymentServiceSelection* Request-Response sequence, the real identification procedure can take place. In this test case, the contract certificate is already installed and it is still valid such that the *Authorization* process starts. In terms of the check method used by the communication interface, the homonymous messages simply exchange a random alphanumeric identifier called *genChallenge*, installed in the EV during the PaymentDetails. The contract certificate is checked by the EV itself and the call to the SECC starts directly by the EVCC when it is triggered by the expiration date or the absence of it.
In addition, together with the *genChallenge* the message SignatureValue is checked.

The next step involves the choice of the departure time, which can be set to 24 hours or with a 15-minute gap, and the charging profile. The proposed charging

schedules are 3 and represent the *SalesTariffs* passed by the SECC in the *ChargeParameterDiscoveryRes*.

The "Charge Scheduling" section is completed by the exchange of the *PowerDelivery* message couple set. It works like a "ready flag" for the EVCC to confirm the chosen charging profile and its readiness to start the charging session.

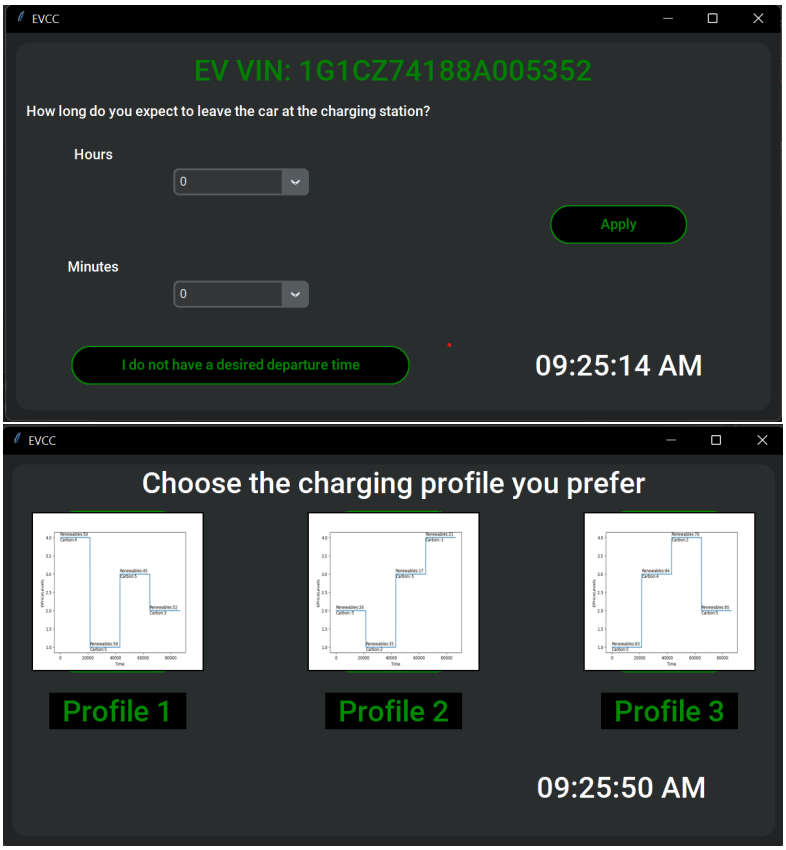


**Figure 5.3:** EV GUI pages of departure time and charging profiles selection.

During the charging session, a message exchange loop is required. In the AC mode, the couple *ChargingStatus* and *MeteringReceipt* messages are provided in the loop. The first one is mandatory while the other one is optional. The compulsory message is used mostly by the SECC to keep alive and under control its status, while the metering info of the current charging session is considered with the optional one.

During the charging loop, the EV infotainment and the EVSE screen show 2 types of information: passive data, like State of Charge, VIN, charging profile selected, timestamp and labels, and active data. The second ones are interactive and are displayed as buttons. In the EVSE display, the Renegotiation and the Stop are proposed, while in the vehicle infotainment, the Pause option is added.

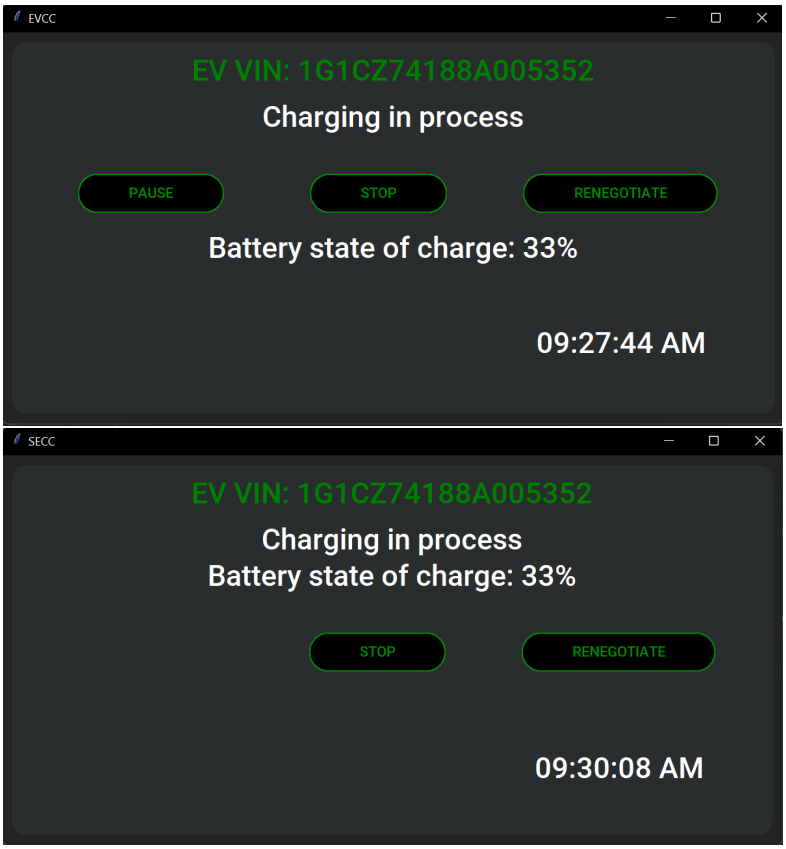


**Figure 5.4:** EV (left) and EVSE (right) GUI charging loop page.

If none of the options are selected, the charging session ends with a *SessionStop* message exchange with which all the communication protocols are closed.
Otherwise, the following procedure will be followed:

- **Stop**: the end of the charging session is forced and the *SessionStop* messages are exchanged. It can be triggered either by EVCC or SECC.
- **Renegotiation**: it consists of re-proposing updated charging profiles to the

driver. Thus, it involves the re-start from the *ChargeParameterDiscovery* Request-Response sequence. It may be forced either by EVCC or SECC.

- **Pause**: it consists of an interruption in the charging session. When it restarts, the communication resumes from *ChargeParameterDiscovery* Request-Response sequence.

## 5.2 Test case: DC mode standard charging session simulation using Plug&Charge payment option

The Request-Response message sequence in a DC mode charging scenario is described in the following flow chart:

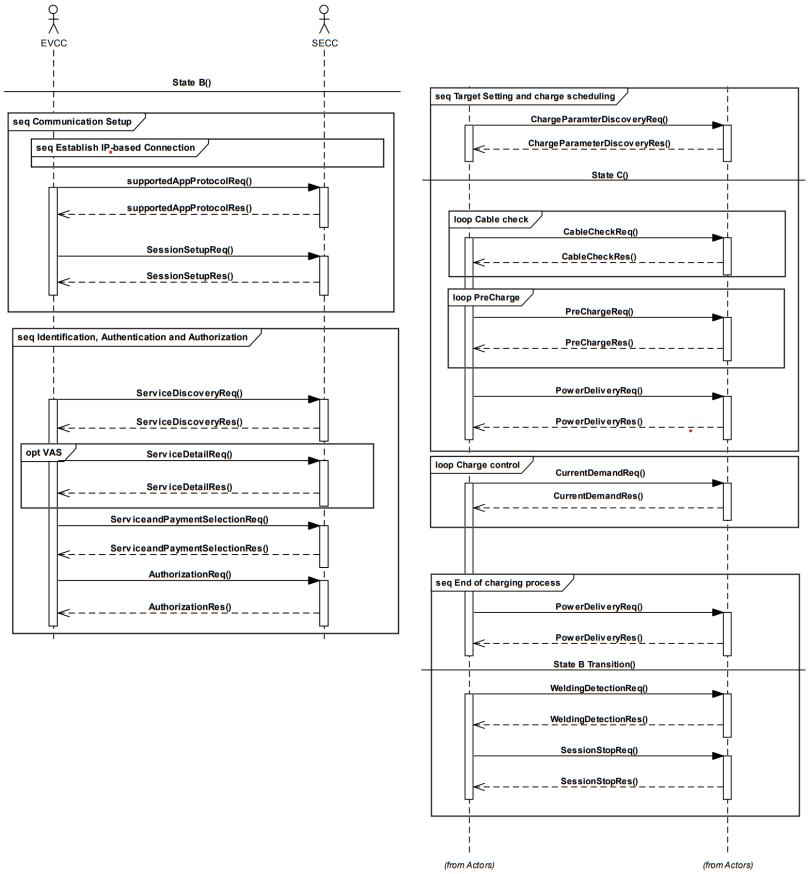


**Figure 5.5:** DC Request-Response Message Sequence PnC payment option.

The first 2 blocks involve the same message sequence as in AC mode. Therefore, the steps are not more analyzed, but they are assumed to be the same.
Actually, in GUI there are also no differences between the 2 charging modes. The choices offered to the driver are the same, what changes is the message flow. In fact, from *ChargeParameterDiscovery* message forward, the ISO15118-2 protocol requires different messages.
*CableCheck* and *PreCharge* are the message sets responsible for communicating the target voltages and especially the EV and EVSE status.

The Charging loop consists only of a single message exchange called *CurrentDemand*. Acting on these message fields, the SECC may trigger the Renegotiation or the Stop. The rest of the communication is the same as well as the selection of Renegotiation, Pause or Stop options.

The final GUI page summarizes the most important parameters for the driver to know (Fig. 5.7).

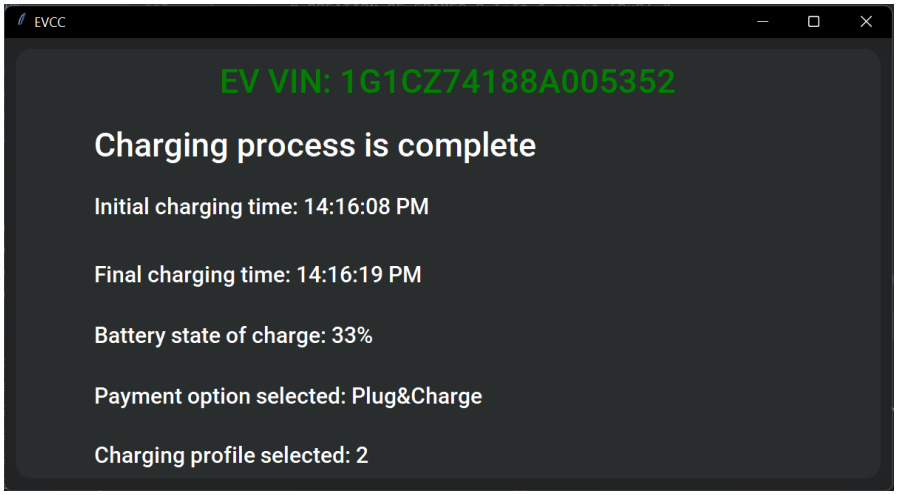


**Figure 5.6:** GUI page of the end of the charging session.

A further optional scenario shall be described. There is the possibility that the cable will be unplugged before the end of the charging session. In this case, the *CableCheck* verifies that the cable is actually locked and plugged. Otherwise, a GUI page (Fig. 5.8) is displayed either on the EV infotainment and EVSE screen.

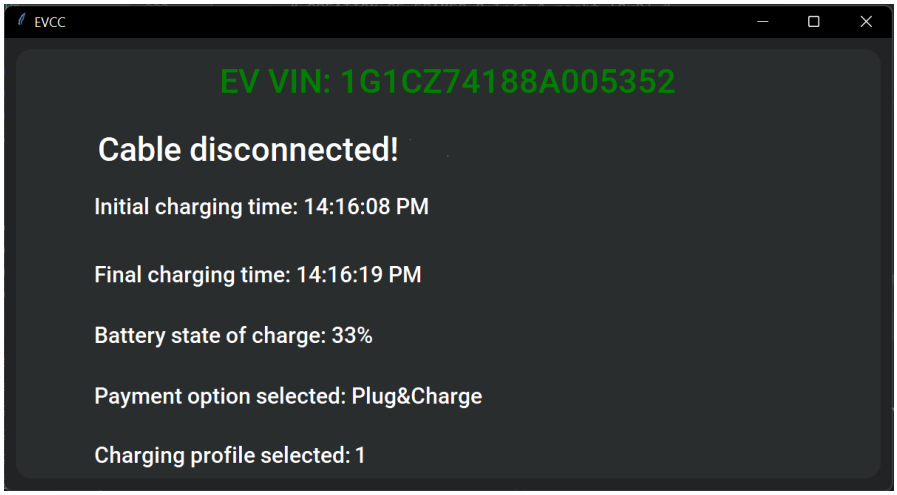


**Figure 5.7:** Disconnection cable GUI page.

## 5.3 Test case: Certificate handling

The last test case concerns the contract certificate handling. 2 actions are provided for this type of certificate: its installation in the EV and its update.
They are treated in the same way either in the AC scenario and in DC charging mode, so the message sequences are the same in both cases.

### 5.3.1 T.1: Certificate Installation

The installation of the contract certificate is triggered by the EVCC in the first place. Choosing the Plug&Charge feature triggers a vehicle internal check on the presence and the validity of the Contract Certificate inside the Truststore.jks.
The first time the EVCC connects itself to the EVSE and chooses the PnC as the payment option the Contract Certificate installation page (Fig. 5.9) is displayed on the screen.

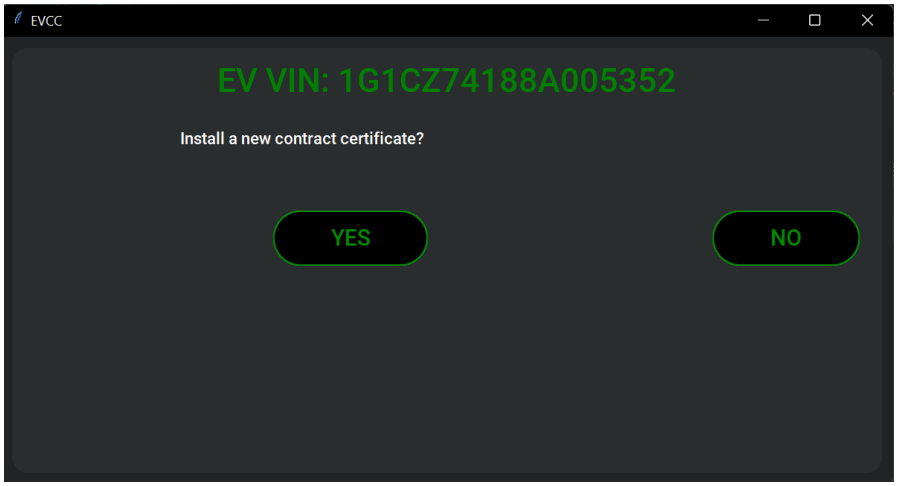


**Figure 5.8:** Certificate Installation GUI page.

Choosing to install the new Contract Certificate, the *CertificateInstallation* Request-Response sequence will take place. At first the EVCC load the *CertificateInstallationReq* with its XML representation filling its fields with the OEM Provisioning certificate chain and the List of Root Certificate IDs. This latter contains the Issuer name and the serial number of the V2G Root.
In addition, the Signature for this message is generated using the procedure described in Chapter 2.
Once the SECC receives the Request message, it verifies the signature and, if it results valid, the *CertificateInstallationRes* is prepared.
The Response message stores the new Contract Certificate, its associated encrypted private key and contract signature, the eMAID, DHpublickey and the certificate chain of the Secondary Actor, so the eMSP in this case. Again, in this case, the Signature is generated and attached to the message.
It is sent back to the EVCC which verifies the Signature and stores its new Contract Certificate.

### 5.3.2 T.2: Certificate Update

The update of the Contract Certificate is triggered by the EVCC in first place. Choosing the Plug&Charge feature triggers a vehicle internal check on the presence and the validity of the Contract Certificate inside the Truststore.jks.
If the EV verifies positively the presence of the Contract Certificate, a further check on its validity period is done. If it results near the expiration date, the certificate update is required; so the appropriate GUI page (Fig. 5.10) is displayed:

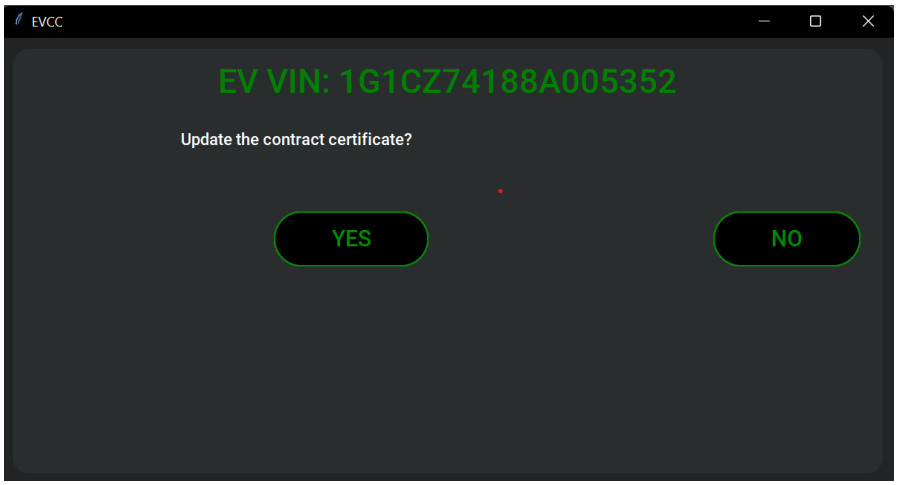


**Figure 5.9:** Certificate Update GUI page.

If the driver answers positively to the GUI page, the *CertificateUpdateReq* is built. It is sent to the SECC once a message signature is generated and attached to it. When the EVSE receives it, verifies the signature and, if it results valid, the *CertificateUpdateRes* is generated. As it has been done for the Installation, the same correspondent fields are filled in the XML Response message together with the signature. The EVCC receives it and installs the new Contract Certificate erasing the old one.

# Chapter 6

# Conclusions

This last chapter serves as a summary of the entire project highlighting the strengths and weaknesses. The expected results achieved, the tasks that are still to be developed and the improvable parts.

## 6.1 Achievements

The design of this Proof-of-Concept covers most of the assumed and desired objectives, with one exception for the Value Added Service, but this point will be analyzed later.
The charging simulation meets all the requirements specified by ISO15118-2. Although RISE -V2G leaves some undeveloped, they are not among the tools required to make the code functional and compliant. The missing "TODO" items address functional or linkage issues between the code parts, but do not impact the compliance with the ISO15118 protocol
The biggest shortcoming is the backend mock-up. For the current simulation, its integration resulted inevitably, as an implementation that could be considered close to reality would have required a lot of working time and the involvement of all the stakeholders presented in the VDE-AR -E application guide. In addition, the

implementation of backend message communication would also have required the study and integration of multiple protocols such as OCPP, OCPI, and OSCP.
For these reasons, a simplified local backend version was loaded on the EVSE site.
The goals set a priori by the project have been achieved almost for their entirety.
In fact, the implementation claims the following features:

- **strong data security**:
  all the requirements of security levels are guaranteed, from the message encoding to the certificate handling.
- **high customizable charging session**:
  the driver has the maximum capability to adapt the charging session parameters to his needs.
- **strong code flexibility**:
  the code is well-organized, easy-to-use and it owns good adaptability.

The only target that cannot be developed is the integration of the VAS feature as an additional service for the Plug&Charge charging option. Here the limit is not technological, but is represented by the protocol version. In fact, ISO15118-2 has introduced the concept of Value Added Service and also provides the possibility to exchange it as a field in the messages dedicated to the information about the services. However, version -2 does not provide a state or message type to implement an actual addition.
A possible solution could be to extend the code with a further Python unit capable of providing additional services, but the risk is overcome by the communication protocol. Therefore, compliance with the standard should be prioritized.

The other improvable part is the test environment. The testing results are extrapolated by simply running the firmware on the Raspberry Pi boards and looking that all test cases worked properly. An appropriate and customized test platform will be able to give better feedback for sure.

## 6.2 Future developments

The ISO15118 standard already knows the new -20 upgrade version [11]. Like the -2, it represents a great advancement in the eMobility world from more points of view:

- **V2G technology**[15]:
  states and messages are adapted to welcome the V2G requirements. The advantages of using this technology are numerous. They apply to a small environment like a private house recycling and minimizing the energy waste till bigger scenarios, like corporate and public buildings. It will help also the interconnected energy market to stabilize the energy Demand-Response.

- **Certificate handling**:
  multiple contract certificates can be stored in a single EV. These certificates will be able to be sent in the same message. It implies also greater interoperability between PKI ecosystems.

- **Value Added Service**:
  dedicated states and messages are required to integrate VAS. The eMSPs will be able to offer to the driver a series of additional secondary services accessible during the charging session.

- **Energy Flexibility**:
  a new dynamic charging mode will be available for the driver. It involves the dynamic adaptation of the charging power in the proposed charging schedule to satisfy the energy grid needs.

# Bibliography


[1] IEA. «Electric Vehicles». In: (2022) (cit. on p. 3).

[2] Sesha Gopal Selvakumar. «Electric and Hybrid Vehicles – A Comprehensive Overview». In: *2021 IEEE 2nd International Conference On Electrical Power and Energy Systems (ICEPES)*. 2021, pp. 1–6. DOI: 10.1109/ICEPES52894.2021.9699557 (cit. on p. 5).

[3] Navpreet Hans. «Trends In Electric Vehicle (EV) Charging and Key Technology Developments». In: *International Journal of Engineering Research and* V9 (Sept. 2020). DOI: 10.17577/IJERTV9IS090042 (cit. on p. 13).

[4] Lonneke Driessen and Paul Klapwijk. «Public Key Infrastructure for ISO15118: Freedom Of Choice For Consumers & An Open Access Market». In: (June 2022), p. 196. DOI: 10.13140/RG.2.2.11878.09282 (cit. on p. 16).

[5] *X.509 : Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*. ITU, 2019. URL: https://www.itu.int/rec/T-REC-X.509-201910-I/en (cit. on p. 19).

[6] *VDE-AR-E 2802-100-1 Anwendungsregel:2019-12 Handling of certificates for electric vehicles, charging infrastructure and backend systems within the framework of ISO 15118*. DKE, 2019. URL: https://www.vde-verlag.de/standards/0800642/vde-ar-e-2802-100-1-anwendungsregel-2019-12.html (cit. on p. 21).

[7] Vincent Lozupone. «Analyze encryption and public key infrastructure (PKI)». In: *International Journal of Information Management* 38.1 (2018), pp. 42–44. ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2017.08.004. URL: https://www.sciencedirect.com/science/article/pii/S0268401217303195 (cit. on p. 21).

[8] Minho Shin, Hwimin Kim, Hyoseop Kim, and Hyuksoo Jang. «Building an Interoperability Test System for Electric Vehicle Chargers Based on ISO/IEC 15118 and IEC 61850 Standards». In: *Applied Sciences* 6.6 (2016). ISSN: 2076-3417. DOI: 10.3390/app6060165. URL: https://www.mdpi.com/2076-3417/6/6/165 (cit. on p. 25).

[9] *BS EN ISO 15118-1:2019*. BSI Standards Limited 2019, May 2019. URL: https://www.iso.org/standard/69113.html (cit. on pp. 36, 37).

[10] *BS EN ISO 15118-2:2019*. BSI Standards Limited 2019, May 2019. URL: https://www.iso.org/standard/84207.html (cit. on pp. 36, 37, 40).

[11] *BS EN ISO 15118-20:2022*. BSI Standards Limited 2022, 2022. URL: https://www.iso.org/standard/77845.html (cit. on pp. 38, 117).

[12] GeeksforGeeks. *Socket Programming in Java*. Accessed 2022, https://www.geeksforgeeks.org/socket-programming-in-java/?ref=gcse. n.d. (Cit. on p. 76).

[13] GeeksforGeeks. *Thread-based parallelism in Python*. Accessed 2022, https://www.geeksforgeeks.org/thread-based-parallelism-python/?ref=gcse. n.d. (Cit. on p. 94).

[14] GeeksforGeeks. *Socket Programming in Python*. Accessed 2022, https://www.geeksforgeeks.org/socket-programming-python/?ref=gcse. n.d. (Cit. on p. 96).

[15] Byungchul Kim. «Smart charging architecture for between a plug-in electrical vehicle (PEV) and a smart home». In: *2013 International Conference on*

*Connected Vehicles and Expo (ICCVE)*. 2013, pp. 306–307. DOI: 10.1109/ICCVE.2013.6799811 (cit. on p. 117).