FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Vehicle On-Board Charging Security Scanner
Student:	Bc. Pavel Khunt
Supervisor:	MSc. Thomas Sermpinis
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

Functionality of a car relies on dozens interconnected computer systems, so-called ECU. Those ECUs always formed closed internal network with limited exposure to the outside world. Nevertheless, to support advanced features of modern cars, its manufacturers are forced to incorporate new complex externally facing interfaces dramatically increasing cyber security attack surface. One such interface, is the On-Board Charging (OBC) port used to connect electrical vehicles to power grid and support smart charging. Introducing new complex externally facing interface to the vehicle introduces potential cyber security vulnerabilities which must be addressed by security and development teams of the automotive sector. Goal of this thesis would be to design and develop a security evaluation software, which will help enumerate and partially evaluate the publicly accessible OBC port on the European electrical vehicle.

1. Survey existing research on OBC, OBC security and modern vehicle architecture.
2. Analyze the functionality of the interface and consider the scope and network layer protocol(s) on which the security evaluation tool should focus. Agree the exact scope with the thesis supervisor.
3. Implement the security evaluation tool (scanner).
4. Prepare detailed documentation for the usage of the tool, as well as the target scope.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 7 March 2023 in Prague.

Master's thesis

VEHICLE ON-BOARD CHARGING SECURITY SCANNER

Bc. Pavel Khunt

Faculty of Information Technology
Department of Information Security
Supervisor: MSc. Thomas Sermpinis
January 11, 2024

Contents

List of Figures

List of Tables

List of code listings

I would like to especially thank the thesis supervisor, Msc. Thomas Sermpinis, for the guidance of my thesis, the consultations and comments provided, thanks to which I was able to continuously improve the thesis. I am also grateful to my father and my entire family for their provision of facilities, unwavering support, and sympathetic understanding, without which this thesis could not have been completed. Finally, my deepest gratitude goes to my girlfriend, MUDr. Renata Sobíšková, for her support and patience throughout my master's studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on January 11, 2024

Abstract

This thesis focusses on the cybersecurity of electric vehicles, with an emphasis on security issues associated with the On-Board Charging (OBC) port. I have developed a security tool that enables the enumeration and partial evaluation of the publicly accessible OBC port on European electric vehicles. The work involved exploring existing research in the field of On-Board Charging (OBC), OBC cyber security, and modern vehicle architectures. This was followed by an analysis of the functionality of the OBC interface with a focus on High-Level Communication (HLC) according to ISO 15118-2. The result was the successful implementation of a security assessment tool, with its usage and target scope meticulously documented. The whole process was aimed at enumerating and partially evaluating the publicly accessible OBC port in European electric vehicles to improve cybersecurity in the automotive industry. The outcome of the work is a tool that effectively supports the enumeration and testing of the implementation of the Electric Vehicle Communication Controller (EVCC) through the OBC port. This tool is designed to provide an overview and perform tests on the implementation of EVCC, which is accessible through the OBC port and functions as a controller that manages communication with the Supply Equipment Communication Controller (SECC).

Keywords automotive, electric vehicle, OBC, V2G, ISO15118, security tool

Abstrakt

Tato práce se zaměřuje na kybernetickou bezpečnost elektrických vozidel, s důrazem na bezpečnostní otázky spojené s On-Board Charging (OBC) portem. Vyvinul jsem bezpečnostní nástroj, který umožňuje enumeraci a částečou evaluaci veřejně přístupného OBC portu na evropských elektromobilech. Tato práce zahrnovala průzkum existujícího výzkumu v oblasti On-Board Charging (OBC), kybernetické bezpečnosti OBC a moderní architektury automobilů. Následovala analýza funkcionality OBC rozhraní s důrazem na High-Level Communication (HLC) podle ISO 15118-2. Výsledkem byla úspěšná implementace bezpečnostního hodnotícího nástroje, jehož použití a cílový rozsah byly pečlivě zdokumentovány. Celý proces byl zaměřen na identifikaci a částečné hodnocení veřejně přístupného OBC portu na evropských elektromobilech s cílem posílit kybernetickou bezpečnost v automobilovém průmyslu. Výsledkem práce je nástroj, který efektivně podporuje enumeraci a testování implementace Electric Vehicle Communication Controller (EVCC) prostřednictvím OBC portu. Tento nástroj slouží k získání informací a provedení testů ohledně implementace EVCC. EVCC je dostupný skrze OBC port, a funguje jako kontroler řídící komunikaci s Supply Equipment Communication Controller (SECC).

Klíčová slova automobilový průmysl, elektrické vozidlo, OBC, V2G, ISO15118, bezpečnostní nástroj

List of abbreviations

AC	Alternating Current
ADAS	Advanced Driver Assistance Systems
BMS	Battery Management System
CAN	Controller Area Network
CCU	Charging Control Unit
DC	Direct Current
ECU	Electronic Control Unit
EIM	External Identification Means
EV	Electric Vehicle
EVCC	Electric Vehicle Communication Controller
EVSE	Electrical Vehicle Supply Equipment
EXI	Efficient XML Interchange
HLC	High Level Communication
HMI	Human Machine Interface
HPGP	HomePlug Green PHY
IMD	Insulation Monitoring Device
IP	Internet Protocol
IPv6	Internet Protocol version 6
IVN	In-vehicle network
LIN	Local Interconnect Network
MOST	Media Oriented System Transport
OBC	On-Board Charging or On-board Charger
PDU	Protocol Data Unit
PEV	Plug-in Electric Vehicle
PHEV	Plug-in Hybrid Electric Vehicle
PKI	Public Key Infrastructure
PLC	Power Line communication
PnC	Plug and Charge
SDP	SECC Discovery Protocol
SECC	Supply Equipment Communication Controller
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
V2G	Vehicle to Grid
V2GTP	V2G Transfer Protocol
VCCU	Vehicle Charging Control Unit
VCU	Vehicle Control Unit
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Introduction

This chapter contains a basic introduction to electric vehicles and motivation for the topic of this thesis. It also includes the project goals.

The functionality of modern cars is based on interconnected computer systems, known as electronic control units (ECUs). Together, these units form a closed internal network with limited access to the external world. With the development of hybrid and electric vehicles, manufacturers have been forced to add additional control units associated with the operation of newly added electric components of the car, such as electric motors and batteries.

Furthermore, with the aforementioned modification of the vehicle architecture, external charging interfaces needed to be added. The charging socket is directly connected to the charging controller (also called charging ECU) located in the internal network of the vehicle. The charging ECU is responsible for communication between the electric vehicle and the charging station. The ECU, which is responsible for controlling the charge, is so-called a charging control unit (CCU). This ECU can be a standalone device or a built-in device called an on-board charger (OBC); this depends on the manufacturer and type of charging (AC or DC).

The main task of the on-board charger is to convert alternating current (AC) to direct current (DC), so it was originally designed for AC charging. The OBC has also evolved with the development of modern cars, and its other task is to support smart charging, which is done by the ECU placed in this OBC device. Smart charging means charging using the following standards: ISO15118, DIN SPEC 70121, and SAE J2847/2 standards [1]. This enables high-level communication that follows those standards. Based on the fact that the ISO15118 standard is preferred in Europe, this research is orientated only to this standard. The SAE standard is favoured in North America, so they are not included.

The high-level communication significantly expands the attack surface of a given vehicle through this publicly available port (charging socket), since communication with the OBC (or Charging Control Unit) is done through this port. Therefore, an attacker can potentially send malicious packets that cause a failure in the charging control unit. It may have a significant impact on vehicle security.

For the above reasons, it is important to have a tool that allows simulation of the attacker's behaviour. That is, it facilitates the enumeration and partial evaluation of the control unit processing the communication. The motivation for this thesis is to provide a security tool for experts and development teams working on charging control units. The tool can be useful to overcome security or logic flaws in the implementation and design of smart charging controllers. In addition, it can also serve security teams engaged in penetration testing in the automotive industry.

The main goal of this thesis is to implement a tool that can enumerate, send malicious messages, and partially evaluate the charging control unit (the ECU in an OBC module or a

standalone CCU).

The implementation of the tool is preceded by research on OBC, OBC security, and modern vehicle architecture. Detailed documentation, as another sub-objective, is included in this text. The text provides a detailed description of the launch of the tool, the process of building the tool, and the process of working with the tool, as well as examples of the various parts of the tool.

Chapter 1

Theoretical background

This chapter introduces the architecture of modern vehicles. The main focus in terms of architecture is on the network within the vehicle and the associated publicly accessible interfaces. The attack surface of a modern vehicle is examined in the following section. In that section, I include a comparison and discussion regarding the expansion of the attack surface for electric vehicles in general versus gasoline or diesel cars.
The next sections of this chapter build on the attack vectors mentioned for electric vehicles, specifically the attack vector associated with charging. Therefore, in turn, the basics related to EV charging, the associated charging interface, a more detailed description of the OBC and the security of the OBC are explained.
The final section deals with the international standards for communication during charging. This part also provides a detailed description of the requirements and how to establish communication with respect to the ISO15118 standard between the SECC (Supply Equipment Communication Controller) and the EVCC (Electric Vehicle Communication Controller).

Since this thesis is focused on one of the components of the internal network that is responsible for communication during vehicle charging, it is necessary to start by introducing the internal network architecture of modern vehicles.

In this introduction, the network architecture of a conventional vehicle (gasoline or diesel) versus an electric vehicle is provided. In addition, I examine the external interfaces that provide access to the aforementioned internal network. This background is essential to cover the distinctions of the internal network and the interfaces between conventional and electric vehicles.

A further section called the attack surface is based on the introduction to architecture of modern vehicles. In this section, the possible attack vectors for modern vehicles are described in general terms. It also mentions the variation in the attack surface due to the addition of charging-related components to the vehicle.

For this reason, I also focus on charging electric vehicles, in general. In particular, I describe smart charging, charging methods, charging setup, charging interfaces, DC vs. AC charging, standards, and protocols used.

As the main goal of this thesis is to design and develop a tool focussing on the publicly available OBC port, the following section is dedicated to OBC, its security and the standards used for communication between SECC and EVCC. These controllers can be integrated into the OBC device or as standalone devices, depending on the type of charging and the manufacturer of the vehicle in question. ISO 15118 is a standard that plays an important role in communication during charging.

This ISO is a series of standards that describe and define high-level communication (HLC) between the vehicle and the EVSE [2]. This standard is adopted internationally, especially in

Europe and the United States. It is also implemented in electric vehicles by major manufacturers such as Tesla, BMW, Mercedes, and Volkswagen. [3]

Therefore, this theoretical chapter ends with a description of the OBC, its security, and the ISO 15118 standard. I conclude with a more detailed discussion of the requirements and a description of the communication between EVCC and SECC according to ISO 15118-2.

1.1 In-Vehicle networks

The in-vehicle network, also called the automotive bus, plays a major role in data communication inside the vehicle. Generally, this network can be defined as a specialised internal communication network designed to interlink the various components within the vehicle. These include electronic control units (ECUs), various sensors, gateways, and other electronic systems in the vehicle [4, 5].

In the past, the approach used for the addition of new electronic components to the vehicle was to add a new electronic control unit (ECU) and associated circuitry for each new electronic sensor or application. Following this approach, the in-vehicle network has grown considerably. This resulted in a complex and often heterogeneous system in a vehicle. In order to solve this problem, the relevant ECUs were interconnected to exchange data and to enable the implementation of more advanced capabilities. The efficiency of this approach rapidly decreases with the number of ECUs because the use of point-to-point links exponentially increases the number of interconnections required depending on the number of ECUs. [5]

With the development of newer, more sophisticated sensors and further innovations, the internal network of modern vehicles has also had to expand. According to [6], the outcome of these innovations is that the internal network of a modern vehicle can contain up to 100 ECUs.

In today's modern vehicles, ECUs are used in various systems, including ADAS, infotainment, body control, and comfort systems, among others. This implies that ECUs process a large amount of information or data. Due to the growing complexity of vehicle systems and data transmission, it is no longer feasible to use a point-to-point connection.

As a consequence of the higher complexity of the systems, the point-to-point approach leads to an increase in the number of cables required to interconnect the various components. It is considered that a significant quantity of cables is not suitable due to space limitations and the increase in weight of the vehicle. According to [7] the cable bundles weigh more than 91kg and their length is several kilometres. Furthermore, the higher complexity makes it significantly more difficult to detect an error in the event of failure of one of the units. [8]

Consequently, due to the aforementioned issues, distributed systems have been developed over time to efficiently link components. In these systems, electrical components are connected through buses [8]. The buses are connected to each other through gateways designed to provide fast protocol conversion between the two networks [9]. Bus-connected ECUs and gateway-connected buses together form a completely interconnected network. The interconnection structure of the components differs depending on the architecture used (e.g., central gateway architecture, domain-based architecture, zone architecture) [7].

With the advent of the use of in-vehicle network (IVN) buses, various protocols have been developed for this purpose in the automotive industry. The choice of protocol depends on the purpose and functionality of the connected units in a bus network, so different parts of the vehicle's internal network use different communication protocols. The following protocols are widely used in the automotive industry: Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented System Transport (MOST), FlexRay, and Automotive Ethernet.

Finally, a diagram of a typical modern vehicle internal network is presented in Figure 1.1. The diagram outlines the interconnection of ECUs within a bus network and the interconnection of different buses via gateways.

The second diagram (Figure 1.2) is more detailed in the sense that it demonstrates the interconnection between individual sensors, ECUs, buses, and gateways. The Figure 1.2 illustrates

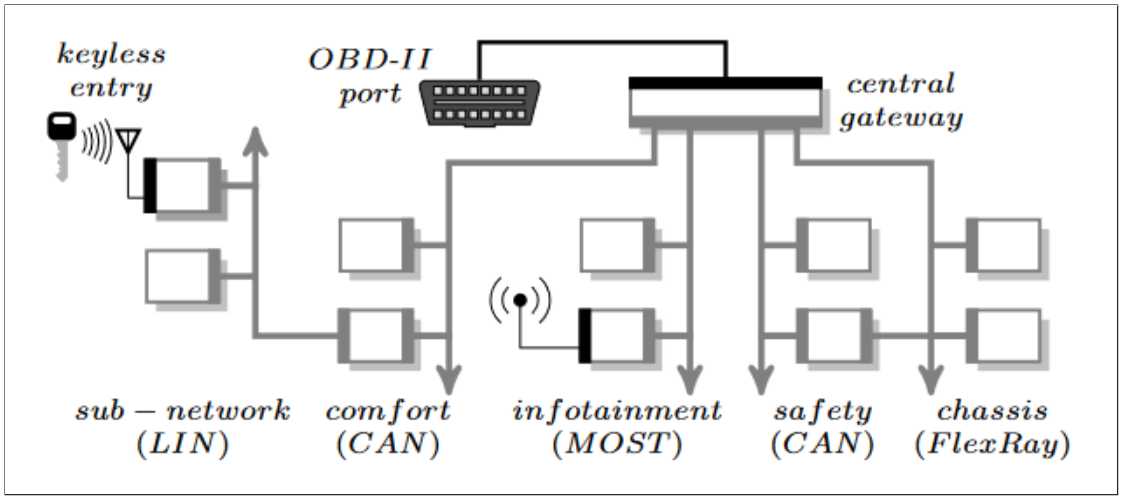


Figure 1.1 Simplified in-vehicle network architecture [10]

a more specific use of automotive protocols to interconnect individual network components in a vehicle.

Figures (Figure 1.1, Figure 1.2) shown for the vehicle network architecture are generic and simplified compared to the real schematics. The technologies and architecture design used depend on the type of the vehicle and its manufacturer.

1.1.1 CAN

CAN an abbreviation for the Controller Area Network is an electronic communication bus. The controller area network was first developed by Bosch in 1986 to replace a complex wiring harness with a two-wire bus in the automotive industry. It was later adopted as an international standard in 1993 and is called ISO 11898. [15]

In a CAN network, each ECU or "node" can communicate with any ECU. In this network, messages are broadcast to the entire network, ensuring data consistency at every node of the system. The consequence of using broadcast is that every node has access to all transmitted data. CAN hardware provides local filtering so that each node can decide whether to accept or ignore the message. [15]

CAN nodes can initiate transmissions when an idle bus is detected. Therefore, it may occur that more than one node tries to transmit data through the bus. In order to overcome this problem, bus access is allocated using a bus arbitration process. This process ensures that the message with the highest priority receives access to the bus. [8] If a particular node detects that another node is sending a message with a higher priority than it, it terminates sending and listens only to incoming messages on the bus [16]. Therefore, it serves as a communication bus, operating on the principle of event-driven communication.

CAN is considered a medium-speed bus with speeds up to 1Mbit/s. The CAN network carries many short messages, such as temperature or RPM, and the main use of the CAN bus is in sensor-based systems (applications). [8]

Another use of the CAN standard is that other higher-layer protocols are built on top of it. These include SAE J1939, OBD2 (on-board diagnostics), and CANopen. The main benefits of CAN are the simplifying of the in-vehicle network complexity and the reduction of costs due to reduced number of cables necessary for the connection of ECUs. It also allows for centralised diagnostics and data logging for all ECUs in the network. [15]

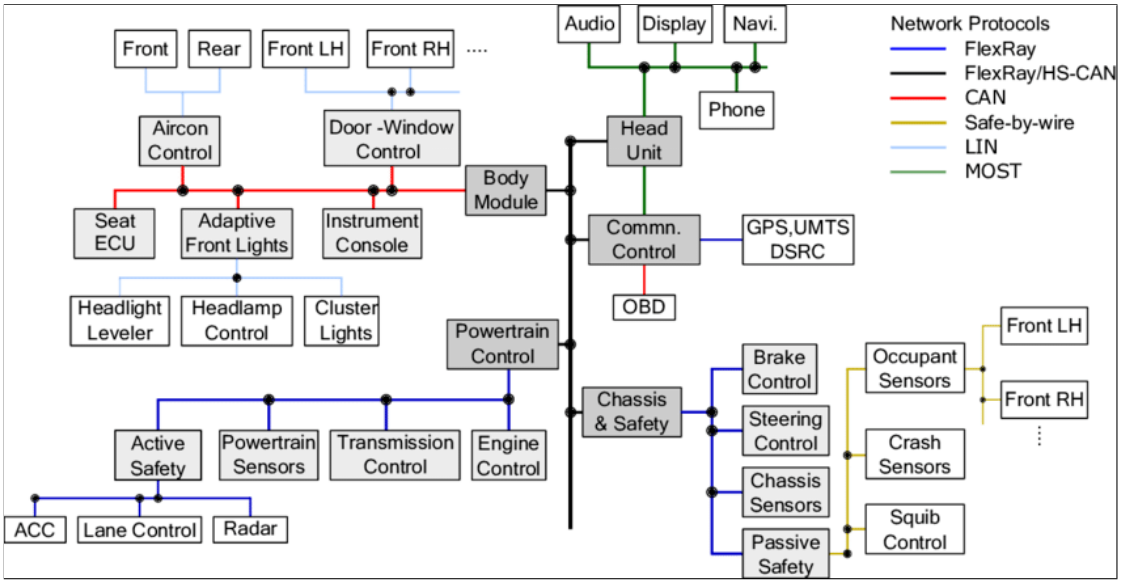


Figure 1.2 Detailed in-vehicle network architecture [11]

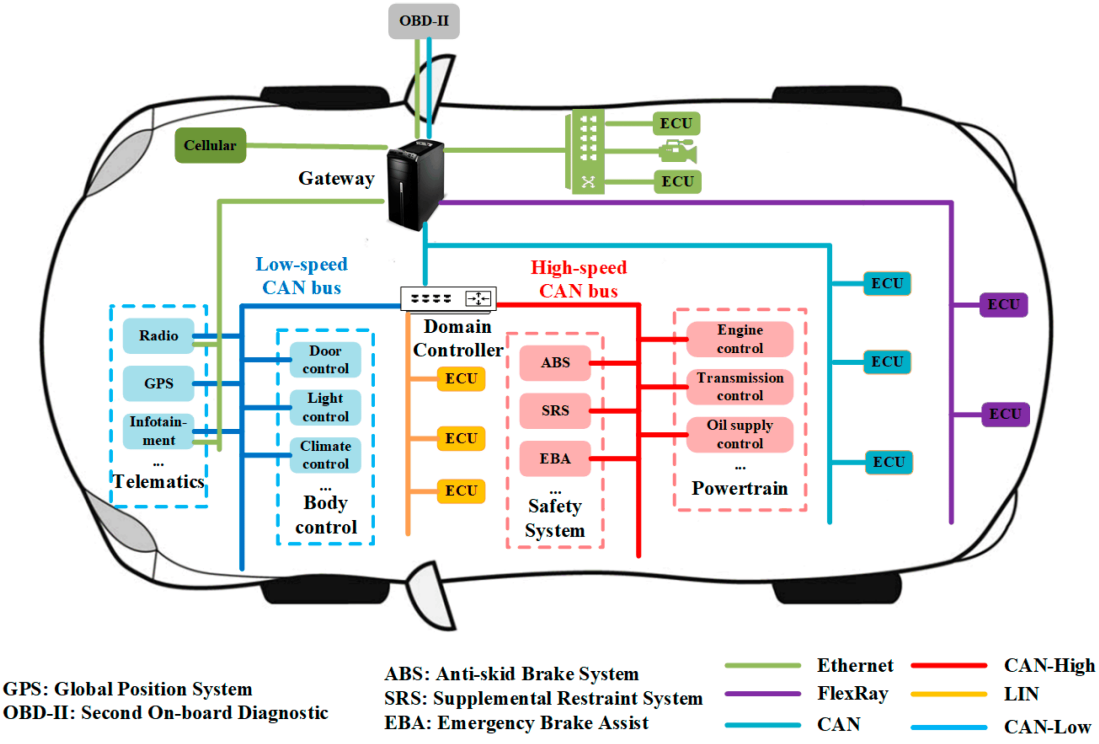


Figure 1.3 Architecture of the IVNs [12]

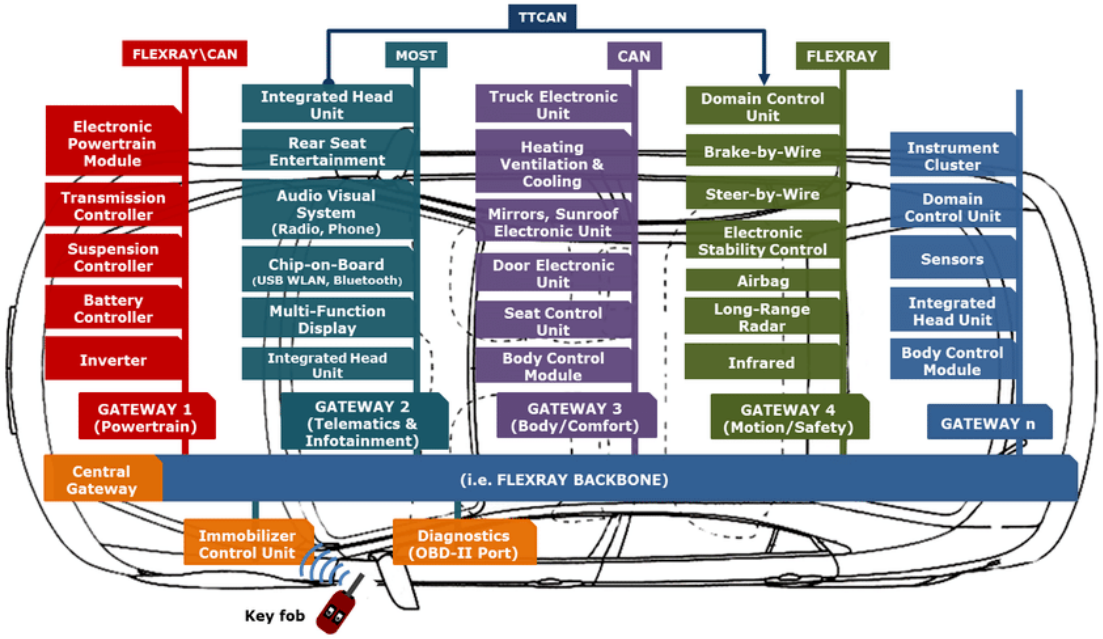


Figure 1.4 Paradigm of the automotive network architecture. [13]

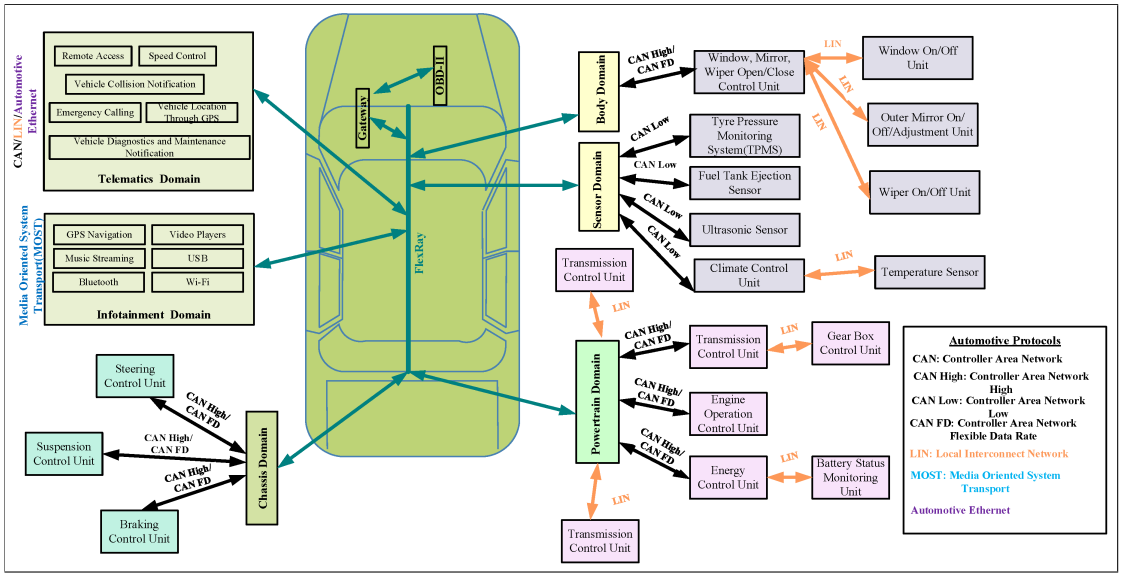


Figure 1.5 Network Architecture with Automotive Protocols [14]

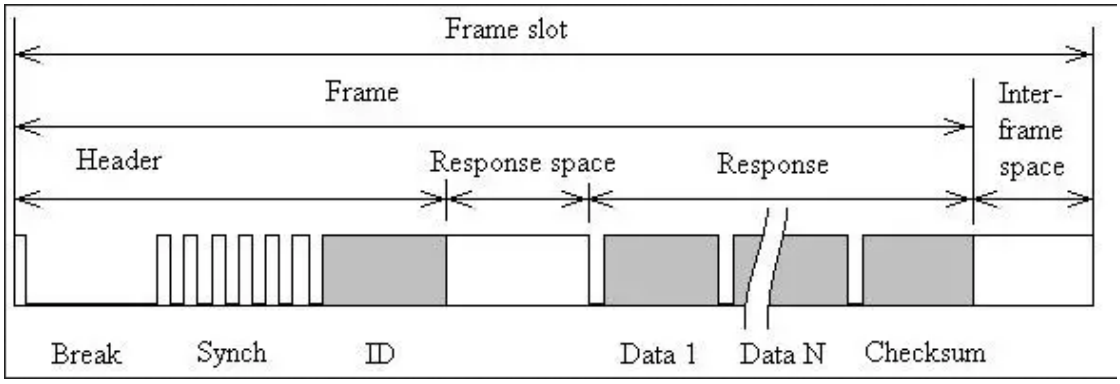


Figure 1.6 Example of LIN message [18]

1.1.2 LIN

LIN also known as the Local Interconnect Network was developed by the LIN Consortium (BMW, VW, Audi, Volvo, Mercedes-Benz, Volcano Automotive & Motorola) and its first release was in 1999. The latest version LIN2.2A was released in 2010 and is now widely implemented. [17]

The LIN bus is a complement to the CAN bus. It is a single-wire communication network that uses a master-slave architecture. The LIN master typically plays the role of the gateway to the CAN bus. LIN communication is quite simple in nature: the master node sends requests to the slave nodes, and then each slave node responds with data when a request is received. LIN bus message is a frame consisting of frame header and response (Figure 1.6). The master node sends a frame containing a header, and the slave node responds with a frame. The slave node fills the frame with data (response). [8]

LIN bus is suitable for applications where speed and fault tolerance are not critical because LIN is not as reliable as CAN. Compared to CAN, the speed rate is lower, about 1-20 kbit/s, but is more cost-effective. In terms of price, it is more advantageous due to the use of single-wire communication which is not required so many cable harnesses. Examples of LIN applications in the vehicle: window lifts, mirrors, wipers, air condition, and rain sensors. The role of LIN is as an interface between the sensor/actuator and the master ECU. ECUs are usually interconnected using the CAN protocol. [17]

1.1.3 MOST

Media Oriented Systems Transport was developed by the MOST Cooperation, which was founded in 1998. The development of MOST technology was largely driven by the rapid evolution in the technology of infotainment in vehicles. This development was mainly related to the increase in the number of devices that must communicate with each other in the infotainment system. These include devices such as a mobile phone, navigation system, voice control, radio, and others. To minimise driver distraction, all of these devices must work together and be able to be controlled from one central panel. To satisfy these requirements, the previously developed CAN was no longer sufficient due to the low bandwidth and the inability to send audio and video signals. As a result, MOST technology was developed to meet specified requirements. The first car that implemented MOST was introduced in 2001. [19]

There were multiple MOST versions, and the latest MOST150 supports high bandwidth, namely 150 Mbps. The latest version provides high-bandwidth audio and video signals without the overhead of addressing, collision detection/recovery, or broadcasting. As further enhancements, the MOST150 has an isochronous transmission mechanism to support extensive video applications and an Ethernet channel for efficient IP-based packet data transmission. [19]

In addition to the fact that MOST contains all layers of the OSI model, it also defines the interface to applications (e.g., AM/FM tuner) using function blocks [19]. Function blocks are a key part of the MOST standard. The function blocks contain all properties and methods of the MOST device (e.g., radio) that are necessary to operate the device. Communication with these function blocks is carried out via the application protocol. A more detailed description of the function blocks and communication within MOST is available from [19].

The interconnection of the MOST with other buses present in the vehicles is implemented by gateways. In practice, such gateways include, e.g., CAN – MOST or Bluetooth -– MOST. The first mentioned gateway is integrated into the Head Unit (Human Machine Interface) where, for example, the WheelSpeed signal is converted for display via the Head Unit to the driver. For instance, Bluetooth – MOST is used for hands-free in which the mobile phone communicates via Bluetooth with the hands-free unit. In the chapter "Gateways to MOST" from [19] it is explained in detail exactly how the communication between the individual components takes place.

1.1.4 FlexRay

FlexRay is another widely used automotive standard that was developed by the FlexRay consortium. The reason for FlexRay was the arrival of modern applications for which the event-triggered arbitration used in CAN is not suitable. These modern applications such as steer-by-wire or brake-by-wire, commonly referred to as x-by-wire, replace hydraulic control (e.g. steering or braking system) using ECUs and sensors. [9]

FlexRay technology offers, compared to CAN, high bandwidth up to 10Mbps, and is a time-triggered network protocol and also has TDMA behaviour (support is also available for event-triggered message [9]). The mentioned features are important for use in modern applications. These advantages also have the consequences of the fact that FlexRay nodes are more expensive than CAN nodes. The solution to these issues is to implement CAN in systems which do not require high speeds and are not as critical, and hence gateways are again needed to ensure conversion from CAN network to FlexRay network and vice versa. [5, 9]

FlexRay is therefore used in applications requiring high data rates and in safety-critical systems (x-by-wire). It is also suitable as the backbone of an in-vehicle network, to which other buses such as CAN, LIN, and MOST are connected via gateways.

1.1.5 Automotive Ethernet

Authomotive ethernet (AE) is a standard of physical layer based on 802.3 Ethernet technology. Ethernet technology is inexpensive, fast, and flexible. The main reason for the development of automotive Ethernet was the high bandwidth offered by the technology, which is superior to previously used technologies. Although the advantage of original automotive technologies such as CAN and MOST is that they were designed with the internal vehicle network in mind, as new systems become more demanding in terms of speed and bandwidth, the original automotive technologies are no longer sufficient. This has resulted in the adoption of technologies based on Ethernet. Ethernet has thus gained enormous importance as an in-vehicle network for connecting devices. [5]

Despite the positives mentioned above, Ethernet has disadvantages for some automotive applications, the disadvantage is that Ethernet is event triggered, which is not suitable for safety critical applications such as "x-by-wire". This was why modifications to the original Ethernet were necessary. [5]

Several approaches are proposed to overcome the disadvantage mentioned, and most extended standards are as follows: Time Sensitive Networking (TSN) and Time-Triggered Ethernet (TTEthernet) [20].

TSN is based on Audio Video Bridging (AVB) technology. TTEthernet is an improvement of Avionics Full Duplex Switched Ethernet (AFDX). The distinction between these standards and

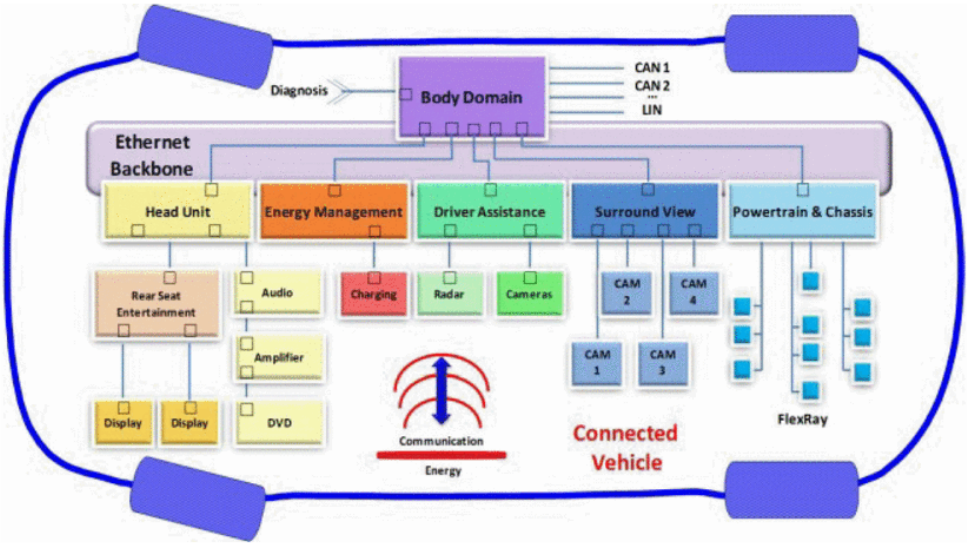


Figure 1.7 Ethernet backbone in domain architecture [22]

a more detailed description can be found in [20].

The first application of automotive Ethernet was the use of an OBD and firmware update procedure. This was followed by other applications such as infotainment, cameras, and sensors. Another use of AE is as the backbone for the vehicle's internal network, illustrated in Figure 1.7. [21]

1.1.6 Interfaces

Here, a brief overview of the interfaces[1] with which the common user comes into contact during normal vehicle operation is provided. The purpose of this section is to provide a better picture of the interfaces of a modern conventional vehicle. In contrast to this, the following subsection (Addition in EV networks), lists changes in electric vehicles, that is, the added components and the new associated interfaces.

Interfaces for the general user are primarily related to infotainment. The user can connect to the car primarily via USB, smartphone connection (i.e. Android Auto and Apple CarPlay), WiFi, or Bluetooth. Another interface with which the user often interacts is the user's ability to unlock and lock the car, for example, using "Central Locking" or "Keyless Entry System".

Another interface with which the user comes into contact through the service station is the diagnostic interface (OBD-II port). In this case, the user does not interact with it directly, but knows that this interface exists.

The information presented here is intended to serve as a preface for the following subsection, in which I discuss what electric vehicles contain in addition to the components and interfaces mentioned above. Interfaces with which the normal user does not come into contact during normal vehicle operation, but can serve as an intrusion point, are listed in the Attack surface section.

[1]The term "interface" in this context means something through which the user interacts directly with an electrical component of the vehicle

1.1.7 Addition in EV networks

From the user's point of view, the most significant difference is the change in the powertrain of the car and the associated differences. In this subsection, I discuss and describe these differences.

The main differences include the switching of fuel (i.e., the switching from diesel and petrol (or other fuels) to electricity). Along with the transition to the new fuel, a new refuelling mode has also emerged, namely charging.

A more detailed description of charging is given in section 1.3, but for the purposes of this subsection, it is sufficient to know that there are two basic types of charging: AC and DC. AC charging requires an AC to DC converter inside the vehicle, called an on-board charger (OBC), which is why the AC charging port is often referred to as an OBC port. Conversion is necessary because the car battery can only be charged by DC. The second type (i.e. DC) is used for fast charging, so here the OBC is not needed for conversion. A simplified diagram is on Figure 1.8, which shows the AC and DC charging and thus the use of OBC.

The charging inlets can be two to distinguish AC and DC or one combined (AC & DC). In addition to electricity, communication between the EVCC (Electric Vehicle Communication Controller) and SECC (Supply Equipment Communication Controller) takes place through these charging ports. This brings me to the most important point: the charging ports provide an interface to communicate with a component located in the vehicle internal network.

As part of the fuel change, some components had to be removed, and others added. The removed components are not relevant to this work; they will not be listed here. It is primarily the removal of components related to the type of fuel (engine change, tank removal, etc.). In particular, the addition of components is important as with their addition new interfaces are created. Through these interfaces, it is possible to communicate with the in-vehicle network.

Therefore, the high-level components that were added as a result of the switch to electric drive are given below. Considering the topic of this thesis, the most important added components are as follows: battery management system (BMS), electric vehicle charging controller (EVCC, also known as vehicle charging control unit (VCCU)), on-board charger (OBC), vehicle control unit (VCU), battery, charging inlet. [24]

The BMS monitors and controls the performance, state of charge, and health of the battery pack to optimise efficiency, ensure safety, and prolong overall battery life. The electric vehicle charging controller (EVCC) manages communication with the supply equipment charging controller (SECC). In electric electric vehicles and hybrid vehicles, the VCU acts as a domain controller that reads sensor signals, balances system energy, optimises torque, controls motor, and provides control of the OBC [25]. The OBC is responsible for converting the incoming AC into the outgoing DC to charge the EV battery. The battery is used to store fuel (in this case, electricity). The charging inlet is used as an input for the charging plug of the charging station. All of the key components (which were added in the electric vehicle) are directly or indirectly related to the charging process.

To clarify this, I present here a general simplified diagram (Figure 1.9), where the new components (except EVCC) are shown. EVCC can be a component of, for example, VCU or as a standalone device according to the manufacturer's design [26]. The Figure 1.9 demonstrates the interconnection of components in the vehicle's internal network. The individual components typically communicate with each other via CAN. Furthermore, Figure 1.9 shows that the OBC port allows access to the vehicle's internal network. I also attach a second diagram (Figure 1.10) from [27] showing the EVCC without OBC.

In this subsection, I demonstrated how the internal network of an electric vehicle is affected as a result of the addition of components. Due to the need for charging, a new interface is created through which to communicate with a component in the car's internal network. This component in is called an EVCC / Charging ECU / Charging Control Unit (CCU) / Vehicle Charging Control Unit (VCCU), depending on the resource. In the next parts of this work, I will focus more on communication with this component because it is also accessible through the

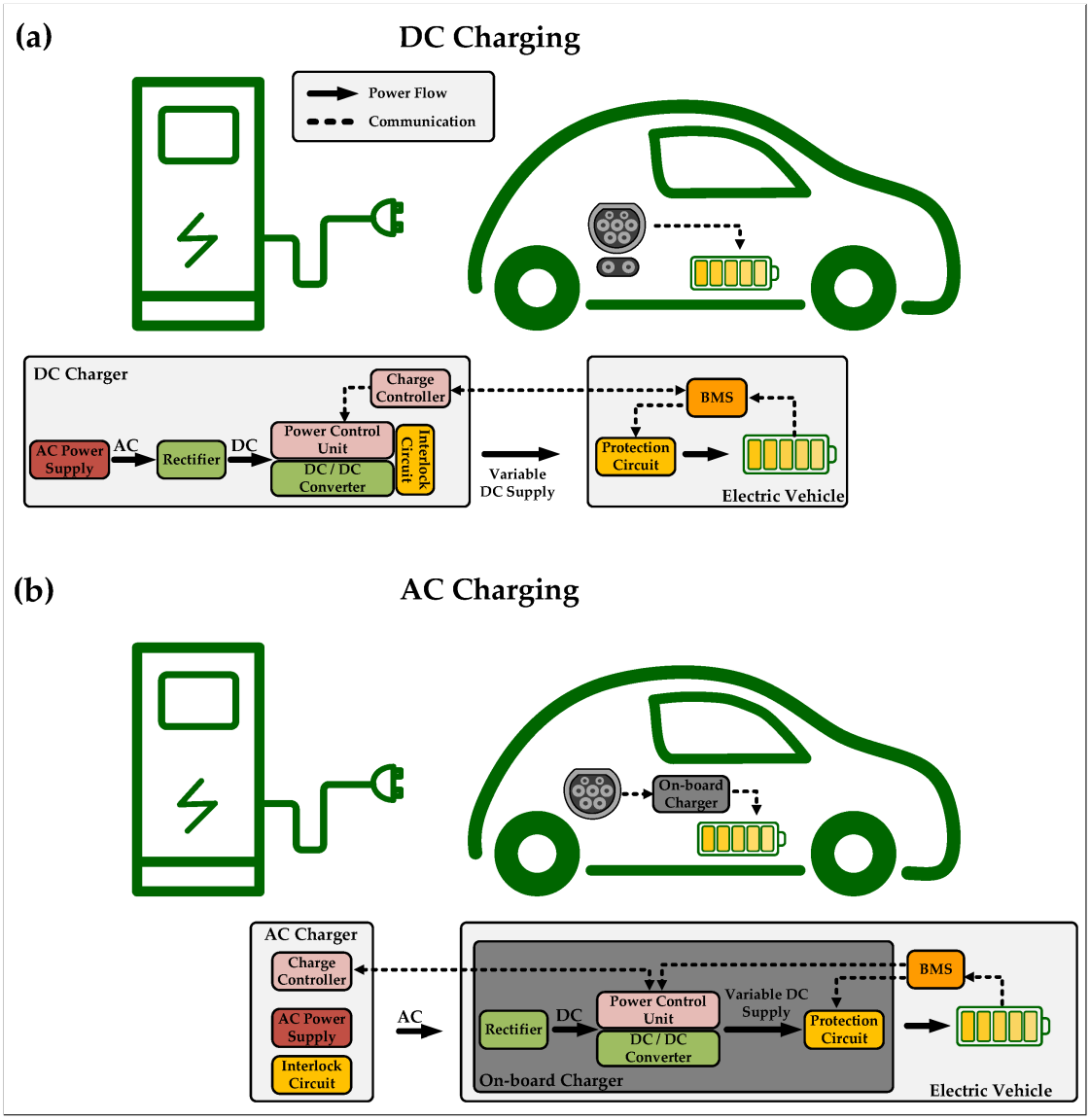


Figure 1.8 AC vs. DC charging [23]

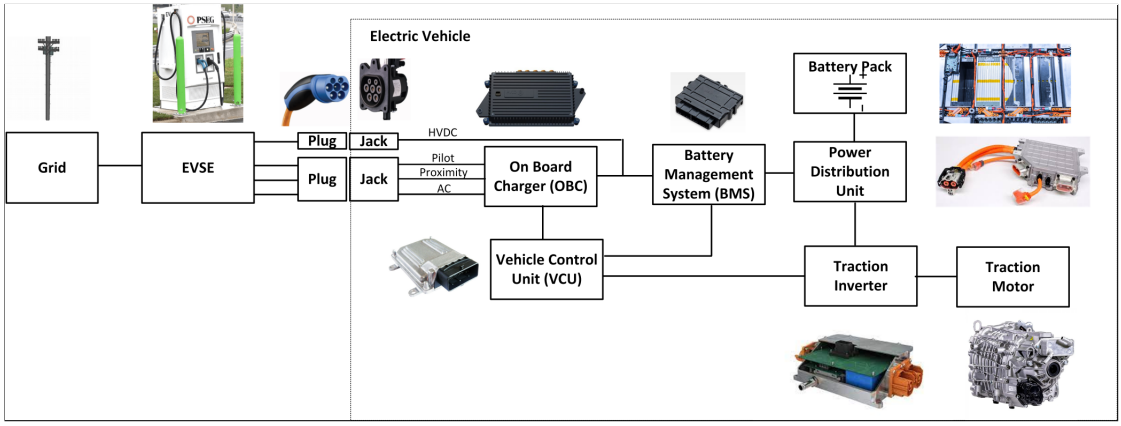


Figure 1.9 EV new network components [28]

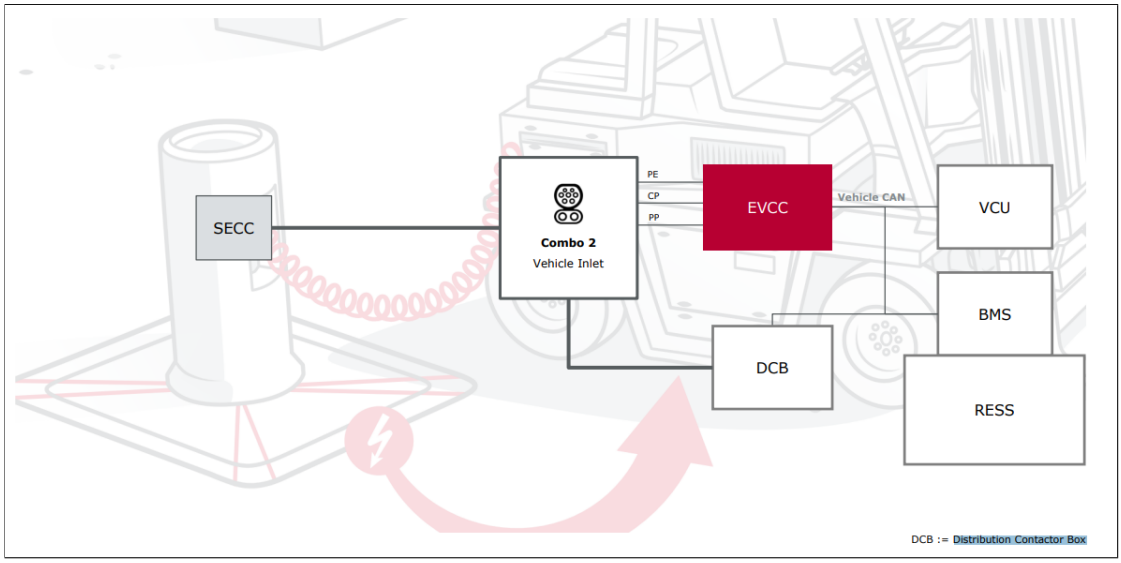


Figure 1.10 Vector charging architecture overview [27]

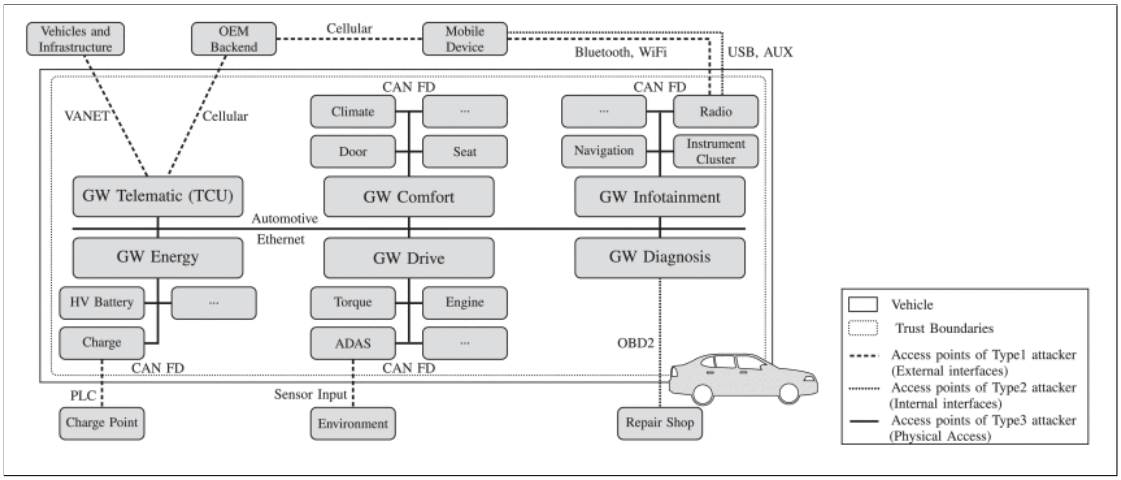


Figure 1.11 Generic EV architecture [29]

OBC port, as you can see in Figure 1.9. Finally, I attach an image (Figure 1.11) of the generic architecture for EV.

1.2 Attack surface

In this section, I focus on the attack surface of modern vehicles. In the following, I describe the other attack vectors that arise with the development of electric vehicles. An attack vector is a path that an attacker uses to intrude or compromise a system. The attack surface is just the set of these attack vectors. Therefore, the attack surface summarises the ways in which an attacker can attempt to gain entry into a system.

The basis of this section builds on section 1.1, where I mentioned that, with the advent of new technologies, modern vehicles generally contain more and more electronic control units, sensors, and the possibility of connecting various devices to the vehicle network (e.g. connecting a smartphone). These improvements mainly aim to improve the safety of roads and improve the user experience. However, the integration of smart technologies, such as ADAS (Advanced Driver-Assisted System), aimed at improving safety, introduces new devices and ways for an attacker to attempt to attack the vehicle.

The constant improvement of modern vehicles, particularly in terms of safety and user experience, has inadvertently led to an increase in potential attack vectors. The introduction of various wireless technologies in modern vehicles has made them more vulnerable to attacks, as they are no longer the closed systems they once were. As a result, they provide significantly more means for attackers to gain entry into these systems.

In subsection Classical vehicle, I present typical examples of technologies and devices used in modern vehicles. On the basis of this information, the possible attack vectors for conventional vehicles[2] are further described and categorised. This basis is the same for modern vehicles in general, i.e. both internal combustion engine vehicles and electric vehicles. In addition, I focus on the technologies used and possible attack vectors specific to electric vehicles. Both subsections extend the foundation provided in In-Vehicle networks. For better continuity with the description of the attack surface, I provide more detailed examples of the technologies and equipment used in this section.

[2]Term conventional vehicles in this context means vehicles with internal combustion engines.

1.2.1 Classical vehicle

Modern cars offer access to the in-vehicle network via the OBD-II port. The OBD-II port provides the possibility to read vehicle diagnostic information. As this port directly provides access to the internal network of the vehicle, it is one of the most attractive interfaces for an attacker. Nevertheless, an attacker must have physical access to the vehicle to access it, since the OBD-II port is located under the steering wheel in the driver's seat (in most vehicles). [30]

Another widely used technology is remote vehicle unlocking. For example, the traditional remote key fob which uses radio frequency (RF) signals. The user presses a physical button on the device and the device sends a signal to the receiver in the car, then the car is unlocked/locked. Another remote unlocking option that does not require the physical press of a button is referred to as a remote keyless system (RKS), an example being RFID car keys. This technology requires only the approach of a certain distance, followed by pressing, e.g., the car door handle. [30]

There is also an entertainment system in modern cars that can include a USB or CD input. These inputs give the user the capability to synchronise their mobile device data with the in-vehicle system or play the content in the vehicle. Modern vehicles offer wireless connectivity through Bluetooth to connect smartphones. This connection enables the use of Apple CarPlay or Android Auto for in-car infotainment. Other wireless technologies through which modern cars can communicate include WiFi, cellular networks (LTE, 3G, 4G, and 5G). WiFi and cellular networks are also used for digital radio, traffic reports, and global positioning system (GPS) purposes. The telematics unit allows, among others, the access to the Internet, the transmission and reception of telematics data and the communication with back-end servers. Another application of wireless technology is the use of over-the-air (OTA) software updates. Another important technology is Dedicated Short Range Communications (DSRC), developed to establish communication between a vehicle and another vehicle (V2V) or vehicle and infrastructure (V2I). DSRC facilitates communication between vehicles, allowing them to exchange information while driving. For instance, if the first vehicle encounters an obstacle and brakes suddenly, it can immediately transmit this information to surrounding vehicles via DSRC, allowing them to respond more quickly to the situation. DSRC uses RF, so radio signals have more applications in the automotive field.[30]

The use of wireless technologies is causing the network subsystem inside vehicles to become more connected to the external world. This provides a wider attack surface for the internal network of the vehicle than the legacy network of the vehicle. This allows the attacker to exploit multiple attack vectors to attempt to penetrate the internal network of the vehicle.

However, not only is the attack surface expanded due to wireless technologies, but the expansion is also due to the large increase in the number of different sensors and ECUs. Through physical access to these sensors (or ECUs) and direct injection of malicious messages, an attacker can attempt to penetrate the internal network of the vehicle. Therefore, the attack vectors are generally also ECUs, sensors, and actuators, specifically, for example, Tire Pressure Monitoring System (TPMS), ABS (Anti-lock Braking System), Engine Control Unit, ADAS sensors and ECUs, and others. [30]

Figure (Figure 1.12) illustrates the attack surface for a vehicle where individual units are connected via the CAN bus. This is analogous to vehicles that use, for example, FlexRay or AE as a backbone network. In case the attacker enters the in-vehicle network, the next steps and options of the attacker depend on the type of bus used and also on the ECU that is attacked/-compromised. For example, FlexRay or LIN requires a predefined schedule of communication with a node. This restriction limits the ability of an attacker to send messages over the network. On the other hand, CAN allows to add a new device as "Plug and Play", so an attacker can send messages with its priority anytime [6].

I have attached a summary table of attack vectors for a modern vehicle, in general. In addition to the attack vectors shown in the table (Table 1.1), there may be others. It should also be added that the attack vectors and their number always depend on the architecture of the

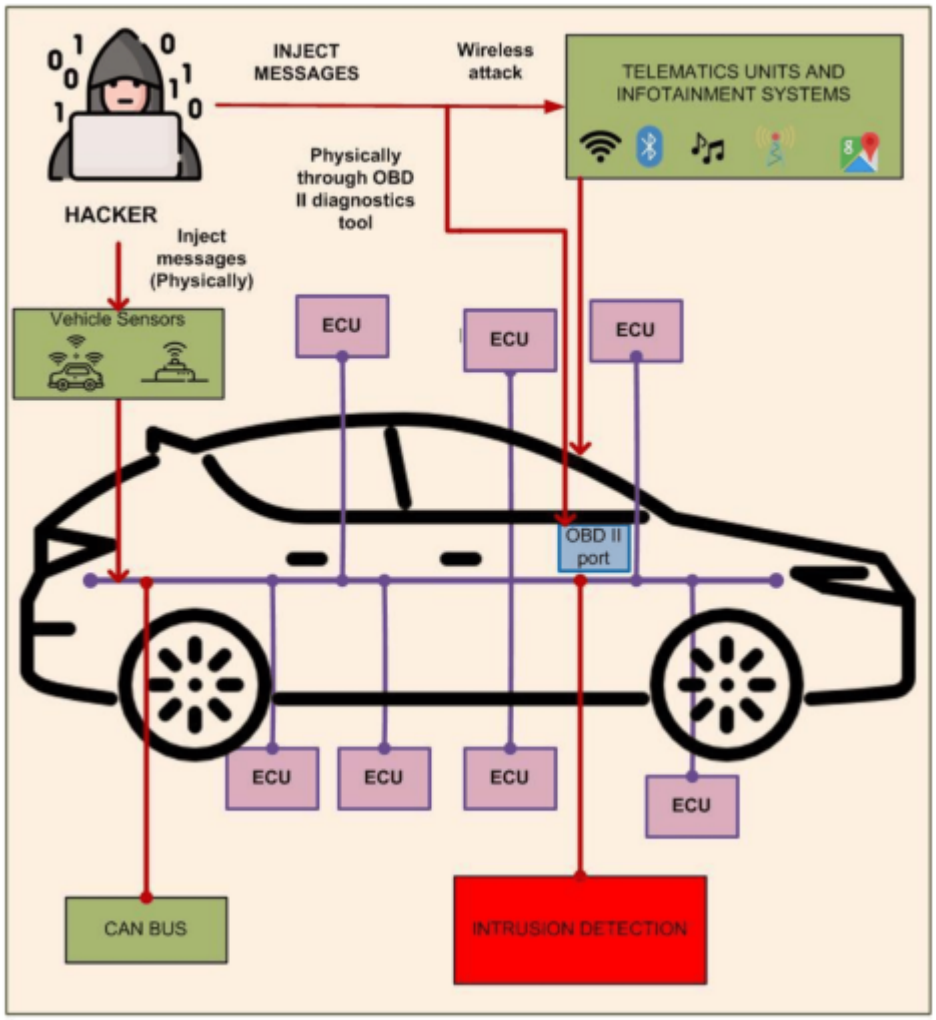


Figure 1.12 CAN bus attack surface [31]

Attack Vector	Type of access	Distance to car / acccess point
OBD-II port	Physical Access	Direct access to the port
Unmnounted ECU or sensor	Physical Access	Direct access to the ECU/sensor
CD	Physical Access	Direct access to the CD player
USB	Physical Access	Direct access to the USB port
Bluetooth	Wireless Access	Near Field
RKS	Wireless Access	Near Field
DSRC	Wireless Access	Short Range
Cellular (3G, LTE, 5G)	Wireless Access	Long Range
OTA updates	Wireless Access	Long Range
GPS	Wireless Access	Long Range

Table 1.1 General attack vectors of modern vehicle [6, 30]

vehicle as well as on the technology used.

As an illustration of the possible consequences of abuse, I attach practical examples from various authors. The first example comes from C.Miller and C.Valasek, who documented their remote hacking and stopping a Jeep Cherokee driving down the highway in their paper [32]. Other authors discovered several vulnerabilities in BMW models, one of the vulnerabilities that led to wireless compromise of control units connected to the CAN network [33]. Other examples mentioned in [30], such as remote hacking of the Tesla Model S, injection of malicious firmware via OTA, etc.

This subsection provides a general overview of the attack surface of modern vehicles. In the next subsection, I extend on this foundation by adding a discussion of possible attack vectors specific to electric vehicles.

1.2.2 Electric vehicle

In general, electric vehicles share many attack vectors with conventional vehicles. A conventional vehicle, unlike an electric vehicle, has only a diesel or gasoline inlet through which no communication flows, only fuel. In contrast, an electric vehicle uses charging to replenish fuel (electricity), in which data are exchanged between the station and the vehicle, in addition to electricity. Therefore, electric vehicles create another possible entry point for the attacker; this entry point is the charging inlet.

In addition to the charging input, another threat is hidden in the electric vehicle battery and its associated BMS. According to reports of the battery catching fire, the battery may be the target of attackers who may attempt to cause damage or injury to drivers and passengers. The BMS is responsible for monitoring and controlling the voltage and temperature of the individual cells that make up the whole electric car battery. It also has the responsibility for selecting the charging strategy and controlling the energy flow into and out of the battery. It is also responsible for selecting the charging strategy and controlling the flow of energy to and from the battery and serves as a preventive mechanism against battery damage.[6]

If an attacker gained control of the BMS, he could control all functions associated with the battery. Using this control, an attacker could ignore various critical values (e.g., over voltage and current), resulting in damage to the battery or even in a flame hazard. In order to prevent ignition, the batteries have various safety mechanisms, pressure release or burst valves, which reduce the pressure and thus reduce the risk of ignition. Other mechanisms include hardware sensors that disconnect the battery cell when a certain temperature is reached. The mechanisms mentioned above may prevent a flare-up, but, if the attacker controls the BMS, it can still cause injury to passengers. For example, the attacker disconnects the battery from the engine via the BMS while the car is accelerating. These threats are important to consider when designing a

vehicle network to protect the BMS from unauthorised access. [6]

Another potential problem with the arrival of batteries is the counterfeiting of batteries, which can cause financial difficulties for suppliers and manufacturers in particular. Counterfeit batteries also carry high safety risks as they often do not meet safety requirements and do not, for example, possess the aforementioned flame-retardant mechanisms. In order to prevent the use of these batteries, an authentication implementation is essential to verify that the installed battery is genuine. [6]

In [6] the charging input is listed as another threat to electric vehicles, which serves as another intrusion point. The charging input is an essential part of an electric vehicle, which for the first generation of electric cars served only as electrical connection. The original purpose of the charging inlet has evolved with the advent of the next generation of electric vehicles and now allows communication between the charging station and the electric vehicle using various communication protocols (discussed in more detail in the following section, called Charging). For example, the CHAdeMO standard is used in Japan, which uses a CAN bus for communication between the EVSE and the EV. In contrast, the IEC 61851 standard used in Europe uses power line communication.

In particular, communication using the CAN protocol can be high risk if it is directly connected to the in-vehicle network and no message filtering is used. It is common practice in obsolete CAN-CAN gateways not to implement message filtering. The above issues relating to the use of the CAN protocol may result in an attacker being able to reprogram the ECU on the in-vehicle network or program a key to unlock the vehicle via the charging inlet. Another critical threat is man-in-the-middle, in which an attacker connects between a charging station and the charging input of a car and can then intercept communications or modify packets. [6]

The new ISO 15118 standard, which among other things proposes the use of Transport Layer Security (TLS), should contribute to better security of the charging inlet and charging communication [6]. More details of ISO 15118 standard are described in section ISO 15118. This protocol also has various pitfalls, e.g., that for some situations the use of TLS is only recommended and not required by the standard [34]. Again, this may result in unauthorised reading of sensitive data transmitted between the charging station and the electric vehicle.

In this regard, charging inlet is one of the main attack vectors for EV. This brings me to the OBC device, which is directly accessible through the charging inlet (see image Figure 1.9). If the vehicle is supplied with a separate AC and DC input, then the AC charging input can be referred to as the OBC port. In the case of combined Charging System (CCS) plugs, one input is used for both types of charging.

Important for this work is the fact that the OBC device is accessible through the charging port, so in this case, the OBC port can be generalised to a charging inlet. The main point is that, during charging via the charging port, the electric vehicle communicates via a controller with the charging station. The controller can be integrated into an OBC device or another ECU such as a VCU, BMS, or CCU, but its functionality remains the same [24].

In this thesis, the chosen attack vector is the OBC port (generally the charging inlet), through which I target the controller responsible for communication between the charging station and the vehicle, often called the Electric Vehicle Communication Controller (EVCC). Therefore, it implies that the implemented tool does not only serve for on-board charging (according to the topic) but also for off-board charging because the communication is always handled with the mentioned controller (on the EV side). To better understand this idea, the image Figure 1.13 illustrates how OBC, DB (Distribution Box) BMS and EVSE can be connected, in general. In the figure, you can see the interconnection of the components through the in-vehicle network, as well as the power flow paths and the communication path between the charger and the vehicle.

In the context of new threats to electric vehicles, the last item mentioned in [6] is x-by-wire technology. For example, this technology is intended to increase the range of an electric vehicle by braking recovery (brake-by-wire). In addition to its advantage of increasing the distance covered, [6] noted that this system can also pose a major security risk, as it theoretically allows

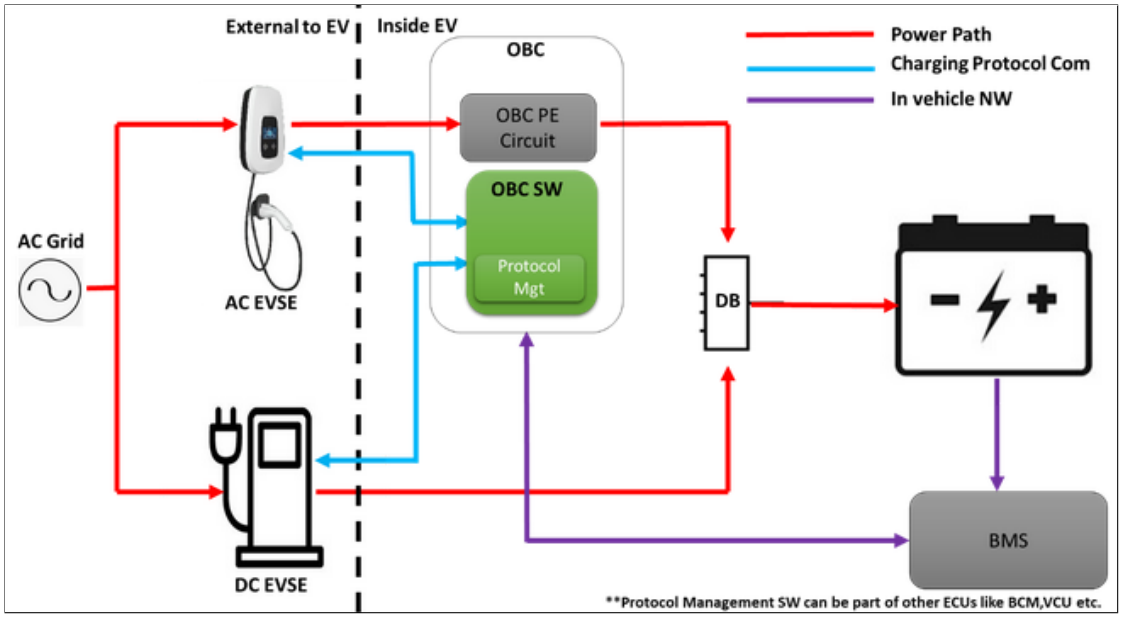


Figure 1.13 EV Charging High Level Overview [35]

an attacker to remotely control the vehicle or disable the brakes. In order to minimise this risk, it is important to implement a secure network within the vehicle, isolated from the environment, and support for authentication [6].

1.3 Charging

The development of electromobility and related technologies is also influenced by the drive of the European Union to be climate neutral by 2050 (European Green Deal) [36]. Electric mobility is evolving at an ever-increasing pace with associated benefits such as improved efficiency, extended range, larger battery capacities, etc. With the increasing market, car manufacturers are continuously pursuing new technologies and developing improved electric vehicles to make them a complete replacement for current cars with internal combustion engines. This mainly concerns the range of electric vehicles, with the associated development of faster charging technologies and batteries with higher capacity while maintaining the lowest possible weight. These developments are constantly pushing the boundaries and open new possibilities for electric vehicles. Innovations are not limited to electric vehicles alone, but affect the whole charging infrastructure.

As an alternative to existing vehicles with internal combustion engines, ensuring the same refuelling capabilities (especially refuelling speed) of electric vehicles involves the need to optimise the charging process. This is one of the major challenges faced by charging station and vehicle manufacturers. In terms of the future of smart grid, the charging process is more complex than considering basic data such as available charging station power, battery status, and charging demand. Consequently, more complex information must be exchanged during the charging process, such as whether the vehicle needs to be charged quickly, how authentication and payment will take place, what energy sources are available, and more. [37]

This optimisation of the charging process is closely linked to the development of charging interfaces and standards to ensure compatibility between different EV models and charging stations. For these reasons, several international standards have been and are currently being developed to define and specify the communication between EV and EVSE. Furthermore, the standards specify the technologies used for communication and the charging interfaces, etc.

Challenges associated with the development of new charging standards include not only technical parameters and compatibility, but also security and safety issues, dependency on different energy networks, and the need to adapt to market dynamics and increasing user demands. The key standards to ensure interoperability between the vehicle and the charging infrastructure are as follows: IEC 61851, ISO 15118, DIN 70121 [38]. The term "smart charging" is used for charging that adheres to standards such as ISO 15118 and DIN SPEC 70121. Smart charging refers generally to advanced technologies and systems which improve the efficiency, safety and convenience of the EV charging process. Smart charging has the primary goal of creating a very efficient, flexible, sustainable and low-cost environment for EV charging using intelligent charging management features. One example from the ISO 15118 standard is the method called Plug and Charge (PnC), which simply plugs an EV into a station to provide secure identification and authorization for charging [39].

In order to provide a better understanding of the types of charging, charging methods, charging standards used, charging interfaces, it is worth to describe the things mentioned in this section. In this section I therefore focus in particular on the following: what are the types and methods of charging; what are the charging interfaces; what are the standards used in the charging process. What communication looks like during charging is included in a separate section called Charging Communication, because in the context of this thesis, it is necessary to separate this topic in order to better decompose the different parts of the communication, so it is for better clarity.

1.3.1 Charging Methods

Generally, charging can be divided into two types of charging: conductive and wireless charging. This division is based on the technology used to connect and transmit electricity, i.e., the first type requires a connection by wire (physical connection) between two entities and the second does not use physical connection of entities, as the name of the second type suggests. [40]

Another way to refuel an electric vehicle is by battery swap technology [41]. This technology is not mentioned in [40], probably because it is not charging per se, i.e. energy transfer, but the exchange of a discharged battery for a charged one.

Based on [40], we can consider bidirectional charging as another charging method. This method differs from the above methods not on the basis of transmission type but on the direction of transmission. Bidirectional charging enables energy transfer in both directions, not only from the charging station to the vehicle but also from the vehicle to the charging station. This allows the vehicle to function as a temporary and mobile power source, for instance during a blackout. [40]

Conductive charging

The first type presented here is conductive charging, which is a constant in the field of charging electric vehicles. This method requires a physical connection between the charging station and the charging inlet of the electric vehicle through a conductor. It is a reliable and proven method that forms the basis of the charging infrastructure. The undeniable advantages are mainly simplicity and low cost, as plugs and sockets are used to connect the charging station and the vehicle. The cable used for charging connects the power source plug to the electric vehicle socket.

In this method, we further divide the charging sources for electric vehicles into the following: wallbox, charging station, pantograph [40]. The first possible source is a wallbox, it is a compact EVSE that usually supplies electricity in the form of AC (typically $<= 22$ kW), but now wallboxes also support DC charging (typically < 50 kW). Wallboxes are commonly used in private or semi-private locations. In general, a charging station is larger than a wallbox and also offers a DC charging, as opposed to a wallbox. Another distinction between charging stations and wallboxes is their location, which is primarily in public areas. Charging stations are therefore designed to

withstand various damage types such as wind gusts, inclement weather or vandalism. Charging stations usually have the capacity to charge multiple vehicles at the same time and provide more charging power (often exceeding 100 kW) than wallboxes. [40]

The last source of charging according to [40] is the pantograph, which is mainly used for public transport buses and trucks as a temporary connection for DC charging. It is a device used as a mobile receiver of electricity. Pantograph charging is standardised as ACDP (Automatic Connection Device Pantograph) in ISO 15118-20. There are two types of pantograph, the first is the inverted pantograph, which is part of the charging mast and is connected to the contacts on the roof of the vehicle. The inverted pantograph has the advantage for the electric vehicle that the weight of the device is carried by the charging station. The previous advantage is associated with the disadvantage of greater complexity on the charging station side and higher risks of failure. The first type is standardized as OppCharge (DC standard for opportunity charging) and in ISO 15118-20. In the case of the first type of pantograph, the use of WLAN technology is mandatory for the communication between the EV and the charging station. The second type is the Roof-mounted pantograph, which is located on the roof of an electric vehicle and connects to a compatible charging mast using the ISO 15118-2 and ISO 15118-20 standards. For the second type, WLAN communication is not required, which can be considered as an advantage. The disadvantage of the second type of pantograph is that the pantograph weight increases the weight of the entire vehicle and thus reduces its range. [40]

Moreover, conductive charging is divided into two categories of charging according to the type of current transferred, which are AC and DC charging. Depending on the type of electricity used, a distinction can also be made between on-board and off-board charging, depending on where the conversion from AC to DC takes place. Current conversion is necessary because electric vehicle batteries can only be charged using direct current. The conversion device can be located inside the vehicle, in which case it is called an on-board charger. If the conversion from AC to DC takes place in a power supply (e.g. a wall box or a charging station), the device is located in this power supply and is called an off-board charger. In the case of DC, and therefore the use of an off-board charger, real-time communication between the charging station and the vehicle is required. The purpose of this communication is to ensure that the current power requirements are communicated so that the battery is charged in the best possible way. With DC charging, power is essentially sent directly to the battery (via the BMS) and without communication, the battery could be damaged. I describe the AC and DC resolution more in the section called Charging Types. [40, 42]

Wireless charging

Wireless charging, also known as Wireless Power Transfer (WPT), is now mainly inductive charging. You may find that these terms are used interchangeably, depending on the source of the information. Wireless charging allows you to charge an electric vehicle without the use of cables. This type of charging works on the principle of an electromagnetic connection between coils. One coil is placed in the EVSE and the other in the EV, and communication is transmitted using WLAN technology. The efficiency of wireless charging is currently over 90 %. [40]

The advantages of wireless charging are convenience for the user and a more suitable charging method for self-driving cars. Disadvantages may include: higher cost than conventional charging stations, increased vehicle weight associated with the charging pad. [43]

Wireless charging can be divided into static and dynamic charging. Static charging takes place at a fixed location, while dynamic charging is possible while the vehicle is on the move. During dynamic charging, according to [43], it is possible to charge the vehicle with up to 20 kW at a speed of 100 km/h.

Battery swap

Battery swap technology has been tried before; for example, Tesla opened a battery swap station in 2014. Tesla later rejected battery swap technology in favour of expanding its network

of charging stations, called Superchargers. The problem with this technology is that there is no standard that specifies a single battery for all types of vehicles. Currently, each battery is manufactured to fit the architecture of a particular vehicle, making it difficult to deploy battery swap technology globally. [41]

A new startup, Ample, from California, introduces a simplified battery swapping station that reduces the swapping process to about 5 minutes. The company also makes modular batteries that can be used in electric cars of different sizes. These batteries can be swapped when discharged, allowing manufacturers to install them at the point of manufacture. [41]

Another company using battery swapping is the Chinese company NIO. The company uses a standard battery in all of its electric car models. The network of battery swapping stations is proprietary, i.e. only for NIO vehicles. The above examples show that battery swapping is becoming more attractive again. This attractiveness is also due to the fact that one of the main obstacles for electric vehicles is the significantly longer refuelling times compared to combustion engine vehicles. If this technology enables faster refuelling than existing solutions, it could become the most widely used. However, conductive charging is still the most widely used, as the cost of charging stations is another important factor, as is modularity. Conductive charging is not directly dependent on battery size standardisation. [41]

Bidirectional charging

The last charging method I'll introduce here is bi-directional charging (also known as bidirectional power transfer), which is independent of the technology used for power transfer (as above). The distinction here is in the direction of the power transfer, as conventional EV charging technologies use only one direction of transfer (from the power supply to the car battery). BPT allows energy to be transferred in the opposite direction so that it can be used by other devices. Depending on the destination, we can classify them as follows: vehicle to loads (V2L), vehicle to vehicles (V2V), vehicle to home (V2H) or vehicle to grid (V2G), as defined in the ISO 15118-20 standard. As I mentioned earlier, AC to DC conversion is required to charge the battery, and conversion in the opposite direction (i.e. DC to AC) is required to support power transfer from the EV. Smart charging is used to ensure efficiency during this process. [40]

Biderictional charging allows drivers who are infrequently driving and use only a fraction of the battery's capacity to use the battery as a backup power source. The battery can be used, for example, to store unused energy from a home photovoltaic system and use it at night. Another use could be to store electricity in the vehicle's battery for low tariff periods, saving costs. This energy can then be sold back to the utility for further distribution, which also saves money. [40]

1.3.2 Charging Types

In this section, I focus on conductive charging and, more specifically, on the two types of this charging method: alternating current (AC) charging and direct current (DC) charging. The current coming from the mains is always AC, but the battery of an electric vehicle only needs to be charged with DC. It is therefore necessary to convert from AC to DC, which can be done in the electric vehicle or in the charging station. Thus, the electric vehicle can be powered by AC (conversion in the vehicle) or DC (conversion in the EVSE). [39]

In the case of an AC charging station, the conversion is performed by a device in the vehicle. In contrast, for DC charging, the conversion from AC to DC takes place at the charging station. The AC and DC charging setup according to [39] is shown in the figure Figure 1.14. The Figure 1.14 shows the AC power flow through the on-board charger, which is located inside the vehicle and whose function is the conversion from AC to DC. For DC charging, the converter is located in the EVSE, labelled "Power Electronics" in the figure. In addition to the power flow, the figure also shows the communication path between the station and the vehicle. On the vehicle side, communication is handled by the "Charging ECU", whose location in the network and interconnection with other components in the vehicle depends on the specific architecture.

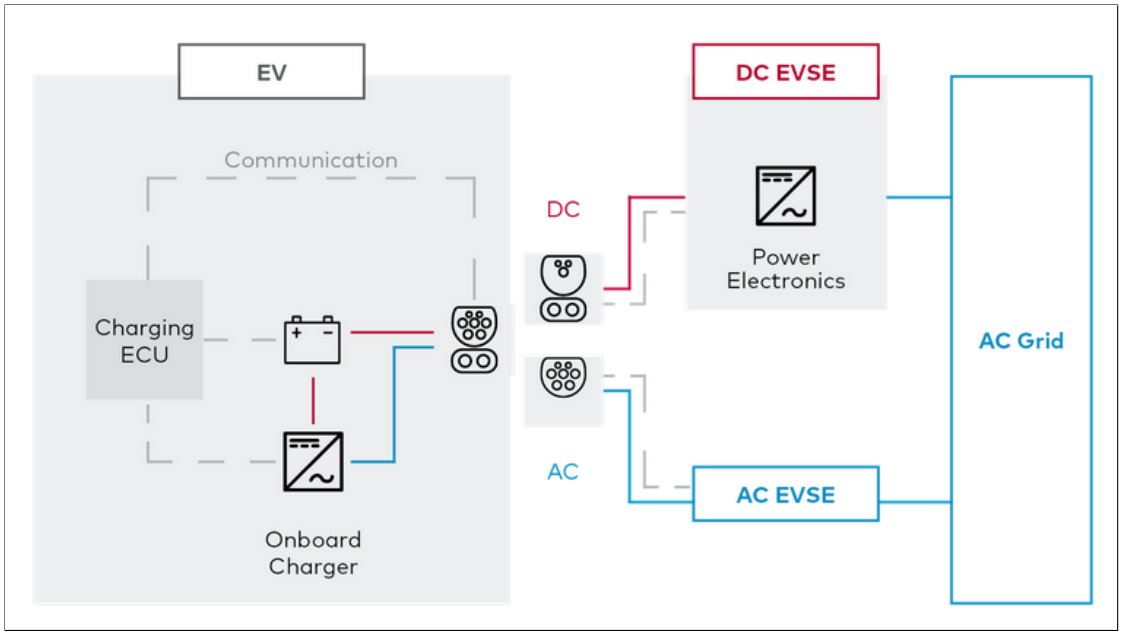


Figure 1.14 Simplified AC and DC charging setup [44]

The figure therefore serves as a simplified diagram to capture both types of charging and their basic characteristics. [39]

AC Charging

I will begin by describing the characteristics of AC charging, as this is currently the most common way of charging [39]. AC power comes directly from the mains. AC charging stations take current directly from the grid and send it to the vehicle. These chargers do not need a converter as they are placed inside the vehicle and therefore these charging stations are not as expensive. As mentioned earlier, AC chargers use an on-board charger to convert AC to DC and then charge the battery with DC. This is known as on-board charging (OBC) and the inlet is called an OBC port. A standardised vehicle inlet and charger plug are used to connect the station to the vehicle, more information on these interfaces can be found in the sub-section called Charging Interfaces. [39, 43]

The first advantage of this type of charging is the ability to charge the electric vehicle battery anywhere using AC power (from the mains) and an on-board charger. In terms of charging stations, this charging has the additional advantage of current conversion. The advantage of the charging station is that conversion takes place in the vehicle and thus does not need to incorporate additional complex electronics for conversion, with the associated lower cost of these stations. [43]

There are some disadvantages associated with AC charging and the location of OBC equipment. OBC units have to meet the requirements of light weight while providing acceptable performance. These two conflicting requirements lead to the disadvantages I list below. As the conversion takes place inside the vehicle in the OBC, its performance is limited by its weight and size requirements. According to [45], the OBC's performance is lower for single-phase AC than for three-phase AC. The power limitation is related to the fact that charging takes too long due to the low power. However, a longer charging time is better for the life of the battery. [43, 42]

From the perspective of an electric vehicle, the most important components for AC charging are the OBC, EVCC, BMS, VCU [24]. The placement and possible interconnection depends on

the architecture of the vehicle, but in general you can imagine the interconnection and placement of the components as shown in Figure 1.9 and Figure 1.10. In the first image (Figure 1.9) the EVCC is not marked because according to [24] this controller can be part of the OBC, BMS or VCU for example. On the other hand, the second image (Figure 1.10) shows the EVCC as a standalone device and its interconnection with other components involved in the charging process. Another important component is the charging inlet or port. For AC charging, this port can be referred to as the OBC port, as the charging station is connected to the OBC device (see Figure 1.9). The main components of the AC charging station are the AC power supply, the interlock circuit and the charge controller, as shown in Figure 1.8. The charging station and the vehicle are connected by a plug and a charging inlet or port. Depending on the charging interface used, the charging input may be compatible for AC and DC charging, i.e. one input for both types of charging plugs (connectors).

For example, for CCS (Combined Charging System) there is one inlet for AC and DC and only the charging connector is different, this is shown and explained in Charging Interfaces. In Europe and North America, according to [27], it is CCS that is used. For this reason, in this paper, which is intended to focus on the OBC port, I can generally consider the charging inlet to be an OBC port. So for European vehicles it is a single charging inlet. It is through this inlet that both the power supply and communication between the controllers, the SECC in the charging station and the EVCC in the vehicle, take place. The communication between the controllers is the most important for this thesis, because it is through the OBC port (generally the charging inlet) that I can communicate with the EVCC. Therefore, it is the access through the charging port (OBC port) that is important, not the location of the EVCC itself. So it does not matter if the controller is directly in the OBC device or in the BMS or VCU. [24]

To conclude on AC charging in general, I will describe the charging process using a diagram (Figure 1.8). As soon as the vehicle is connected to the station, the charge controller (SECC) starts communicating with the electric vehicle. It exchanges basic information about the connection, limits and fault conditions. It then supplies AC to the EV rectifier in the OBC, which converts the AC to DC. The Power Control Unit (in the OBC) then controls and adjusts the voltage and current for the DC/DC converter, which is then supplied to the battery. The PWC communicates with the BMS regarding battery status, voltage and current requirements. There is also a protection circuit in the OBC that can be activated by the BMS if any of the battery operating limits are exceeded, to isolate the battery and prevent damage to it. Communication between the controllers is normally via PWM (Pulse Width Modulation), which is referred to as low level communication. The ISO 15118 standard also defines high-level communication for AC. I discuss controller-to-controller communication in more detail in the Charging Communication section.

DC Charging

In contrast to AC charging, DC charging does not require an on-board power converter. In this case, the converter is off-board (outside the vehicle), which is why DC charging is sometimes referred to as off-board charging. The converter can be located directly in the charging station to provide DC power suitable for charging the battery. The EV battery is thus charged directly by the charging station without the use of an OBC device. An example of the interconnection of components for DC charging is shown in Figure 1.8. It should be noted that the OBC device does not perform any conversion, the current conversion is performed in the charging station and therefore communication between the SECC and the EVCC (which may be part of the BMS) is necessary to ensure correct charging conditions. Again, as with AC charging, it depends on the architecture where the communication controller (EVCC) is located in the vehicle. The simplified scheme (Figure 1.15) shows the EVCC as a separate component of the vehicle network that communicates with the BMS via the CAN bus. The BMS communicates the necessary information about battery demand to the EVCC, which in turn communicates with the charging

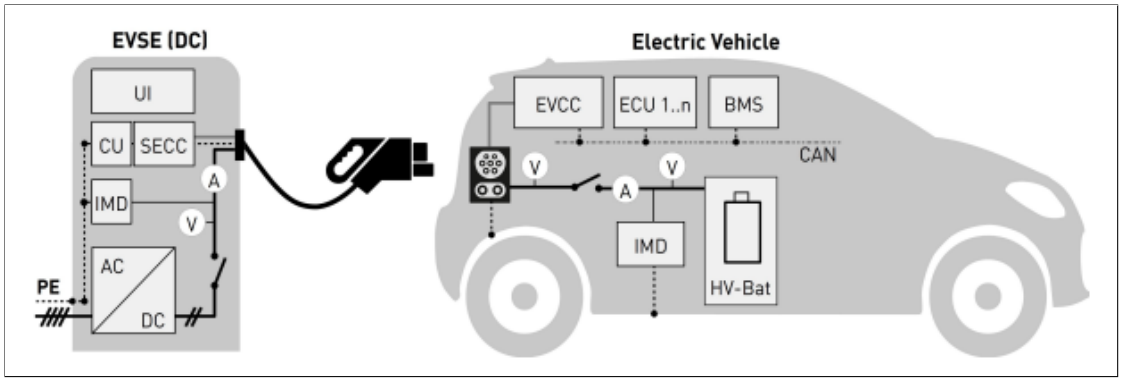


Figure 1.15 DC charging scheme [46]

station. [39]

Due to the position of the converter, it is possible to use larger converters with more power. Larger and more powerful converters allow for faster charging, which is why DC charging is also known as "Fast Charging". For these reasons, DC charging will play an important role in electric fleets, such as electric buses or logistics vehicles. [39]

DC charging, like AC charging, has its advantages and disadvantages. The undeniable advantage is that a DC charger can be designed for fast or normal (similar speed to AC) charging. Another advantage is that the converter is not limited by size or weight, as is the case with an on-board charger. With the capabilities of an off-board charger, another advantage is that it allows for fast charging, thus rapidly reducing the time required to charge the EV battery. The advantage of an off-board charger is that it offers high performance, which is associated with a large initial investment in a DC charging station. According to [42], a DC station with the same performance as an AC station is 5-7 times more expensive. Another disadvantage is the impact on the power system/grid depending on the higher power demand (needed for higher power output). As a disadvantage, the off-board charger and the BMS are physically separated, but reliable communication between these components is important to ensure valid charging conditions. [42]

According to [46], charging components can be divided into several categories: High voltage components, controllers, sensors & actuators, safety components. The most important components of this thesis are in the "Controllers" category, namely EVCC and SECC, which communicate via the charging inlet. Other controllers listed in [46] are: the EVSE control unit (CU), the battery management system (BMS) and other electronic control unit (ECU).

The above need for communication to maintain the correct charging conditions and also to ensure battery protection requires high level communication for DC charging. Basic signalling using PWM is no longer sufficient, so high-level communication is also required. Several standards define high-level communication and their use depends on the site and the charging interfaces used. According to [27], charging interfaces can be divided into three basic types: CCS, GB/T, and CHAdeMO. Combined charging system (CCS) uses PLC for high level communication transmission and the standards used are IEC 62196, IEC 61851, ISO/IEC 15118, DIN SPEC 70121, SAE J2847/2. The other two types, GB/T and CHAdeMO, use CAN for high-level communication. Tesla has developed its own standard for connectors (Tesla Supercharger) and charging inlets. According to [24], Tesla uses PWM for the communication of AC charging and PLC communication of DC charging and is compliant with DIN SPEC 70121. Within the EU, Tesla vehicles use CCS [47]. There are two types of CCS: 1 and 2, the former being used in Server America and the latter in the European Union. The GB/T standard is the exclusive standard in China, and CHAdeMO is used in Japan. [27, 48]

	N. America	Europe	Japan	China	Tesla (Except EU market)
AC	J1772 (Type 1)	Mennekes (Type 2)	J1772 (Type 1)	GB/T	Tesla Supercharger
DC	CCS Type 1	CCS Type 2	CHAdeMO	GB/T	Tesla Supercharger

Table 1.2 Country based connectors used [47]

1.3.3 Charging Interfaces

Basically, charging connectors are distinguished by the type of current, i.e. whether they are used for AC or DC charging. According to [47], it is also possible to distinguish which type of connector is standard in a given region. Based on [47], here is a summary table (Table 1.2) showing which connector types are used in the regions. The major e-mobility regions have developed their own charging systems and it is important to unify these standards with one solution, the Combined Charging System (CCS) for global AC and DC charging. CCS is used in the EU and North America, but the types of CCS plugs used are different, in North America for AC Type 1 and for DC Combo 1 (also CCS1 or CCS Type 1). The most important from the point of view of this thesis is the Type 2 connector for AC charging and the CCS Type 2 (or CCS Combo 2) connector for DC charging, because these two types are the standard in the EU. In the EU market, therefore, the input for AC charging is the Type 2 inlet or the Combo 2 inlet, which is compatible with the Type 2 connector. For DC charging, the Combo 2 inlet is used, which is compatible with the Combo 2 connector. The mutual compatibility between CCS connectors and inlets is illustrated in the figure (Figure 1.16). The figure shows the areas of use and also describes the function of the connector types. The difference in functionality when comparing CCS for the EU and North America is that the EU AC connector and inlet can also handle 3-phase AC, thus offering a higher charging rate for the electric vehicle battery (when using 3-phase AC). [49]

It is also important to note that for CCS there are two control mechanisms for AC charging. The first is based on basic signalling (BS) using PWM and the second is based on high level communication using PLC. For AC charging, CCS supports one or both of these control mechanisms during the charging process. For DC charging (within CCS) only high level communication based on PLC is possible. For DC, high level communication is required to allow control of the external DC charger (off-board charger).

In general, for a complete picture of the possible connector types, below i give a short description and a diagram of each connector type mentioned according to [39]. I will start with the interfaces used for AC charging and then move on to DC charging.

AC charging interfaces

Type 1 AC connector is defined in IEC 62196-2. This type is based on the SAE-1772 standard and is designed for single phase alternating current from 6 to 32 A. Due to the use of single-phase alternating current, it allows charging up to 7.4 kW. In contrast, the AC Type 2 connector, also defined in the IEC 62196-2 standard, is designed for three-phase AC use up to 63 A. The second type, due to its design (three-phase AC use), allows charging power up to 44 kW. The Type 2 connector is also designed to support single-phase AC charging. For single-phase AC charging with the Type 2 connector, pins L2 (AC phase 2) and L3 (AC phase 3) are not used. This makes it possible to adapt between the two mentioned connectors. An illustration of the plugs is shown in the images (Figure 1.17a and Figure 1.17a), which also show the layout of the contact pins.

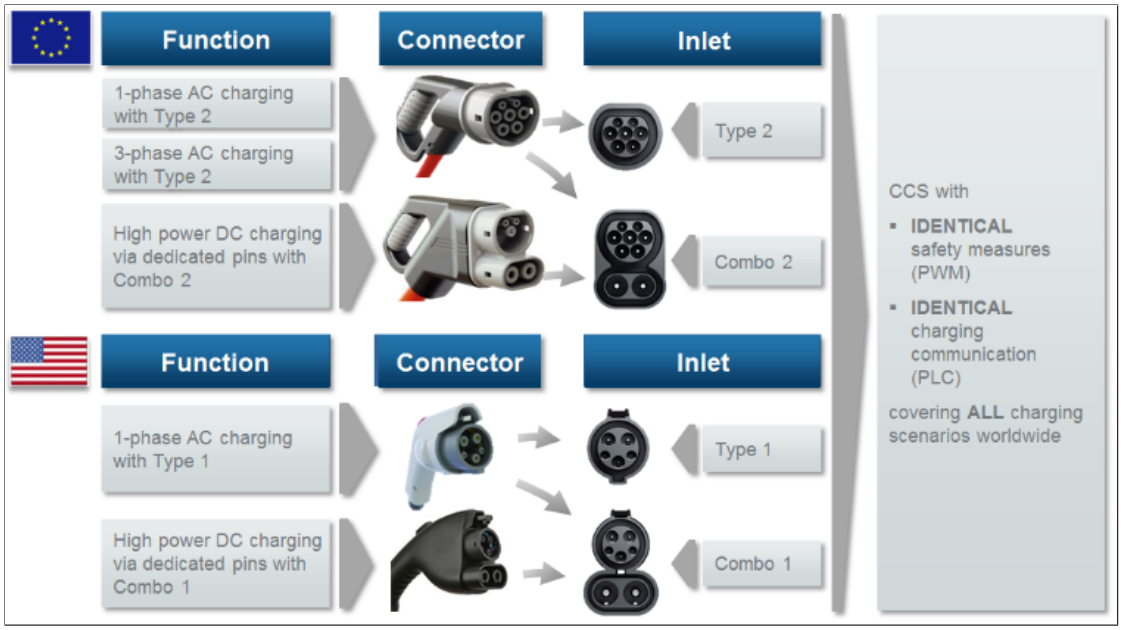
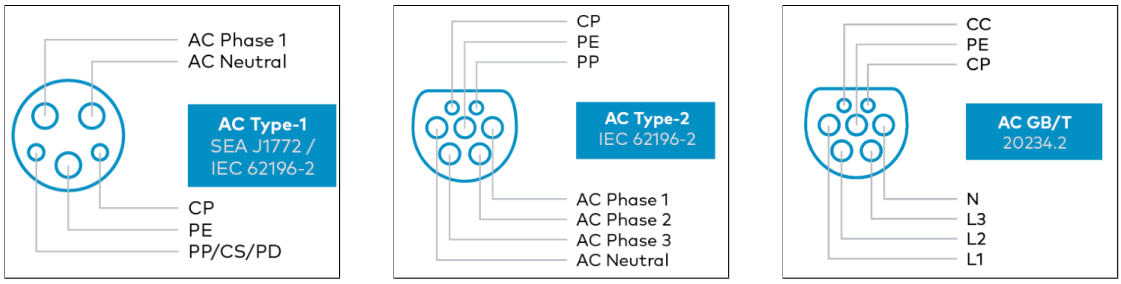


Figure 1.16 Connectors and inlets for CCS [50]



(a) AC Type-1 **(b)** AC Type-2 **(c)** AC GB/T

Figure 1.17 AC – EV side cable connector faces

[39]

The Type 1 connector is used in the North American market and in Japan, while the Type 2 connector is used for AC charging in the European Union. [47]

The GB/T AC connector type is defined in the GB/T 20234.2 standard and allows both types of AC charging, i.e. single phase and three phase. GB standards are national standards in China and the suffix "/T" means that these are recommended standards but not mandatory. The charging cables for this type of connector use identical male connectors as Type 2. [39, 47]

Tesla uses a unified connector for both AC and DC charging, called the Tesla Supercharger connector type [39]. This connector and its contact pins are shown in the image (Figure 1.20). According to [47], from 2018 onwards, Tesla vehicles must be equipped with a CCS Type 2 (i.e. European CCS) charging inlet to support the AC Type 2 connector.

DC charging interfaces

The CCS Type-1 (also known as Combo 1) and CCS Type-2 (also known as Combo 2) connectors are an extension of the AC Type-1 and Type-2 connectors for the DC charging process. CCS Type-2 is defined in IEC 62196-3 and CCS Type-1 is defined in IEC 62196-2 (as AC Type-1).

Thus, both types have only two extra large power contacts, which are used to transmit DC positive and DC negative. During the charging process, the pins for AC transfer are not used, but the other pins (CP, PP and PE) are used as in AC charging. When comparing the pictures for AC Type 1 (Figure 1.17a) and CCS Type 1 (Figure 1.18b) and similarly for AC Type 2 (Figure 1.17b) AND CCS Type 2 (Figure 1.18b), the aforementioned compatibility for AC and DC connectors can be seen, i.e. one vehicle inlet can be used for AC and DC charging. As a result, CCS DC connectors only add positive and negative contacts to allow DC charging. In this case, the CP pin is also used to transmit high level communication (via PLC). In the following section, I will discuss the Combined Charging System in more detail, as it is an important part of the charging process in the EU.

The other type of connector is aimed more at heavy-duty trucks, as it allows large batteries to be charged with high power, up to 3.75 MW. This connector is used for the Megawatt Charging System (MCS), shown in Figure 1. The MCS connector includes two additional communication lines that are separate from the CP and PP. The MCS and its communication are based on ISO 15118-20, but use a different connector. Furthermore, it is important to note that PWM communication is not used to precede high level communication as defined in IEC 61851 (PWM must precede HLC). It is also crucial to ensure proper cooling during the charging process when using MCS. [39]

Finally, I will discuss the DC charging interface standards used for Asian countries, namely CHAdeMO, DC GB/T and ChaoJi. Firstly, DC GB/T, as I mentioned, is a protocol used exclusively in China. Unlike the CCS type, the DC connector is not compatible with the AC inlet because the DC connector has different pins compared to the AC connector. The CP is used for communication within the AC charger, but for DC the CAN-H and CAN-L pins are used. The differences in the pins used can be seen in the pictures of the standard GB/T connectors (Figure 1.17c and Figure 1.19b). As a result of this inconsistency, the electric vehicle must be equipped with two inputs, one for AC charging and one for DC charging. The protocol used for communication during the DC charging process is defined by the GB/T 27930 standard, and the CAN protocol is used for communication. [39]

Another is the CHAdeMO standard connector, which, like the DC GB/T connector, uses CAN for communication during the charging process. This type of connector is mainly used in Japan and is also used by Japanese manufacturers in other markets (outside Japan) [39]. This type of connector is shown in the picture (Figure 1.19a). The importance of this standard and connector type is declining as Japanese manufacturers increasingly add CCS inlets to their EVs destined for the EU market. [47]

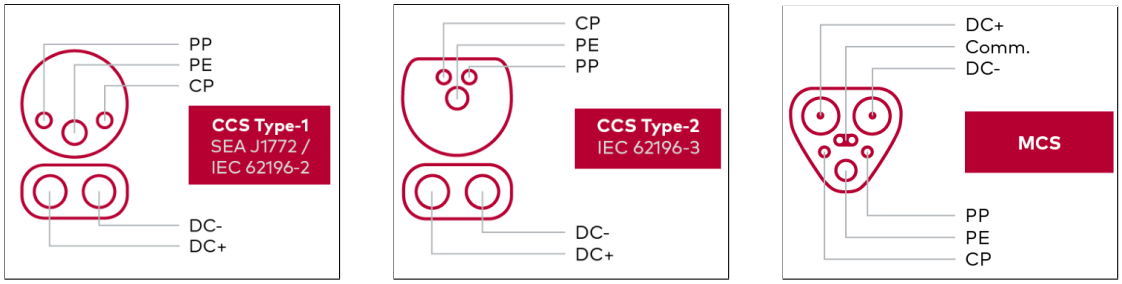
ChaoJi is a unified charging system for GB and CHAdeMO, providing up to 900kW of charging power. The pin layout of this type of connector is shown in the image (Figure 1.19c).

Tesla for DC has developed a proprietary communication protocol for its Tesla Superchargers and associated custom charging connectors. In the European Union, Tesla for DC is required to use CCS Type 2 inlets, as are other EU manufacturers. This regulation is part of EU Directive 2014/94. [39]

Combined Charging System

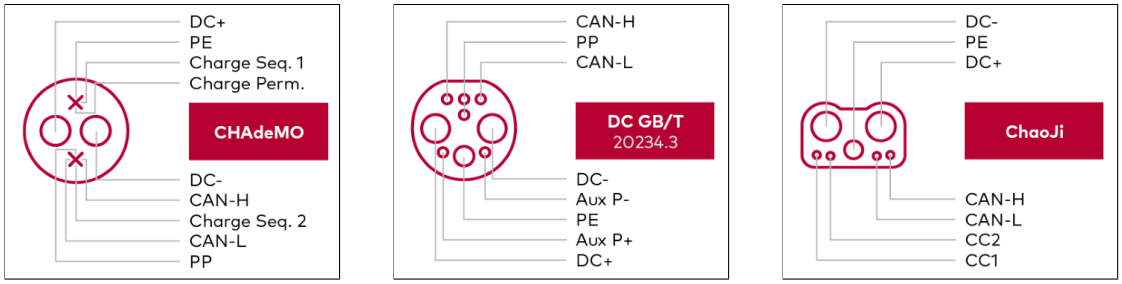
CCS is generally a universal system that attempts to combine individual AC and DC charging solutions into a single system. The result is a single vehicle inlet for 1ph AC, 3ph AC, and DC charging. This system not only includes plugs and sockets, but also defines which standards are used for communication during charging, and more. In short, this is a system that tries to globalise e-mobility and everything that goes with it, such as defining inputs, connectors, communication, safety requirements, etc. [51]

I describe the Combined Charging System in this section because it defines, for example, the connectors used for charging. Furthermore, this system is used within the European Union and is therefore closely related to the topic of this thesis. For these reasons, it is desirable to note



(a) DC CCS Type-1 **(b)** DC CCS Type-2 **(c)** Megawatt Charging System

Figure 1.18 DC – EV side cable connector faces



(a) CHAdeMO **(b)** DC GB/T **(c)** ChaoJi

Figure 1.19 DC – EV side cable connector faces 2

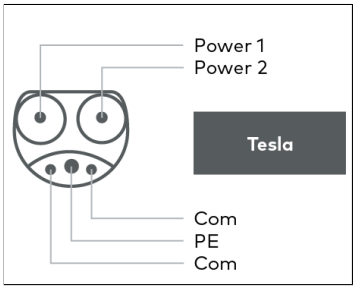


Figure 1.20 Tesla AC and DC connector

here the most important things that the CCS includes.

The CCS defines the use of charging connectors and charging inputs for the EU and US markets as described above. The functions, names of connector and inlet types, and their compatibility are illustrated in the figure (Figure 1.16). In addition, CCS includes a communication interface based on the international standard ISO/IEC 15118 and the German standard DIN SPEC 70121. [49]

The main advantage of the unified approach in the form of CCS is a combined input for EV charging (either AC or DC charging process is used), as the connectors for AC and DC are compatible with the Combo inlet (works for both Type 1 and 2). For the EU, the naming convention for the inlet is Type 2 AC inlet (abbreviated as Type 2 inlet), Combo 2 AC and DC inlet (abbreviated as Combo 2 inlet), and the naming convention for the connectors is Type 2 AC connector (abbreviated as Type 2 connector), Combo 2 DC connector (abbreviated as Combo 2 connector). Another advantage is the single communication interface and associated uniform charge control logic. [49]

According to [49] the charge control methods can be divided for AC and DC. In the case of AC charging, either basic signalling based on PWM or high level communication using PLC is possible. CCS supports one or both of these methods. For DC charging, CCS only allows high level communication based on PLC because of the need to control and communicate with the off-board charger. This means that communication for DC charging cannot be handled using only basic signaling.

The CCS uses four charging modes that differentiate and define the various conditions and communication protocols during charging. Three of these modes are for AC charging and therefore an OBC is required in the vehicle for the first three modes. These modes are defined by the IEC 61851-1 standard, and they are described in detail in Charging Standards.

Due to the use of CCS connectors in electric vehicles on the EU market, the most important pins of the charging connector or inlet will be described here. The pins that carry the current have self-describing labels in the pictures (Figure 1.17b and Figure 1.18b). The first pin is the Control Pilot (CP), which is used for communication between the EV and the EVSE, i.e. it ensures the transmission of control signals. For low level communication, basic signalling using PWM is used; this communication also serves as a safety function. For PWM, a 1 kHz square wave is used between ± 12 V. The EV can manipulate the amplitude by means of a load resistor placed in the EV, which allows controlling the charging of the [52]. This manipulation is done by switching the load impedances between pins: CP and PE (CP circuit). For AC charging only PWM communication can be used; for DC high level communication is required. I describe the details about PWM in Charging Standards, because the meaning of volt values and the duty cycle of PWM is defined in the IEC 61851-1 standard. [39, 52]

The Proximity Pilot (PP) pin is crucial in preventing the EV from moving while connected to the EVSE. The EV can only be driven away once the charging plug is disconnected. For the Type 1 plug, PP is also used to manually unlock the plug. Furthermore, the Type 2 connector utilizes this pin to detect the current load of the charging cable. In previous versions of the standards, PP was referred to as Connection Signal (CS) or Proximity Detection (PD). [39]

The Protective Earth (PE) conductor is grounded on the EVSE side to ensure that the electric vehicle is also grounded during charging. It serves as a reference pin for CP and PP and also helps prevent electric shock. [39]

Another important aspect of CCS is the distinction between authorisation modes for charging [49]. External Identification Means (EIM) and Plug and Charge (PnC). The basic difference between these methods is that EIM requires user interaction or other identification operations for authorisation and payment, whereas PnC does not require or even need this. The EIM method offers several identification and authorisation tools, including credit cards, RFID cards, and mobile applications. In contrast, PnC provides a more secure and convenient method of identification and authorisation by connecting the EV to a charging station. PnC uses asymmetric cryptography, public key infrastructure (PKI), and stores certificates in both the EV and

EVSE, as defined in the ISO 15118 standard. [49, 39]

1.3.4 Charging Standards

In this section, I describe the standards rather superficially to give an overall picture of the standards used. Due to the instructions in the assignment for this thesis, I mainly focus on the standards used within the Combined Charging System [49]. The CCS specification contains information about connectors, charging modes, charging authorisation types, communication technologies, standards and more (see [49] for more information). The standards listed below are selected on the basis of the descriptions in [53] and [49]. I cover standards related to conductive charging (connectors, general requirements), high level protocols, WPT, charging station communication with Charging Station Management System and backend. Finally, I describe standards originating from Asia.

The standards listed below can also be logically divided on the basis of the type of standard, i.e. international and national. The international standards are IEC 62196, IEC 61851, ISO 15118, IEC 61980 and IEC 63110. The national standards are the following: SAE J1772, SAE J2847/2, SAE J2954, GB/T 18487.1, GB/T 20234, and GB/T 27930 and CHAdeMO. Alternatively, they can be logically divided based on the type of use as follows:

- Connectors, Inlet, Plugs: IEC 62196, SAE J1772, GB/T 20234, CHAdeMO;
- Onboard Charger, EVSE: IEC 61851, GB/T 18487, GB/T 27930;
- Wireless Power Transfer (WPT) Systems: IEC 61980, SAE J2954;
- Communication EV To EVSE: ISO 15118, DIN SPEC 70121, GB/T 27930, SAE J2847/2;
- Communication EVSE To CSMS: IEC 63110.

IEC 62196

The full name for this standard is **Plugs, Socket Outlets, Vehicle Connectors and Vehicle Inlets – Conductive Charging of EVs**. As its name implies, this standard defines the requirements and tests for plugs, socket outlets, vehicle connectors, and vehicle inlets. [53]

Type 1 and Type 2 connectors are defined in this standard, specifically in IEC 62196-2. The combo 1 and combo 2 connectors are also defined by this standard, specifically IEC 62196-3. [49]

SAE J1772

The title of this standard is **Electric Vehicle and Plug in Hybrid Electric Vehicle Conductive Charge Coupler**. SAE J1772 is the standard for conductive charging in North America and includes general physical, electrical, and functional recommendations for charging EVs and PHEVs. In addition, this standard defines functional and operational requirements for the vehicle inlet and its associated Type 1 connector. It also includes requirements for PWM signaling. It is the predecessor of IEC 61851 and ISO 15118. This standard uses a division into 3 levels (3 for both AC and DC). [53]

IEC 61851

This is another international standard called **Electric Vehicle Conductive Charging System**. IEC 61851 describes the characteristics and operating conditions for EVSE, specifies the connection between the electric vehicle and EVSE and defines the electrical safety requirements for EVSE. According to [53], the most important parts of this standard are IEC 61851-1, IEC 61851-23, and IEC 61851-24. The first part, IEC 61851-1, contains general requirements. IEC

Mode	Characteristics
`Mode 1`	This mode of AC charging uses a low current up to 16 A via 1ph / 3ph normal main sockets commonly found in households. The cable used for this mode does not contain a protection device. It is important to note that no communication takes place and an RCD installed in the home is required.
`Mode 2`	AC charging, where the EV is again connected via normal mains via 1ph/3ph sockets. A higher maximum possible value of the supplied current (32 A) than in Mode 1. The protection device must be integrated in the charging cable (so-called in the cable control box, ICCB). Signalling between the ICCB and the EV is possible via CP using PWM.
`Mode 3`	AC charging using dedicated AC charging stations (Type 1 and Type 2 connectors). It can deliver up to 63A of current and offers improvements over the previous two modes. The station is equipped with a safety device, eliminating the need for an ICCB located in the cable. PWM is mandatory for communication, with optional high-level communication (HLC). Compared to modes 1 and 2, HLC allows for power feedback, i.e. charging power can be controlled via HLC. Control and lock of the plugs are also possible.
`Mode 4`	DC charging utilises dedicated DC charging stations with Combo 1 and Combo 2 connectors. The charging system can adapt to the battery system by providing different current and voltage values. An HLC is required to control and manage the charging process, enabling the aforementioned adaptation. This mode is the only one that uses an off-board charger, providing a higher charging rate. Both PWM and HLC communications are mandatory.

Table 1.3 Charging modes as per IEC 61851-1

61851-23 covers DC EV charging stations, while IEC 61851-24 covers digital communication between a DC charging station and an EV to control DC charging. [53]

IEC 61851-1 includes definitions of various charging modes for conductive charging as part of its general requirements. These modes are classified based on the type of charging and needed equipment. The standard also specifies the necessary protection and communication protocols. In table (Table 1.3) I list the individual communication modes and their brief description based on [53] and [49].

IEC 61851-1 defines, in addition to the listed charge modes, the meaning of the duty cycle for PWM and also the meaning of the individual signal voltage values (Table 1.4). The meaning of the voltage values is given in the table (Table 1.4) and the meaning of the duty cycle values is described in the table (Table 1.5) according to [38]. PWM is a tool for low level communication between EV and EVSE. The PWM signal is applied to the CP circuit, i.e. the circuit between the CP and PE. The voltage of the signal is controlled by the EV, while the duty cycle is controlled by the EVSE. [54, 51]

Voltage	State	Description
+12 V	`State A`	No coupler engagement, no EV is connected to the EVSE.
+9 V (1kHz PWM)	`State B`	Coupler engagement detected (EV is connected to the EVSE), but EV not ready for charging. EVSE does not supply energy.
+6 V (1kHz PWM)	`State C`	EV is connected and ready for charging. EVSE supplies energy.
+3 V (1kHz PWM)	`State D`	EV is connected and ready for charging. EVSE supplies energy. Ventilation is required.
0 V	`State E`	Short of CP to PE on the EVSE, no power supply.
−12 V	`State F`	Charging station is not available.

Table 1.4 States of low level communication as per IEC 61851-1 [38]

Duty cycle	Description
Duty cycle > 97%	Charging is not allowed.
96% < duty cycle 97%	Maximum current consumption for AC charging is 80 A.
85% < duty cycle 96%	Available current = $(dutycycle - 64) * 2A$.
10% duty cycle 85%	Available current = $dutycycle * 0.6A$.
8% duty cycle < 10%	Maximum current consumption for AC charging is 6 A.
7% < duty cycle < 8%	Charging is not allowed.
3% duty cycle 7%	Force use of high-level communication protocol (ISO 15118 or DIN 70121). If pilot function wire is used for digital communication, then the duty cycle 5 % shall be used. [55]
Duty cycle < 3%	Charging is not allowed.

Table 1.5 PWM duty cycle as per IEC 61851-1 [38]

DIN SPEC 70121

DIN SPEC 70121, fully entitled **Digital Communication Between a DC EVSE and an EV for Control of DC Charging in the CCS**, defines the high-level communication between the electric vehicle and the charging station during the DC charging process. This standard is the German predecessor of ISO 15118-2. However, unlike DIN SPEC 70121, the successor provides DC PnC and also defines AC HLC, as DIN SPEC 70121 only defines DC charging and only EIM is allowed as an authorisation method. Conformance tests are included in DIN SPEC 70122. [53]

SAE J2847/2

Communication Between Plug-In Vehicles and Off-Board DC Chargers is the heading of this standard and specifies the requirements for communication between plug-in vehicles and a DC charging station (i.e. off-board charging). It is a similar standard to DIN SPEC 70121. [53]

ISO 15118

This standard is known as the **Road Vehicles - Vehicle To Grid Communication Interface**. It defines processes such as AC and DC charging, charging with pantographs, bidirectional power transfer, and wireless charging. The standard includes the definition of high-level communication for the charging process (AC and DC), which uses Power Line Communication (PLC). The protocol is described in detail in a separate section of this thesis. [38]

Within CCS, this protocol (in addition to DIN SPEC 70121) is used for HLC [51].

IEC 61980

IEC 61980 is an international standard relating to wireless power transfer (WPT) systems. The standard focusses on the specification of equipment for wireless transfer between an electric vehicle and a power supply network, as the name **Electric Vehicle Wireless Power Transfer (WPT) Systems** suggests. Additionally, the standard covers the storage of electrical energy in a rechargeable energy storage system (RESS) or other on-board devices. [53]

SAE J2954

Another standard focussing on WPT is called **Wireless Power Transfer for Light-Duty Plug-in/Electric Vehicles and Alignment Methodology**. This standard focusses on home (private) charging and public WPT for light-duty plug-in electric vehicles. SAE J2954 also specifies safety, performance, and interoperability requirements. According to [53], this standard currently provides guidelines for static (vehicle not moving during charging) WPT and unidirectional charging. It also recommends methods for assessing electromagnetic emissions. The use of bidirectional charging and dynamic WPT may be included in the future. [53]

OCPP

In this case, it is not a standard but a protocol created by the Open Charge Alliance (OCA). The protocol was developed to make EV networks open and accessible. The Open Charge Point Protocol (OCPP) has become a widely used protocol for communication between EVSEs and the Charging Station Management System (CSMS) and is now the de facto standard. The latest version, OCPP 2.0.1, includes further enhancements, such as security and configurability improvements. Currently, the most widely used version is OCPP 1.6J, but unfortunately, these versions are not mutually compatible. In the future, the IEC 63110 standard described below may replace OCPP. [53]

OCPI

Another protocol, known as the Open Charge Point Interface, focusses on communication between CSMS and the clearing backend. The use of this protocol enables the improvement of mobility services, such as providing tariff information with flexible prices or monitoring the charging process. These enhancements can be offered to customers through an application. [53]

IEC 63110

The standard is named **Protocol for management of electric vehicles charging and discharging infrastructures**. Its purpose is to define a communication protocol for exchanging messages between the charging station and the backend interface. In the future, it may enhance or possibly replace OCCP. [53]

GB/T 18487.1

The GB/T series of standards are issued by the Standardization Administration of China (SAC) and are the Chinese National Standards [53]. The full name for GB/T 18487.1 is **Conducting Charging Systems for Electric Vehicles – Part 1: General Requirements**. It outlines the general requirements for conductive charging, including communication requirements, protection against electric shock, connection between EV and energy transfer equipment, and special requirements for EV plug and socket. [56]

GB/T 20234

The name of this standard is **Connection set for charging - Conductive charging of electric vehicles** and the two most important parts are the second (GB/T 20234.2) and the third (GB/T 20234.3). The second part is called “Part 2:AC charging coupler” and as the name suggests, this part defines the requirements for the AC charging coupler (connector). This AC connector is similar to AC Type 2 except that it uses different control signaling. The third part (GB/T 20234.3) also covers the connector but for DC charging. Both standards are designed for conductive charging. [53]

GB/T 27930

The original name of the first two versions is **Communication protocols between off-board conductive charger and battery management system for electric vehicle**, the last released version is called “Digital communication protocols between off-board conductive charger and electric vehicle”. This standard specifies the requirements and defines the CAN-based communication between the off-board conductive charger and the BMS, specifically defining the physical, data and application layers for this communication. [53]

CHAdeMO

In addition to the connector, CHAdeMO also defines the communication between the EV and the charging station for the DC charging process. Similar to Chinese standards, CAN is used for communication. It supports a charging power of up to 400 kW, which makes it possible to charge large vehicles, such as buses or trucks. This protocol also includes compatibility with the “Plug and Charge” functionality. [53]

1.3.5 ISO 15118

The standard's official title is "Road vehicles – Vehicle to grid communication interface". This standard defines the use of protocols for high level communication between EV and EVSE. The communication controller in the EV is called EVCC and the controller in the charging station is called SECC. Furthermore, a smart charging mechanism is also included in the standard. V2G (Vehicle To Grid) enables, in addition to the mentioned processes (BPT, ACDP, WPT, ...) in Charging Standards, smart charging and the associated grid stabilisation. To stabilise the grid, this standard offers two modes: scheduled and dynamic. The latter mode allows the charging station to control the charging and discharging of the EV. The use of BPT and dynamic mode is used to achieve a better balance in the grid and also to control the load and distribute it appropriately. This can also be used at home in the case of self-sufficient houses that can produce electricity, as the EV battery can act as a buffer for excess energy. [57]

WPT technology combined with PnC is very user-friendly as it does not require any user interaction. With WPT it was also necessary to provide wireless communication, this standard uses IEEE 802.11n as the physical layer for communication (ISO 15118-8). Also because wireless communication is easier to intercept than conductive charging, ISO 15118-20 requires the use of TLS in all cases. Inductive charging is associated with WPT, which is based on IEC 61980. This standard (IEC 61980) offers improvements such as paring, fine positioning and alignment control. On the other hand, ACDP is a conductive charging method, but since it uses an automatic mechanism to connect the pantograph, it is as convenient as WPT. [57]

According to the instructions in the thesis assignment, the target is a European electric vehicle, so I am focussing on the ISO 15118 standard, which according to [51] is used for HLC. DIN SPEC 70121 can also be used for HLC, but ISO 15118 is based on this standard and is its successor, so I am not considering DIN SPEC 70121. Another reason I am focussing on ISO 15118 is that it supports both AC and DC charging, specifically charging modes 3 and 4 (charging modes according to IEC 61851, Table 1.3). It also supports TLS, whereas DIN SPEC does not.

There are several parts to this standard, as shown in the figure Figure 1.21. The figure shows which part of the standard belongs to a given layer in the ISO/OSI model. Each part of ISO 15118 is therefore associated with one or more layers within the ISO/OSI model. This standard also includes another part (not shown in the figure), ISO 15118-10, which is still under development at the time of writing and focuses on single-pair Ethernet as an alternative to PLC [58].

For conductive charging communication, ISO 15118-3 is intended for OSI layers 1 and 2. The remaining layers are covered in the second part (ISO 15118-2) and in the second generation of this part (i.e. ISO 15118-20).

Based on the agreement with the thesis supervisor, we decided that the tool should target layers 3 to 7 of the ISO/OSI model. Accordingly, we chose ISO 15118-2, which defines the requirements for communication within layers 3-7 of the ISO/OSI model (illustrated in Figure 1.21). Although there is a second generation (ISO 15118-20), ISO 15118-2 is still valid. In addition, ISO 15118-20 only came into effect in 12/2022 and contains extensions that are often not needed by vendors (such as WPT). The first real example that implements only ISO 15118-2 is the OpenECU M560 or M580 [60]. Another example of ISO 15118-2 support is the Typhoon HIL toolchain [61]. In addition, the original version of the second part of the standard serves as the basis for ISO 15118-20. Furthermore, according to [62], it is impossible to say when ISO 15118-20 will be adopted and implemented by EV manufacturers. Furthermore, the article estimates that the first manufacturers will release vehicles in 2024 [62]. The important fact is "ISO 15118 is in roll out in a lot of products from what we see in our CharIN Testivals. The new edition 15118-20 will be tested and conformance tests are still in development", this information is from Ricardo Michaelis (CharIN) (obtained via email correspondence). For these reasons, I have chosen ISO 15118-2 instead of ISO 15118-20.

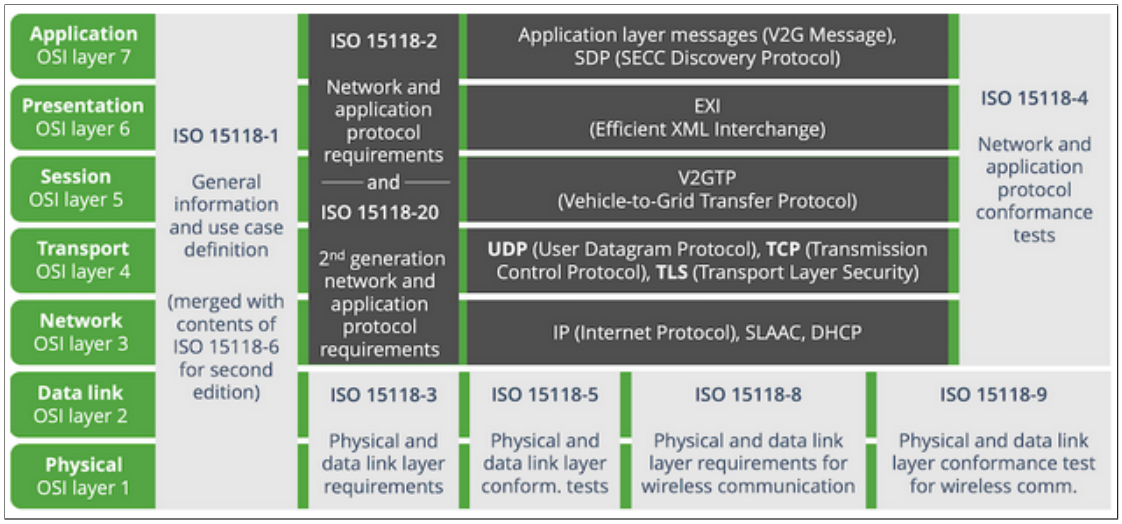


Figure 1.21 ISO/OSI layers according to ISO 15118 [59]

The most important parts of the standard to adequately describe communication and requirements are ISO 15118-3 and ISO 15118-2. These two parts cover the entire ISO/OSI model. I will start by describing the standard from the lower layers, i.e. the physical and data link layers.

The third part of ISO 15118 focuses on HomePlug GreenPHY (HPGP) on the CP link (CP, PE dedicated connection) and it is HPGH that is used for the physical and data link layer [51]. ISO 15118-3 references the HPGP specification for the physical layer that utilises powerline as the communication medium [63]. HPGH defines in its specification the modulation of digital data to a signal (and back) transmitted over a communication medium. Furthermore, HPGP introduces an enhancement known as SLAC (Signal Level Attenuation Characterization) in comparison to HomePlug AV. Additionally, ISO 15118-3 provides further specification for the use of the SLAC mechanism, which is used to establish an AV Logical Network (AVLN) between physically connected EVCCs and SECCs. The AVLN serves to transfer information for higher layer protocols. [63, 64]

The PLC module (PLC device/bridge) handles the first two layers, and the connection on these layers is always based on the MAC addresses of the connected devices (with local bridge/devices) [2]. As mentioned above, my implementation focusses on layers from the network layer to the application layer, so I do not mention detailed requirements and properties for lower layers. I will only focus on the SLAC mechanism, which is essential to understand the entire communication during the charging process. SLAC is described in more detail in the section titled Charging Communication. Successful communication at the data layer is a prerequisite for ISO 15118-2. Therefore, communication at higher layers of the ISO/OSI model depends on it [34].

ISO 15118-2 assumes that the data layer allows the transmission of IP packets. This part of ISO 15118 builds on this assumption and specifies the protocols that are used for communication: IPv6 (Internet Protocol Version 6), UDP (User Datagram Protocol), TCP (Transmission Control Protocol) and TLS (Transport Layer Security). These common protocols are complemented by a protocol specific to this standard, which is V2GTP (Vehicle to Grid Transfer Protocol). For the presentation layer, the EXI (Efficient XML Interchange) format is used, which encodes the original message in XML format into binary form using specific schemas (defined in [34]). The application layer uses SDP (SECC Discovery Protocol) and application layer messages (V2G Message), V2GTP transfers V2GTP PDUs between two V2G entities. The payload according to [34] can be EXI encoded V2G Message, SDP request or SDP response. V2GTP is the standard protocol for communication between EVCC and SECC and transmits the required data before and during the charging process (e.g.: ensuring compatibility and charging conditions, status

detection during charging) using V2G messages. IP-based protocols are used for communication between EVSE and EV, so communication controllers use TCP/IP stack for communication. [34]

1.4 Charging Communication

According to [27], the charging systems can be divided by region as follows: CCS (EU and North America), GB/T (China), CHAdeMO (Japan). The first one, CCS, is the most important, not only because it is used in the EU, but also because it tries to globalise and standardise the charging process (striving for compatibility for all EVs and EVSEs). In addition, [27] specifies the standards and technology used to communicate during the charging process.

I focus only on the communication between the electric vehicle and the EVSE, I do not consider other entities in the charging infrastructure. Therefore, the communication between the charging station and the Charging Station Management System or between the CSMS and the backend (e.g. Dispatcher, Fleet Management, Distributed Systems Operator) is not described here.

Communication for AC charging within the CCS is provided by PWM (basic signalling) or PLC (high level communication), while for DC charging, communication is only via PLC. It is important to note that PWM is used in both cases, but only in the case of DC charging must there be additional communication via PLC. PLC is an optional extra for AC charging, but only for charging mode 3. Standards that are essential for CCS in the context of communication: IEC 62196, IEC 61851, ISO/IEC 15118, DIN SPEC 70121, SAE J2847/2. For GB/T, communication for AC charging is provided by PWM and for DC via CAN. CHAdeMO defines only DC charging and communication via CAN. [27]

From the entire previous theoretical section, it is evident that the most important standards for communication in the EU are IEC 61851 and ISO/IEC 15118. It should be noted that IEC 62196 mainly defines the interface and not the communication itself, which is why it has not been included. I don't consider SAE standards because they are relevant to North America. Similarly, Asian standards are also excluded, as the main focus of this work is on European vehicles. As previously mentioned, Tesla must use the vehicle inlet Combo 2 (compatible with connectors: AC Type 2, DC Combo 2) within the EU.

According to [2], communication can be classified into two categories: basic signalling and high-level communication (HLC). Some sources, such as [38], refer to basic signalling as low-level communication. The communication between an electric vehicle (EV) and an electric vehicle supply equipment (EVSE) can be divided into low-level and high-level communication. The low-level communication is defined in IEC 61851, while the high-level communication is defined in ISO 15118. It should be noted that DIN SPEC 70121 is not considered here as ISO 15118 is its successor [53]. First, I focus on the specifics of low level communication and information where high level communication is required. The first section describes low-level communication, while the second section describes high-level communication based on the previous ISO 15118 description (in section called ISO 15118).

1.4.1 Low Level Communication

Low-level communication or basic signalling refers to physical signalling as defined by the pilot function. The pilot function ensures the necessary safety and data transfer conditions are provided by the electrical means [2]. The pilot function follows the guidelines set out in the IEC 61851-1 standard ([55]). The control pilot circuit (CP and PE) and PWM modulation are used to transmit these signals. [2, 55]

In contrast, the specification for HLC is contained in ISO 15518. If the charging process only uses basic signalling, then the charging process only follows the specifications of IEC 61851-1

([55]) and is called BC (Basic Charging according to [34]).

This low level communication offers basic control of the charging process between the EV and EVSE, as well as status negotiation using the voltage of the transmitted signal [2]. This state is called CP state (or Control Pilot Status) or Vehicle state, because the voltage is regulated by the electric vehicle. This voltage of +12 V is generated by the EVSE. The voltage is adjusted by means of resistors added to the circuit (CP, PE circuit) by the electric vehicle (physically connected to the EVSE) [55]. Depending on the given state resulting from the measured voltage value (the relationship between voltage and state is shown in Table 1.4), the EVSE generates a 1 kHz PWM signal. The EVSE adjusts the duty cycle of the PWM signal to inform the EV of available current, to force HLC, or to inform the EV that charging is not allowed. The duty cycle values and their meaning are explained in Table 1.5. Therefore, the check and measurement of the duty cycle value is done on the EV side. [55]

Therefore, the duty cycle is controlled by the charging station and measured in the vehicle, while the signal voltage is controlled by the electric vehicle and measured in the EVSE (this is demonstrated in "Figure A.2 – Simplified control pilot circuit" in [55]).

According to [55], charging is divided into four charging modes (shown in Table 1.3), the first three modes are for AC charging and the last one is for DC charging. Except for charging mode 1 (no communication), all the others require low level communication. For charging modes 2 and 3, [55] basic signalling is used to indicate the current available for charging.

For mode 4 (DC charging) a duty cycle of transmitted signal is used to indicate force HLC or information that charging is not allowed. Specifically, if the CP wire is to be used for digital communication, then a duty cycle of 5 % should be used [2]. The DC charging process requires high level communication to enable off-board charger control. For AC charging in mode 3, HLC is an optional feature and can be used, for example, to communicate available current information to the on-board charger. Even for AC mode a PWM signal with 5 % duty cycle forces HLC. [55, 2]

Therefore, it implies that the 5 % duty cycle set by EVSE is required for the initiation of the HLC and this HLC communication can be used for mode 3 [2] in addition to charging mode 4.

PWM – Summary

Basic signalling with PWM is done via the control pilot (CP) circuit (i.e. connection of PE and CP). The charging station uses the PWM duty cycle setting to inform the EV of: maximum available charge current, if charging is possible, and use of HLC. The state of the vehicle (or CP state) is determined by the voltage of this signal. EVSE detects whether the EV is connected and ready for charging based on the voltage value of the signal. The EV controls the signal amplitude of the PWM signal using resistors, while the EVSE controls the duty cycle of the PWM signal to communicate with the EV. The definitions of states and the meaning of duty cycle values are given in tables (Table 1.4 and Table 1.5), more detailed information can be found in [55].

According to the standard mentioned, it is a 1 kHz signal and the voltage value generated in EVSE for state A is 12 V and for state B it can be a steady 12 V DC voltage or a square wave voltage of ±12 V. EVSE uses a duty cycle of 5 % to force HLC for charging modes 3 and 4.

It is necessary to use basic signalling (i.e. low level communication) for charging modes 2, 3, and 4. For modes 2 and 3, charging communication can only be handled at the low-level communication level. In the case of charging mode 4 basic signalling serves as a prerequisite for HLC. For charging mode 3, basic signalling can also be used to enforce high-level communication only, and other information can be exchanged through it. In this thesis, I focus on the HLC part of communication and therefore only modes 3 and 4 are relevant. Mode 3 is used for AC charging with an on-board charger, while mode 4 is used for DC charging with an off-board charger.

Within the CCS the same inlet is used for both types of conductive charging (within the EU Combo 2 inlet) and therefore the communication takes place through the same CP circuit, both for AC (on board charging) and DC charging process. The processing and control of basic

signalling via the CP circuit is described in more detail in [55].

Although low level communication is used for charging modes 2 and 3 (i.e. AC charging), there is not as much room for error in terms of security testing in the implementation and interpretation of the standard as for HLC (i.e. ISO 15118-2). In this regard, the HLC is of greater interest, so I focus on it. Successful enforcement of HLC using a 5 % duty cycle PWM is considered a prerequisite for this thesis and the tool developed. The advantage of this is that in addition to on board charging (AC charging) it is also possible to test (DC charging), specifically charging modes 3 and 4 where HLC is used.

1.4.2 High Level Communication

High level communication should provide other enhancements such as: identification, payment, load levelling, energy transfer control, charging parameters (e.g. voltage and current) and value-added services. A description of the compatibility and relationships between low level and high level communication are described in parts of ISO 15118 (specifically ISO 15118-2, ISO 15118-3, ISO 15118-8 and ISO 15118-20). [2]

As stated in the previous section (Low Level Communication), HLC is only available for charging modes 3 and 4. In charging mode 4, HLC is mandatory for communication between the vehicle and the off-board charger. One of the prerequisites for HLC is a PWM 5 % duty cycle sent from the EVSE to the EV, among other things.

Within the ISO/OSI model, we can divide high level communication into two parts containing different number of layers of this model. The division is based on the parts of the standard that describe the layers. The first part is included in ISO 15118-3, which describes the physical and data link layer. The second part is included in ISO 15118-2 (possibly ISO 15118-20), which covers the description and use of protocols within layers 3-7 of the ISO/OSI model.

For the transmission of high level communication, PLC technology is used through the CP circuit (CP, PE). The typical EV architecture for communication is the EVCC and the associated PLC module [2]. This PLC device is connected to the PLC module in the EVSE, which is connected to the SECC within the EVSE. The EVCC is the communication controller in the EV and the SECC is the communication controller in the EVSE, both controllers communicate with each other in accordance with ISO 15118-2 (or 15118-20) [2]. Communication between lower layer devices (i.e. PLC modules) is in compliance with 15118-3, in which data-to-signal modulation is used for data transfer using HomePlug Green PHY (HPGP) [63]. A local matching process takes place between lower layer controllers (see the section describing ISO/OSI Layer 2 for more details). The [63] standard to describe the physical layer and data link layer refers to the HPGP specification and further defines, for example, the interface between layer 2 and layer 3. This interface is called “DATA SAP” and serves to define the interface between layers 2 and 3 to guarantee the transmission of V2G messages (IPv6). Within [63] Data SAP is used as ETH SAP, which is defined in the HPGP specification. HPGP provides a convergence layer between the HPGP data layer and the higher layers using Ethernet II-class SAP. This means that it supports applications using Ethernet II-class packets. [63]

“The convergence layer adapts the generic HomePlug Green PHY MAC to an IEEE 802.3 Ethernet II-class interface through the ETH SAP. The control SAP provides a control interface to access HomePlug GreenPHY specific data and configuration for network management, including encryption key management, SLAC services, and link status information.” [63]

In simple terms, 15118-3 defines the use of HPGP to provide the physical and data link layers within the ISO/OSI model. At the same time, HPGP provides the conversion between the HomePlug Green PHY MAC and traditional Ethernet. The tool in development builds on this by using boards that have implemented HPGP, ensuring communication at the data link layer and allowing the tool to address the implementation of higher layers.

If data layer communication is successfully established using the PLC modem (including HGPG), then higher layer communication is possible. Establishing data link layer communication

requires the [34] standard, which deals with the definition and use of protocols at the higher layers of the ISO/OSI model (layers 3-7).

The aforementioned establishment of communication within the data layer is provided by a process called SLAC (Signal Level Attenuation Characterization), which has been added to HPGP (HomePlug AV does not include it). This process is described in more detail in the ISO/OSI Layer 2 description below.

The following section first describes the prerequisites, tasks and also the result for a given ISO/OSI layer in high level communication. Prerequisites are the necessary conditions on which a given layer is built. Tasks refer to what is to be conveyed, provided, and arranged within a given layer. The result is what the layer has arranged after the given tasks have been completed.

ISO/OSI 1. Layer

The first prerequisite is to connect the EV and EVSE with a charging cable, in particular to connect the PE and CP pins. Both sides (EV and EVSE) must have compatible Home Plug Green PHY modems. A PWM signal with a duty cycle of 5 % must pass through the CP circuit and the measured voltage must be +9% V, that is, CP state B. [51]

The task of this layer is to establish a physical connection on the PE-CP wires between the charging station and the vehicle. Once the conditions are satisfied and the connection is established, the result is that the PLC modules are ready to communicate with an established frequency band of 1.8 MHz to 30 MHz. This layer therefore ensures that the physical connection between the EVSE and the EV is activated to allow two-way data exchange. [51, 63]

The reference standard for this layer is [63].

ISO/OSI 2. Layer

This layer assumes a successfully completed Layer 1 process, i.e. it assumes that the physical connection is established correctly. The task of this layer is to configure the PLC modules to allow communication between physically connected PLC nodes (PLC devices). Another task, which is related to the previous one, is to distinguish which PLC module of the charging station is physically connected to the PLC module in the vehicle. [63, 51]

This distinction is made by a process called SLAC (Signal Level Attenuation Characterization), which, by measuring the attenuation between two Power Line Communication (PLC) modules, ensures that the communicating PLC modules are physically connected (i.e. the EV and EVSE are physically connected by a charging cable). SLAC is designed to prevent crosstalk that can occur when several vehicles are connected to charging stations in close proximity, as shown in the image (Figure 1.22). The PLC modules with the lowest measured attenuation are therefore considered to be physically connected. The EVSE can only respond to a SLAC request (from the EV) if the EV is connected and in state B. At the same time, the PLC module is in an unmatched state within the SLAC process, i.e. it is not yet linked to another EV PLC module. The result of the SLAC process is that the physically connected PLC modules set up a logical network (AVLN). SLAC itself is defined in the HPGP specification, and its use is specified further in ISO 15118-3. The SLAC sequence is illustrated in figure (Figure 1.23). [51]

The successful process results in the establishment of a logical network between the physically connected PLC module of the EVSE and the EV PLC module. This layer provides error-free transfer of data frames from one node to another over the physical layer. Once the data layer is established, the other layers (3-7) utilise the definitions and specifications of ISO-15118-2. [51]

ISO/OSI 3. Layer

The condition for the functioning of this layer is the completed establishment of layer 2. The task at this layer is to use / implement Internet Protocol Version 6 (IPv6), to make sure that

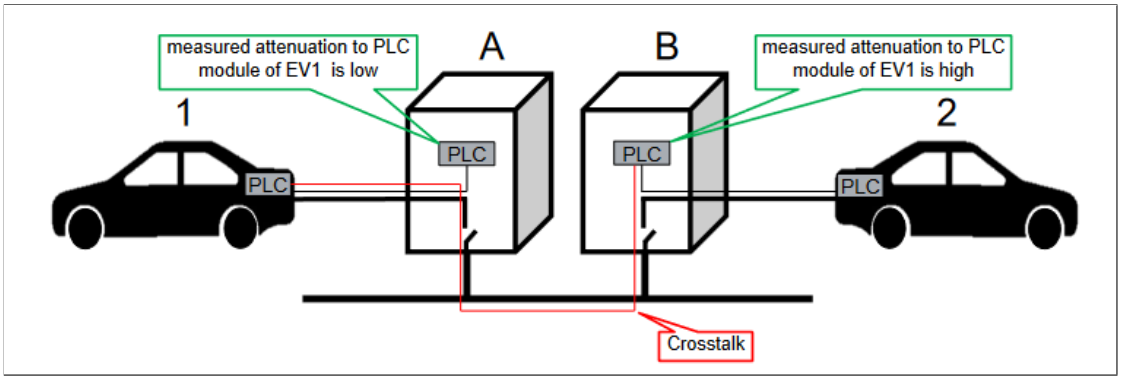


Figure 1.22 Crosstalk problem [51]

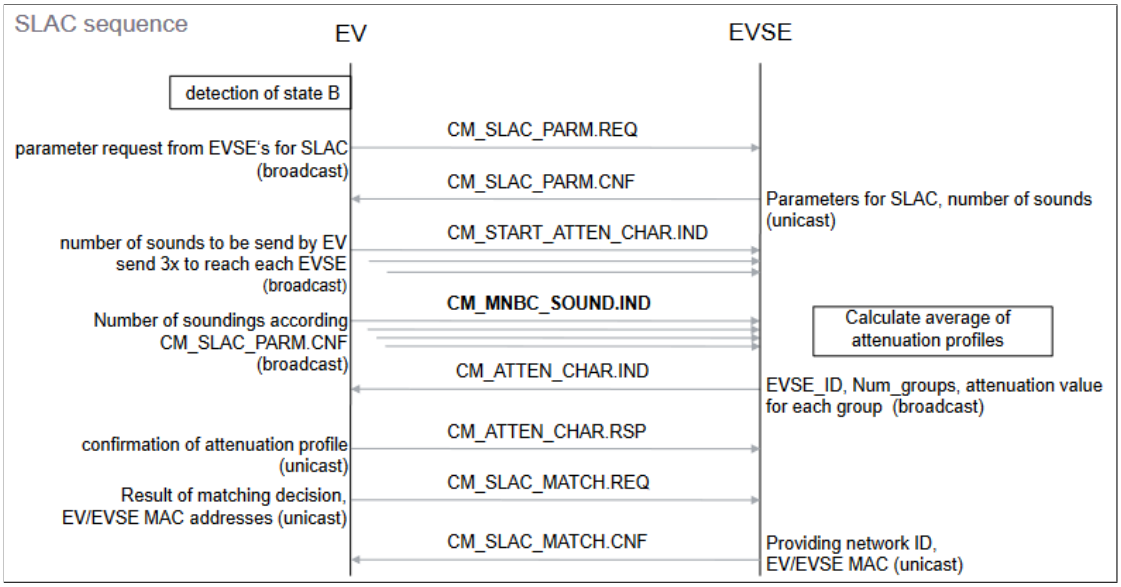


Figure 1.23 SLAC sequence [51]

unique IP addresses are used using Neighbor Discovery Protocol (NDP), and to use / implement ICMPv6 to send error messages. [34, 51]

As a result, the entities communicating during the charging process have valid and unique IP addresses. This layer is responsible for routing and switching connections, as well as determining the physical path for data transmission. [51]

ISO/OSI 4. Layer

Layer 4 assumes the successful establishment of ISO/OSI Layer 3. The task requirements for all entities that communicate with each other in high level communication are as follows: implementation/use of Transmission Control Protocol (TCP), implementation/use of User Datagram Protocol (UDP), implementation/use of Optional Transport Layer Security (TLS). [34, 51]

This layer guarantees a reliable TCP connection, fast UDP and secure TLS for communication between entities (usually EV and EVSE, generally an entity implementing according to ISO 15118-2). The fast UDP connection and data transfer is used only at the beginning of the communication to find the IP address and the SECC port on which the TCP (and TLS) connection is available for communication. UDP is used for SDP (SECC Discovery Protocol), where the client (EVCC) sends a multicast request and expects a response from the server (SECC). The request includes information on whether the EVCC requires TLS communication. The SDP response includes IP and port information, as well as whether a TLS or plain TCP connection will be used. Depending on the SDP response, either the communication is terminated (e.g. SECC does not offer TLS but EVCC requires it) or a plain TCP or secure TLS connection is established between EVCC and SECC. Thus, further communication takes place using TCP or TLS protocols. It should be noted that the definition of SDP is only within the application layer. [34]

The fourth layer is used to guarantee that the data flow flows without errors and congestion without loss or duplication. [51]

ISO/OSI 5. Layer

Similarly to the previous two layers, this layer also requires the successful establishment of the previous layer (i.e. layer 4) as a prerequisite. V2GTP is used in this layer, and it is fully described and defined in [34]. V2GTP is a protocol used for transferring V2GTP messages between two V2GTP entities. The protocol data unit comprises a header and a payload. The payload type is divided into three types: V2G message, SDP request, and SDP response. More detailed information about the payload for these types is available in the application layer description in [34]. Entities must therefore implement and use this protocol to communicate within Layer 5. This layer handles the establishment, management, and termination of connections between entities. It uses IP addresses and ports for this purpose. As a result, it guarantees bidirectional exchange of data between V2G entities. [51]

ISO/OSI 6. Layer

In addition to the fact that this layer, like the previous one, requires the successful establishment of a lower layer, it also has a requirement that all entities must use the encoding format as defined by W3C EXI (Efficient XML Interchange) 1.0 [51]. EXI is used because XML is used to represent V2G messages [34].

The first task that this layer has to provide is encoding and decoding using the EXI format. During encoding process, XML data are converted to EXI (binary format) based on XML Schema (Schema-informed Grammars) to EXI (binary format), as defined in [34]). The next task of this layer according to [51] is to establish/agree on a protocol for the application layer. This is done using a handshake that includes V2G messages: “supportedAppProtocolReq” (for EVCC) and “supportedAppProtocolRes” (for SECC). According to ISO 15118-2, this is also referred to as

“EXI settings for application layer messages”. An agreement is reached on the namespace to be used, for example, “urn:iso:15118:2:2013:MsgDef”, which indicates the use of ISO 15118 2.0 for encoding and decoding EXI streams. The structure of V2G messages is clearly defined within the XML schema. [34, 51]

The successful implementation of this layer ensures compatibility in data transfer between individual V2GTP entities.

“Layer transforms system dependent data into an independent shape and enables thereby the syntactically correct data exchange between different systems. It can be viewed as the translator of the system.” [51]

ISO/OSI 7. Layer

The successful establishment of the sixth layer is a prerequisite. Within the application layer, two types of payload are used: SDP or V2G messages. To obtain the IP address and port number of the SECC, the EVCC sends an SDP request. After obtaining this information, the EVCC initiates a TCP (or TLS) connection to the SECC. After agreeing on the application protocol (as mentioned above), the transmission of V2G messages can begin. These messages contain various information related to the charging process, such as identification, precharge, charge, and security check.

The [34] describes the structure of the PDU for the SPD request and response. Furthermore, chapter “Application Layer messages” defines how communication takes place within the application layer and defines the individual messages that can be exchanged between entities during the charging process. The messages are differentiated into common messages and then specific messages for AC and DC charging. V2G messages use a presentation layer based on the EXI format. [34]

In the context of V2G communication, the EVCC acts as a client (service requester) and the SECC as a server (service responder), creating a client/server architecture. Two message sets are distinguished: V2G application layer protocol handshake messages and V2G application layer messages. The first set is only used to agree on which application protocol and its version will be used. [34]

This layer therefore takes care of the message exchange during the charging process and also the initialization of the charging process. [51]

1.4.3 AC charging communication session

This is a summary of the communication for charging mode 3 (AC charging), which uses HLC communication. It is the most important mode and type of communication for this work and the tool under development. The assumptions for enforcing HLC are the same for both mode 4 (DC charging) and mode 3. The HLC mentioned is in accordance with the ISO 15118-2 standard.

The prerequisite is therefore a compatible EVSE and EV, i.e. a correct implementation according to IEC 61851 and ISO 15118 (especially ISO 15118-2 and ISO 15118-3).

Before passing the EV and EVSE through the charging cable, state A, i.e. EV not connected, is detected in the EVSE. After the charging cable is connected, the CP status (Vehicle state) changes to state B. Following this, the EVSE sends a PWM signal set to 5 % duty cycle (IEC 61851) to force and apply HLC within the charging process.

The PLC modules for EVCC and SECC initiate the SLAC process. After a successful SLAC process, they are connected in the same AVLN, establishing the data link layer. The PLC modules are also used for bridging IP-based communication within this AVLN.

After successfully establishing the AVLN, the next step is to initiate the SECC Discovery Protocol (SDP). The purpose of SDP is to enable the EVCC to obtain the IP address and port of the SECC for communication at the transport layer.During this process, the EVCC sends an SDP request to the local link multicast IPv6 address and UDP port 15118. The SDP request

includes a security byte that specifies whether to use TLS or not. The security byte determines whether TLS should be used or not. Additionally, the SPD request includes a second byte that indicates the required transport protocol, which is typically TCP. The SECC responds to this request with an SDP response that contains the requested IPv6 address and port, a security byte, and a byte indicating the transport protocol. If the security byte in the SDP response is different, then the EVCC decides whether it wants to continue the communication or not. If EVCC chooses to proceed, it can establish a transport layer connection to the SECC using the provided IPv6 address and port. [34]

The previous two points (paragraphs) are summarised in the figure (Figure 1.24) on a label called "Establishment of IP-based connection via PLC".

V2GTP builds on the successful establishment of a transport layer connection (typically using TCP). V2GTP is also used to transfer V2G messages. All messages shown in the figure (Figure 1.24) are V2G messages. It is important to note that within the definition of a PDU for V2GTP, a distinction is made between header and payload. The header contains information about: the V2GTP version, the inverse of V2GTP, the payload type and the payload length. The payload type distinguishes the following types: EXI encoded V2G message, SDP request message, SDP response message. ISO 15118-2 also leaves room for the type of payloads that are specific to each manufacturer. SDP messages were already simplified above with their meaning, more detailed information about the exact structure of the PDUs for these messages is defined in the standard. [34]

EXI encoded V2G Message can be further divided into: V2G application layer protocol handshake messages; V2G application layer messages. Messages used for handshake are: "supportedAppProtocolReq" and "supportedAppProtocolRes". The request is sent by EVCC and offers possible application protocols for communication, specifically the protocol namespace, version (major and minor), priority for the protocol, and its SchemaID. The SchemaID is used by the EVCC to identify the protocol. In response, the SECC sends the response code and SchemaID of the protocol it supports and has the highest priority for the EVCC. [34]

The V2GTP protocol for EXI encoded V2G message transfer requires agreement on the application protocol. The exact version and type of application protocol is important for the communication and exchange of V2G application layer messages. The agreement about the application protocol is done by the aforementioned handshake. After a successful handshake of the application protocol, the exchange of V2G application layer messages can proceed. The next transmitted messages, i.e. V2G messages, contain a header and body. [34]

The header contains a mandatory SessionID that is used to identify the V2G session between the EVCC and SECC. Also included is the optional Notification, used by the SECC to transmit additional information, for example, regarding errors. The last optional part of the header is xmlsig:Signature, which is used if the message requires a signature (based on the ISO 15118-2 message definition). The specific type of V2G message is differentiated by body. An example of a message is for example "SessionSetupReq" (sent by EVCC) and its response "SessionSetupRes" (sent by SECC). In this case it is indicated that the body type is "SessionSetupReq". Each message type (body type) has its own fields (parameters) and their meaning as defined in ISO 15118-2. For a better understanding of the message type distinction, I refer to [34], specifically "Figure 19 - Schema Diagram - V2G message". [34]

The next message after a successful handshake is "SessionSetupReq" and the corresponding "SessionSetupRes", their main task is to exchange the SessionID within the header of the message. Other information such as EVCCID and SECCID are transferred within these messages. The V2G communication session is initiated by the pair of these messages. As already mentioned, the SessionID identifies the V2G communication session. [34]

The message used to discover the services offered by the charging station is called "ServiceDiscoveryReq". The response to this message is "ServiceDiscoveryRes", which contains information on the available services. These services may include AC single-phase charging, AC three-phase charging, Direct Current (DC) charging, and identification mechanisms such as EIM or PnC.

Optional Value Added Services (VAS) such as Internet access to download additional data may also be available. This pair of messages is mandatory for AC charging communication session. [65]

An optional message to get details about the offered services is “ServiceDetailReq”, to which SECC responds with details for the requested service using “ServiceDetailRes”.

The next message is again mandatory and follows when everything is clear regarding the charging mode, identification mechanism and VAS. It is a request called “PaymentServiceSelectionReq” and its purpose is to select the method for authentication and charging authorization. There are two methods to choose from, EIM and PnC, both explained earlier in the text. The corresponding response from SECC is called “PaymentServiceSelectionRes” and contains a response code that indicates the acknowledgment status. [34]

If the PnC method is selected, then the EV must contain a digital contract certificate. This certificate is used for automatic authentication and authorization against the EVSE. In case the EV does not have the certificate installed or has expired, then “CertificateInstallationReq” and its corresponding “CertificateInstallationRes” can be used to obtain and install it. If the certificate is close to expiration, the EV can have a mechanism to detect this situation and “CertificateUpdateReq” can be used to request update of the certificate. [34]

The following message is only for the case where the PnC method is used for authentication and authorization. The request “PaymentDetailsReq” is used to provide a contract certificate from the EV charging station. The charging station performs authentication and authorization based on the received certificate and responds by sending a “PaymentDetailsRes” within this message and also sends a GenChallenge, which is used within the following message. [34]

The pair “AuthorizationReq” and “AuthorizationRes” is to help prevent replay attacks [65]. If PnC is used, then GenChallenge is sent in the request, which EVCC got from the previous message “PaymentDetailsRes”. If EIM is used, then the message is empty. This is a mandatory message pair. [34]

Another mandatory message is “ChargeParameterDiscoveryReq”, which sends the EV to the charging station to provide its charging parameters. In this message, the EV provides information such as: status information about the EV and additional charging parameters. Additional charging parameters can be, for example: estimated energy amounts for recharging the vehicle, capabilities of the EV charging system and the point in time the vehicle operator intends to leave the EVSE. In the response called “ChargeParameterDiscoveryRes”, the SECC informs the EV regarding applicable charge parameters from the grid’s perspective, as well as the charge schedule, information on cost over time, cost in relation to power demand and amount of energy. More information on these messages can be found in ISO 15118-2. [34]

This is followed by “PowerDeliveryReq”, which is used by the EVCC to inform the SECC about the start of charging, or its stopping, or other information about charging such as charging profile (i.e. maximum amount of power drawn over time). If the message parameter called “ChargeProgress” is set to “Start”, then the start of charging is detected and the EV reduces the voltage from 9 V to 6 V using resistors and thus the state C (“EV detected and ready for charging”) occurs. SECC sends “PowerDeliveryRes” as a response with information if power will be available. [34]

After the above messages, if everything went without problems, the charging loop process is started, in which the pair “ChargingStatusReq” and “ChargingStatusRes” is mandatory. These messages are exchanged regarding the charging status and are used as a basis for sanity checks on the meter readings provided by the SECC. These messages are iteratively exchanged between the SECC and the EVCC during the charging loop, allowing the EV to control and verify the power drawn from the EVSE. The SECC can use the “ReceiptRequired” parameter (in message “ChargingStatusRes”) to force the EVCC to send “MeteringReceiptReq”.

The optional message in the charging loop is related to the previous message and it is the mentioned “MeteringReceiptReq”, which serves for the purpose of signing the meter info record. In the sent request, the EVCC acknowledges receipt of the data that was transmitted in the

"ChargingStatusRes" message. The data is the MeterInfo record, SessionID and the SAScheduleTupleID. In response to this request, the SECC informs the EVCC whether the receipt has been successfully received and accepted.

To terminate the charging loop, EV sends "PowerDeliveryReq" but this time with the indication of charging stop, i.e. the "ChargeProgress" parameter is set to "Stop". [34]

V2G communication is terminated using the pair "SessionStopReq" and "SessionStopRes". EVCC indicates in the request if it wants to terminate the communication or just pause. If the session is only paused, then some parameters are temporarily stored in EVSE and are used when the same session is resumed. This was the entire description of an AC charging session within V2G (ISO 15118).

V2G charging session for DC charging is very similar. To control the charging process, the message pair "CurrentDemandReq" and "CurrentDemandRes" is used instead of "ChargingStatusReq/Res". Furthermore, DC charging has three additional message pairs that are specific to this charging and are not used for AC charging, namely CableCheckReq/Res, PreChargeReq/Res, WeldingDetectionReq/Res. For detailed information on DC charging messages, please refer to [34].

More detailed information regarding all messages and parameters is given in [34]. The purpose of this thesis is not to copy the definitions, meaning, or requirements given in the mentioned standard.

1.5 OBC

The abbreviation OBC is used for two terms: on board charging and on board charger, depending on the context. In this thesis assignment, OBC is used as an abbreviation for on board charging. On board charging refers to the process of charging a EV using an on board charger, which converts AC to DC. Therefore, if an on board charger is used, the charging process is AC charging. On board charging (OBC) is another term for AC charging. With these statements in mind, the research of OBC (on-board charging) is mentioned in the above chapters and sections that describe AC charging, for example, in the charging methods, types, interfaces, standards and different charging modes associated with how communication for a particular mode takes place.

From the previous chapters of the theoretical section, it is clear that AC charging is defined within 3 charging modes. There is no communication for the first mode, so it is not interesting at all, in terms of software evaluation or enumeration. The second mode contains only low level communication, i.e. a PWM signal on the physical layer with different voltage and duty cycle values. So the second mode is not very interesting either, as it doesn't contain much room for any implementation errors. Moreover, basic signalling is only a basic communication to transmit the most necessary information to start or stop charging (as can be seen from the description in the tables Table 1.4 and Table 1.5).

More interesting is the up to 3 mode for which HLC is possible. HLC communication enables more complex data transfer, so I focus on AC charging with HLC communication. HLC is enforced for the third mode of AC charging using a 5 duty cycle PWM just like for DC charging (i.e., mode 4). Additionally, in the EU market, CCS is used in vehicles, which defines Type 2 inlet or Combo 2 inlet for AC (at the same time for both AC and DC). If the vehicle has a Combo 2 inlet, the tool under development can also be used for DC charging because HLC communication uses PLC technology via CP circuit for both AC and DC charging. When using the Combo 2 inlet, the OBC (on board charging) port is the same for both AC and DC (I am considering only the communication wires). All this is implied by the information given in the previous chapters of the theoretical part.

Therefore, I do not consider charging modes 1, 2 and also mode 3, which uses only low level communication.

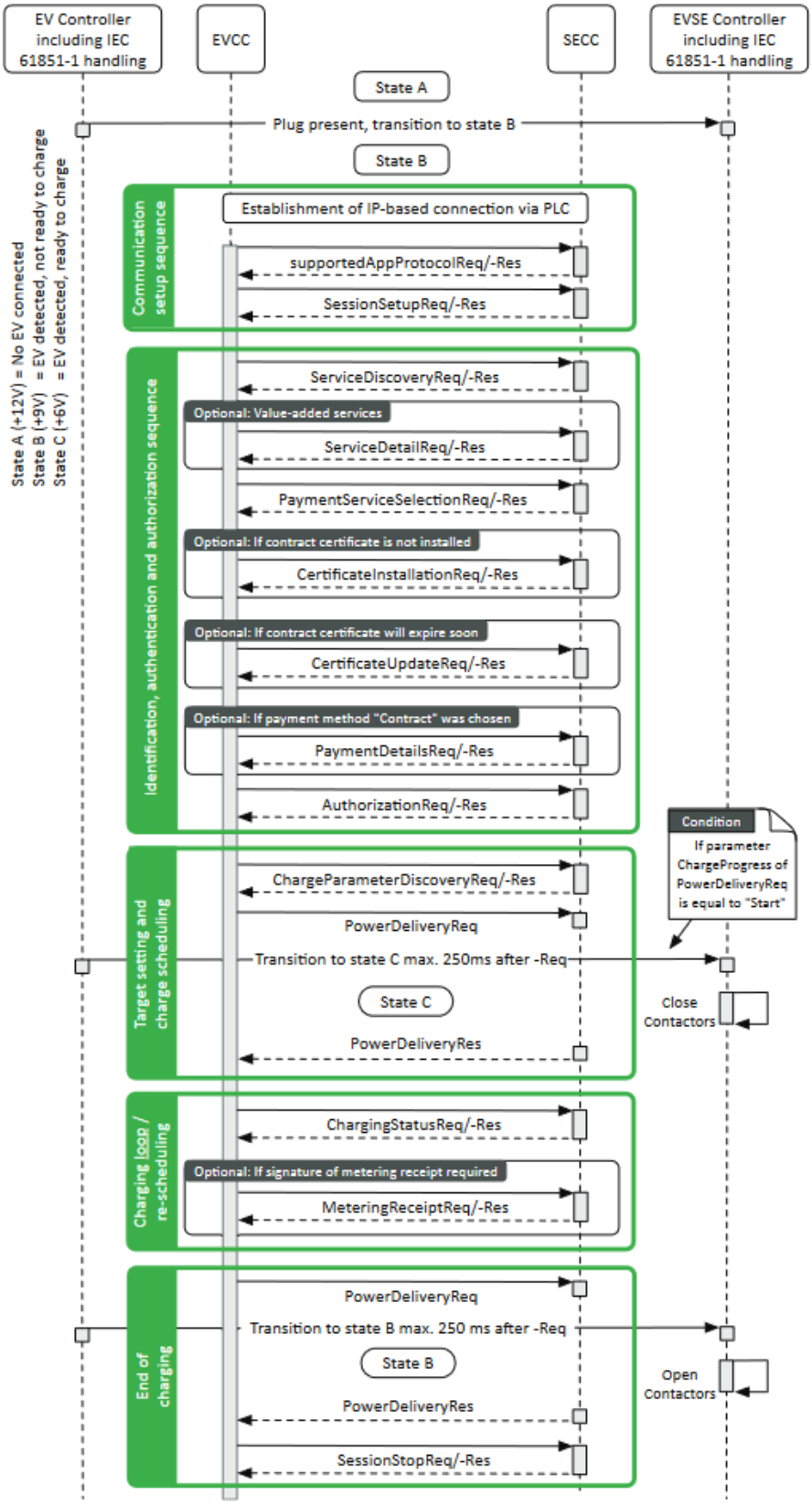


Figure 1.24 AC charging session – communication summary [66]

1.5.1 OBC security

In the same way that doors protect the fuel tank in combustion engine vehicles, this is implemented in electric vehicles (protect charging inlet) and can be considered as physical security against unauthorised access. In the context of OBC (on board charging) security I focus more on the software side and the implementation of the standard for HLC (specifically ISO 15118). I leave out the less interesting low level communication because it does not involve the transmission of complex information as HLC does, and there is less room for error in the implementation of handling this communication.

According to [6], the vehicle inlet is one of the intrusion points for electric vehicles. In the past, this was not such a problem from a security point of view because only a simple electric cable was used without any communication. With the further development of electric vehicles and charging stations, communication during charging has also evolved, for example in Japan CAN is used for communication between the charging station and the vehicle. If the device communicating with the charging station is directly connected to the in-vehicle network without message filtering, the charging port can be a significant risk. This approach could then lead to allowing an attacker to reprogram the ECUs in the vehicle network. [6]

As I have already mentioned several times, I focus on the EU market and therefore CCS, in which PLC technology is used for communication. In the context of testing and evaluation of the charging port, basic signalling is not of interest, only HLC which is compliant with IEC 61851 AND ISO 15118 (or DIN SPEC 70121).

DIN SPEC 70121 is an example of a standard that defines HLC and exposes the communication to attackers because TLS is not required for this communication. This standard focuses and supports only on DC charging. The problem with not using an encrypted connection is not just eavesdropping, but also the possible injection of malicious data that can lead to the transmission of non-valid information. A possible consequence may be incorrect information about the required voltage/current, which can lead to damage to the EV battery. [53]

It is ISO 15118 that defines the use of TLS to prevent the problems listed for DIN SPEC 70121. In ISO 15118-2 TLS communication is only optional for certain cases (AC EIM, DC EIM for EVCC), for the second generation the use of TLS is mandatory for all cases. Mandatory TLS is for a PnC that uses digital certificates that are end-to-end encrypted and verified by a PKI (public key infrastructure) system. More detailed information about the PKI and the certificate can be found in [34] (or in ISO 15118-20). Another mechanism to increase the security of communication (to ensure intergity) is XML signature, which is mandatory for some parameters in selected messages. These are mainly messages related to payment.

Since ISO 15118 defines and requires the use of different technologies and protocols, it opens up space for possible errors in their implementation. This is particularly relevant for the higher layers (layers 3-7) for HLC, which are defined in ISO 15118-2 and its second generation (ISO 15118-20). That is why I focus on these layers within the implemented tool.

From a security perspective for lower layers of the HLC, the SLAC process is important, for which potential security issues are discovered and discussed by [67]. In particular, it focuses on the security issues of SLAC communication where entities argue on AVLN. Since for PLC communication all packets are broadcast, the electric vehicle can be detected by multiple EVSEs and vice versa. As I mentioned during the high level communication session, SLAC has to ensure that only physically connected EV and EVSE are in the AVLN. This process is also intended to prevent potential security problems such as: bad associations and billing errors. [67]

One security issue of the SLAC process is that an attacker can bypass this mechanism by transmitting tampered attenuation values. This makes the EV think it is physically connected to the attacker's EVSE. In the SLAC process, this is a message called "CM_ATTEN_CHAR.IND". This is the first weakness of this process, but [67] describes other weaknesses. A fake PEV that initiates a SLAC sequence to join some EVSE (not physically connected) can do a similar thing. [67]

The first of the other issues is related to the DAK (Direct Access Key), which is used for remote configuration via the NMK (Network Membership Key) for PLC devices via the power line interface. It allows to set the NMK without direct access to the PLC modem, no access via the local interface of the PLC modem (e.g. ethernet interface) is necessary. Using the DAK key to set NMK via the power line has its weaknesses. The discussed weakness in [67] is that the DAK key in HP AV devices was generated by a known algorithm that used a derivation from the MAC address of the PLC device. This made it possible to find the DAK of all Central Coordinators (CCo) PLC devices. Central Coordinators (in home networks) are usually PLC devices that are connected to an internet router, which in this context always means EVSE (PLC modem in EVSE). If V2G PLC manufacturers use the same predictable technique to generate DAK, then this weakness can be exploited in the communication between EV and EVSE. [67]

To introduce the second weakness, it is first necessary to mention that HPGP offers three configuration modes for the PLC modem. These are as follows: PEV, EVSE and unconfigured (act like domestic plug). The second weakness is a design flaw in the SLAC process that any PLC module in PEV mode can capture a packet from EVSE that contains an NMK (Network Management Key). It is the PLC modem in PEV mode that is able to eavesdrop on NMKs sent by all EVSEs during SLAC processes. In addition, the captured NMKs were in clear form because Management Message Entry (MME) packets as well as packets for the SLAC process are broadcast over the power line and are generally not encrypted. So anyone on the same electrical network can intercept and read these packets. [67]

Once connected to the AVLN, it is possible to monitor and discover devices on this network, which is more interesting for attacking SECC devices than EVCC. This is because the SECC acts as a server and so may contain various other services, whereas the EVCC acts as a client and therefore generally offers nothing interesting in terms of additional services. Since my focus in this thesis is on EVs and therefore on EVCC, I will only briefly mention information about services for SECCs here. SECC can generally provide additional services such as SSH/Telnet, FTP/SFTP, and/or (management) web services and others (depending on the design of the EVSE in mind). [67]

In AVLN, an attacker can set his PLC modem to promiscuous mode to intercept all packets. According to [67] two types of MITM (Man in the middle) attack can be performed: the classical way with an ICMPv6 Neighbour spoofing attack; racing the SECC procedure. Racing the SECC procedure offers the attacker a more stable solution than the first mentioned MITM attack, because by subnetting the SDP response with a different port and IP address, it makes the EVCC think that the genuine SECC TCP server is the attacker's fake SECC. The SDP request and SDP response are performed only at the beginning of the communication before establishing the mentioned TCP (or TLS) connection. For a successful attack an attacker must be very fast by retrieving the NMK, configuring his PLC kit and then sending fake SECC replies for a while. The result is that the EV thinks it is communicating with the correct SECC, but instead it is communicating with a fake SECC, thanks to a spoofed IP address and port in the SDP response. [67]

In addition, SDP also contain information about the security used by setting the security byte in the SDP request or response to indicate "secured with TLS" or "No transport layer security". Therefore, an attacker may attempt to prevent the use of TLS by setting a security byte in the SDP response to indicate that the SECC does not support TLS. According to [34] it is the decision of the EVCC whether to accept or terminate the communication without using TLS. Depending on the EVCC implementation, it may be possible to downgrade V2G communication in this way and then read the transmitted data in plain text.

In terms of OBC security and HLC communication, weaknesses can be found in the implementation of ISO 15118-2, particularly in the handling of V2GTP packets and V2G messages. Proper encoding and especially decoding and interpretation of XML documents (decoded EXI V2G message) is also related to V2G messages. These are the aspects that I focus on in my tool. The result of [67] is a tool called "V2GInjector". This implemented tool is used to: analyse

V2GTP layer; extract EXI data; encode/decode data for V2G purpose; inject EXI data. At the base, however, it does not offer any automated mechanisms for testing V2G communication.

1.6 Summary a decision

This section builds on the entire previous theoretical section and mentions the most important things from it. Decisions regarding the implementation of the tool are made based on these points.

The theoretical part described what OBC and OBC port mean. OBC generally stands for on board charging, it is a type of charging that indicates the use of a device called "on board charger". This device is mainly used to convert AC to DC. On board charging is, in other words, AC charging. Therefore, the OBC port is the vehicle inlet through which AC charging takes place. Important features and information are mentioned in the chapters: Charging, Charging Communication a OBC.

Since the target is an electric vehicle in the EU market, then CCS is crucial. It includes what standards are to be used for the different categories within charging (plugs, socket, communication, …). It is evident from the section titled Charging Interfaces that Type 2 or Combo 2 inlets are used for AC charging in the EU. Combo 2 serves as one input for both AC and DC charging, only the connector differs (shown in Figure 1.16). Depending on the manufacturer, the OBC port can be considered as a Type 2 inlet or Combo 2 inlet (for AC charging).

CCS mentions the communication standards to be used, including IEC 61851 and ISO 15118 for both AC and DC, or DIN SPEC 70121 for DC only. IEC 61851-1 defines low level communication, which is described in Low Level Communication. The ISO 15118 standard or DIN SPEC 70121 is used for HLC. ISO 15118 is the successor of DIN SPEC 70121 and in addition to HLC communication for DC charging it also defines communication for AC charging, which is crucial for this work (DIN SPEC 70121 is therefore not relevant).

In this summary it is also important to mention that IEC 61851-1 also describes different charging modes (see Table 1.3). The first three modes are defined for AC charging, but only mode 3 allows HLC communication. In the context of OBC, the third mode is important for this thesis. The explanation for why mode 3 was chosen is provided in sectionOBC. The reason for this is that HLC communication increases the risk of implementation errors. It is important to note that this only applies to AC charging and ISO 15118 is the relevant standard, not DIN SPEC 70121.

As mentioned in High Level Communication, the HLC can be divided into two parts, the first is defined in ISO 15118-3 and the second in ISO 15118-2 (or ISO 15118-20). The first part deals with the physical and data link layer in the ISO/OSI model (for conductive charging). The remaining layers are described in ISO 15118-2. The choice of higher layers is also because the lower layers (specifically the SLAC process) have already been covered in detail, for example, by [67]. Another reason is that the PLC modules and my whole board setup allows to use SLAC for EVSE and PEV (EV) out of the box. Therefore, the lower layers, both physical and data link layers, are guaranteed owing to the modules and firmware used. The selected boards and PLC modules support HPGH. Therefore, the focus is on the higher layers, specifically the ISO 15118 standard. The selection of the part of the standard was discussed in more detail in ISO 15118. The description of the board setup is discussed more in the following chapter.

In my research of tools and implementations, I found more tools that simulate EVCC or SECC. The only tool relevant for security testing that was freely available is "V2GInjector" by [67]. According to [67] their implemented tool can be used to: analyze V2GTP layer; extract EXI data; encode/decode data for V2G purpose; inject EXI data. It is also mentioned that it does not yet offer any automated mechanisms for testing V2G communication. According to my experience and documentation of the tool, more manual work is required for sending and modifying than would be desirable for an automated tool. This is one of the reasons why the tool

I am developing aims to provide automated testing and easier configuration options for V2GTP messages and their content.

The aim of my tool is to help in the enumeration and partial evaluation of EVCC (via the OBC port). So I am focusing on the part of SECC that is supposed to communicate with EVCC according to ISO 15118-2. I assume the following: successful HLC enforcement using low level communication (specifically PWM 5 duty cycle). Subsequently, successful setup of the physical and data link layers using HPGP (per ISO 15118-3). I build on these prerequisites and implement SECC according to ISO 15118-2. From the previous theoretical section it is clear that ISO 15118-2 also supports DC charging and only some message sets differ. So the tool can be used for off board charging (DC charging) in addition to on board charging, if the EV has a Combo 2 input (not just a Type 2 inlet).

The following practical part builds on the entire theoretical part in combination with definitions from ISO 15118-3, ISO 15118-2. In the practical part, ISO 15118-3 does not serve directly for the implementation of the tool, but helps with the selection of hardware that will provide support for HPGP (physical and data link layer). The implementation of the tool is based on ISO 15118-2.

Chapter 2

V2G Testing Environment

This chapter builds on the knowledge from the theoretical part. Using the previous information, this chapter describes what hardware and software were required as a prerequisite for the subsequent implementation of the tool. First, the V2G setup is described, where I focus on what hardware is needed to support V2G communication (i.e., communication according to ISO 15118). In the context of hardware, I also describe the firmware needed and the interconnection of the hardware components. Subsequently, I focus on the software that I have used for a practical exploration of V2G communication. In connection with the exploration of V2G communication, the Wireshark setup that is needed to read this communication is also described.

2.1 V2G setup

Firstly, it is important to note that low level communication (basic signalling) and HLC enforcement using PWM 5 % duty cycle is a prerequisite. My setup is based on the assumption that HLC will be enforced and used, but I do not deal with its implementation and assurance in this thesis. Ensuring this signaling may be a possible extension of my tool (or FW for the board module). According to the theoretical part, it is clear that HLC is done through PLC and the definition of the first two layers of the ISO/OSI model is based on HPGP. Based on these prerequisites, I first searched for available hardware that would meet the following parameters: PLC support; PLC-to-Ethernet bridging support; use of HPGP; local connection to PC via RJ45.

The main purpose of these requirements is to guarantee that two of these boards will communicate with each other via a PLC while using HPGP. Therefore, these boards will ensure successful interconnection within the physical layer and also successful connection to the same AVLN (using the SLAC process). From the perspective of ISO 15118, they are therefore to provide the first two layers of the ISO/OSI model according to ISO 15118-3. According to [2] this is provided by the PLC modem (module). This module is according to the picture "Communication architecture set-up PLC 1" in [2] connected with EVCC if it is an electric vehicle. In the case of an EVSE, the PLC module is connected to the SECC. The PLC modules (in EV and in EVSE) are physically connected and on the same AVLN. These modules provide communication between the EVCC and the SECC via PLC. From the point of view of [2] the HW (i.e. HW boards) can be considered as PLC modules. These PLC modules (the HW solution for my tool) support the communication according to ISO 15118-3. Other layers of the ISO/OSI model within the HLC communication are implemented by EVCC and SECC according to ISO 15118-2, which builds on the successful establishment of the data link layer according to [63] (provided by PLC modules).

The requirements for local RJ45 connectivity and support for PLC-to-Ethernet bridging are

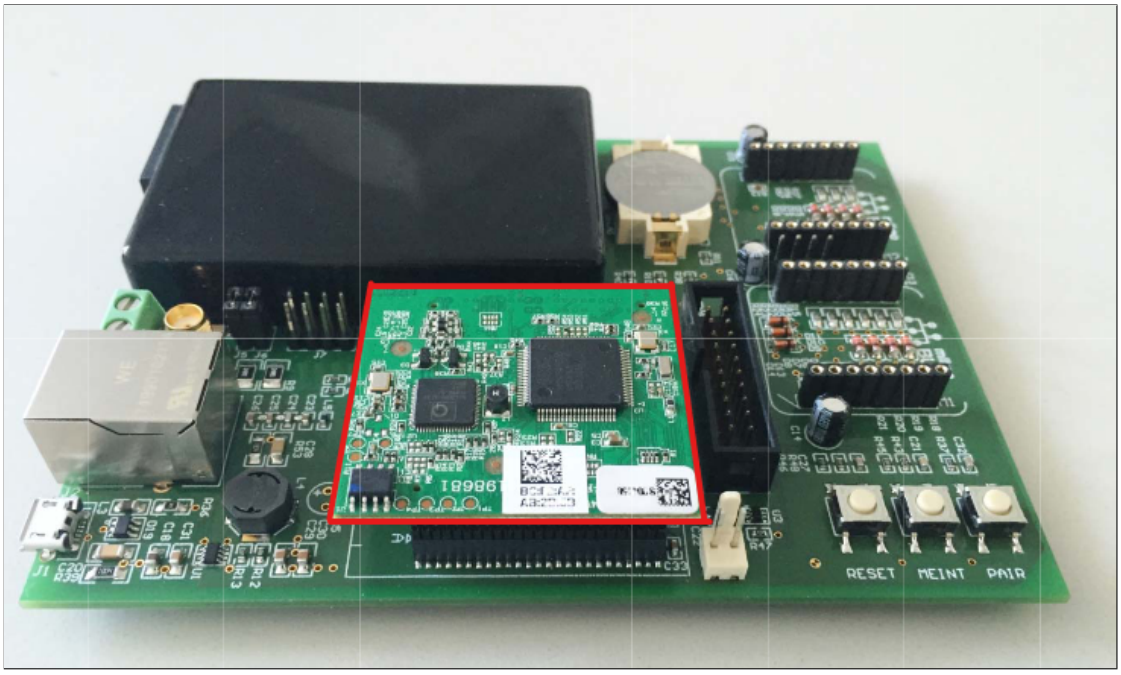


Figure 2.1 dLAN® Green PHY eval board with dLAN® Green PHY Module (red border) [68]

because EVCC and SECC SW run on a PC. PLC modules are used to interconnect EVCC and SECC in order to simulate the same communication architecture (communication over PLC) as specified in [2]. The assumption that EVCC and SECC will be simulated by software running on a PC is based on a previous survey of V2G implementations available on the Internet.

Summarising the above, I arrive at the following facts. Firstly, it is necessary to obtain HW (with firmware) that supports the aforementioned assumptions. Next, use one of the available EVCC and SECC implementations. Establish communication between EVCC and SECC using PLC modules (HW). In combination with the [34] standard, explore the communication between EVCC and SECC using Wireshark. If the communication is not readable find suitable dissectors for Wireshark and then examine the communication. Once these steps have been successfully completed, focus on implementing the tool, which is one of the goals of this thesis.

2.1.1 Boards

Based on previous discussion I have found suitable boards for these purposes. Also thanks to the fact that charging of e-cars is developing a lot and so different HW solutions are available for own (home) use. These are the dLAN® Green PHY eval board II in combination with the dLAN® Green PHY Module (GPM), both are shown in the picture (Figure 2.1), the GPM is highlighted with a red border.

First, I provide a short description for the mentioned board and then for the module. According to the description in the documentation [68] it is an evaluation board, which is designed for the dLAN® Green PHY Module. Furthermore, this board offers PLC to mains-line or to twisted-pair connectors and provides access to many interfaces such as Ethernet, USB and RS232. In terms of functionalities, the most important is the PLC to Ethernet bridging, which is the default functionality. [68]

The dLAN® Green PHY eval board II serves as a translator. This board enables PLC-to-Ethernet bridging and is an evaluation platform for dLAN Green PHY modules over power, coaxial or twisted-pair cables. [69]

The “dLAN® Green PHY Module” is according to the documentation [70]: “an integrated device for transmitting and receiving data over the power line. It holds all functions necessary for the easy creation of Green PHY network devices”. In this module, the QCA7000 Green PHY processor is interfaced and supported with the LPC1758 host processor to add functionality and additional interfaces. [70]

The module supports Ethernet to PLC, then HomePlug Green PHY and thanks to that it enables connection to AVLN [70]. Configuration of this module is enabled by Software Development Kit (SDK), which is available from [71]. According to the description on [72] the module is suitable for intelligent data transfer between EV and EVSE.

This basic information is sufficient for the description of the V2G setup and more detailed information regarding functionalities, block diagrams and other specifications regarding these two HW components can be found in [68] (for eval board) and [70] (for GPM).

2.1.2 Boards configuration

To guarantee access to different interfaces and to allow communication with other devices, the module must be connected (shown on Figure 2.1) and used with the eval board. The eval board can also be used to flash firmware for the module. Using the mentioned SDK, the module can be configured as an emob-charger or emob-vehicle. This means that for the first case the module will be configured to SLAC in EVSE mode and for the second case the SLAC will be in PEV mode.

After obtaining the module and eval board, it was necessary to connect the module to the eval board. For use in ISO 15118 communications, the modules must be configured to use the SLAC process, one in EVSE mode and the other in PEV mode. Therefore, a module configured in EVSE mode acts as a server and in PEV mode as a client (the SLAC sequence is shown in Figure 1.23). The configuration refers specifically to the QCA7000 GreenPHY processor, so the “QCA7000_GreenPHY_Firmware” from [71] is used for the firmware update.

Now I briefly summarize how to perform the mentioned firmware flash. Since the procedure is platform dependent it is important to mention that the configuration was done from a Windows machine. First it is necessary to download “qca7000_1-2-4-00-1_kit_2022-03-11_windows.zip”, then unzip the archive and open the “qca7000_1-2-4-00-1_kit” folder. The tools that are in the “qca7000-update.cmd” script need a packet driver to send and receive Layer2 Ethernet frames. This can be done by installing devolo Cockpit or WinPCAP driver. In my case I used the first mentioned, i.e. Cockpit.

After running the above script without arguments, you can see a help on which arguments to use (shown on Figure 2.2). The first argument of the tool is the config-profile and the second is the MAC address of the module. If no MAC address is provided, the tool uses the first locally connected adapter. To show how I used the tool for flash firmware, I set “emob-vehicle” as config-mode without providing the MAC address (shown in Figure 2.3). The tool correctly detected the connected module and performed a firmware flash. In case the automatic discovery does not find the module, it is necessary to pass the MAC address without separators. For the second module I used config-mode emob-charger.

In the following description, I consider the module and the eval board as one device, which I will refer to as the HPGP device (or HPGP board). After the previous configuration, I have two HPGP devices that support communication via PLC and HPGP. One device is configured to act as an EVSE for the SLAC process and the second device as a PEV. Therefore, they can set up an AVLN with together, just as they would in the real case. This setup complies with ISO 15118-3 and provides an established data link layer for higher layers.

```
PS D:\FIT_CVUT\Ing\NI-DIP\Master's_Thesis\Devolo_Boards\dlan-greenphy-sdk-master\QCA7000_GreenPHY_Firmware\qca7000_1-2-4
-00-1_kit_2022-03-11_windows\qca7000_1-2-4-00-1_kit\windows> .\qca7000-update.cmd
Usage:
qca7000-update <config-profile> [mac-address]

 config-profile must be one of:

  iot-generic    IoT generic, optimized for performance: 50561 off (SLAC off)
  iot-conform    IoT over mains, optimized for conformity: 50561 on (SLAC off)
  emob-charger   e-mobility use as charging station: SLAC in EVSE mode (50561 off)
  emob-vehicle   e-mobility use as vehicle: SLAC in PEV mode (50561 off)

 mac-address is optional, if omitted, the first
 localy attached adapter will be programmed
```

Figure 2.2 QCA7000 GreenPHY firmware Flash – help

```
PS C:\Users\root\Desktop\Master's_Thesis\Devolo_Boards\dlan-greenphy-sdk-master\QCA7000_GreenPHY_Firmware\qca7000_1-2-4-
00-1_kit_2022-03-11_windows\qca7000_1-2-4-00-1_kit\windows> .\qca7000-update.cmd emob-vehicle
CFG: emob-vehicle
MAC: BCF2AFF1E24F
DAK: 8AE1301183DD2184DC59E6CDA5AC5004
MFT: devolo dLAN Green PHY Module [MT2489]
BC:F2:AF:F1:E2:4F: uploading firmware and PIB ... OK
waiting for reboot ... OK
PS C:\Users\root\Desktop\Master's_Thesis\Devolo_Boards\dlan-greenphy-sdk-master\QCA7000_GreenPHY_Firmware\qca7000_1-2-4-
```

Figure 2.3 QCA7000 GreenPHY firmware Flash – SLAC in PEV mode

2.1.3 Boards connection

After the previous steps, I have configured the HPGP devices to use the SLAC process. All that's left now is to physically connect the devices together (both connected to the power supply, of course) in order to use the SLAC process to build the AVLN. For the connection between both boards I use SMA coaxial connectors as PLC interface and based on information from the documentation ([68]) I left required jumpers open or closed. After these steps, the data link layer is ready for higher layers.

The higher layers of HLC communication are handled by EVCC and SECC controllers. The PLC module in the EV is physically connected to the PLC module in the EVSE and these modules assemble the AVLN using the SLAC process.[2]

In my setup the PLC modules are just HPGP devices and the EVCC/SECC are programs running on the host PC. To connect the EVCC and SECC (i.e. the PC) to its HPGP device, an RJ45 connector on the board and an Ethernet cable is used. Both EVCC and SECC each have their own Ethernet interface on the PC (thanks to a USB-to-Ethernet adapter).

To summarize, each board is connected to a power supply. In my case, both HPGP devices are connected via micro-USB-to-USB cables to the HUB, which is connected to the mains. Furthermore, the devices are connected to each other using a coaxial cable, which is connected to the SMA female Coaxial-Connector on each board, which serves as the PLC interface. The last thing left is to connect the HPGP device to the PC, both devices are connected via RJ45 Ethernet cable to the USB-to-Ethernet adapter (each HPGP device to its own). This network adapter is then connected to the PC. The communication between the SECC and the EVCC then goes from the PC via the Ethernet adapter to the HPGP, where Ethernet-to-PLC bridging takes place. The PLC communication is transferred to a second HPGP and from there bridging is again done to the PC.

The above mentioned connection of the components (without PC) is shown in the picture Figure 2.4. The EVCC is connected with a white USB-to-Ethernet adapter and connected to the HPGH with a blue Ethernet cable. SECC uses a black cable and a black adapter. Therefore, the EVCC and SECC run on a PC and each has its own network interface set up.

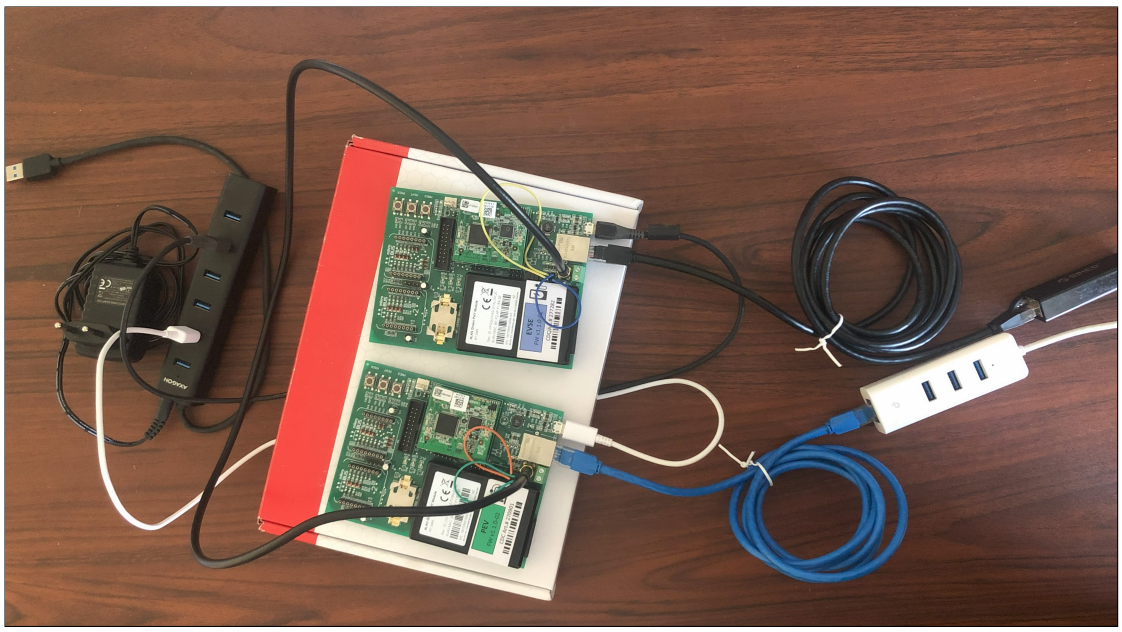


Figure 2.4 Boards Setup

2.1.4 SECC a EVCC

As a guest operating system for SECC and EVCC I use Linux, specifically Ubuntu. I have this operating system within a VM, which is also where the network adapters mentioned in the previous section are connected. This section therefore builds on the previous configuration of the boards and their interconnection as shown in the figure (Figure 2.4).

During my research on SECC and EVCC implementations I found for example "OpenV2G" and "Josev – Joint Operating System for EV chargers". The former is implemented in C and the latter in Python. Josev refers to two repositories, one for SLAC and one for the higher layers of the ISO/OSI model. The second repository referenced in Josev is called iso15118 ([73]) and it is the one that implements SECC and EVCC according to ISO 15118-2 (and DIN SPEC 70121 AND ISO 15118-20).

According to my decision, I chose iso15118 from Josev project, which in "Pro" version is a commercially sold product. It is a constantly updated project and because it is written in Python, it is faster and easier to understand than OpenV2G. Among other things, it contains better documentation and EVCC and SECC are clearly separated.

For an initial introduction to a real example of SECC and EVCC and the communication between them, the iso15118 repository seems to me to be the best candidate. So I continue to work with this repository as a valid implementation of SECC and EVCC. I am using both EVCC and SECC first, because I want to first make the communication between these controllers via HPGP devices work, which is to simulate the real behavior when PLC modules are used to communicate between EV and EVSE. Once this communication is operational, the next goal is to explore different controller configurations. The next task is to use the wireshark tool and configure it to read and interpret the HLC between controllers (more in Wireshark setup). On the acquired knowledge of how the communication works on a real example in combination with [34] I build the implementation of my tool. The tool focuses on the implementation of the SECC part. Then I only use EVCC as a reference implementation that communicates with my tool.

Here, I list the steps that are necessary to make SECC and EVCC via HPGP devices operational (boards connected according to the Figure 2.4 image). To give an easier understanding

of a successful configuration, I do not mention the problems encountered before reaching the correct configuration. So, I only show the final working configuration. I followed steps 1-4 of [73] in the same way. I modified the fifth step listed in the repository and split it into two parts.

EVCC and SECC (for the iso15118 repository) use a single .env file that defines which Ethernet interface will be used for communication. Since this is one file for both controllers, the following procedure had to be used. First change the line defining the interface for SECC and then run SECC. Again, change the .env file with the Ethernet interface for the EVCC and then run the EVCC. A detailed description is given below, this is just to understand why I am changing the .env file twice in the procedure.

For better orientation, the Ethernet interfaces were first renamed so that the name corresponds to the HPGP device the interface is connected to. I renamed the Ethernet adapter connected to SECC to "eth_station" and the adapter connected to EVCC to "eth_car". I then use these names in the .env file. According to [34], the communication is IP-based, specifically using the IPv6 protocol. Therefore, in addition to renaming, it is necessary to set IPv6 link local addresses from the same subnet for "eth_station", "eth_car". The configured IPv6 addresses are fe80::d237:45ff:fe88:b12a/127 for eth_car and fe80::d237:45ff:fe88:b12b/127 for eth_station. In order to rename the original Ethernet interface names in Ubuntu, I created a script (config_boards.sh) that renames the interfaces based on the MAC addresses provided. The script also sets the IPv6 addresses mentioned. With this setup, the adapters are ready for the next step, which is to run SECC and EVCC.

SECC and EVCC use the ".env" file for setting up the Ethernet interface, among other things. Because of the mentioned problem with the ".env" file, I have created my own ".env" files for both SECC and EVCC, which must be copied to the ".env" file before running the controller. For EVCC this file is called ".env.evcc" and for SECC ".env.secc." The files differ in the value of the variable "NETWORK_INTERFACE". For automated change of the ".env" file and starting EVCC or SECC, I created scripts (start_evcc.sh and start_secc.sh). First it is necessary to run the script for SECC. Once the SECC is running, the script for the EVCC can be run (the EVCC must have the SECC available). Then communication is done according to ISO 15118-2.

All mentioned scripts that help to configure and run the controllers are included in the repository of my tool in "scripts" directory. Alternatively, further explanation of them is in the README for my tool. The use of the scripts is demonstrated in the video, which is located on the attached media in the videos folder named "config_run_SECC_EVCC.mp4". In addition to configuration and startup, the video also shows a demonstration of my tool, which can also serve as a live sniffer (more in the chapters on implementation and use of the tool).

The summary for the previous description is given here. For the EVCC and SECC reference implementation I use the iso15118 repository available from [73]. The controllers (SECC and EVCC) use HPGP devices to communicate with each other. The controllers are connected to the HPGP devices via an Ethernet adapter. The EVCC has named the Ethernet adapter as "eth_car" and set the link local IPv6 address to "fe80::d237:45ff:fe88:b12a/127". The SECC has an Ethernet adapter named "eth_station" and a link local IPv6 address set to "fe80::d237:45ff:fe88:b12b/127". First, I use "start_secc.sh" to configure the .env file to use "eth_station" and then start the SECC (if any libraries are missing, they are installed using poetry, part of the script). The last step is to use "start_evcc.sh", which does similar things to the previous script, but for EVCC (interface "eth_car"). After this process, SECC and EVCC communicate with each other via HPGP devices (PLC modules), so the communication is based on ISO 15118 standards.

2.2 Wireshark setup

I use Wireshark (version 4.0.6.) for communication investigation. This communication is made possible by the whole previous process of building the boards, configuring them, configuring

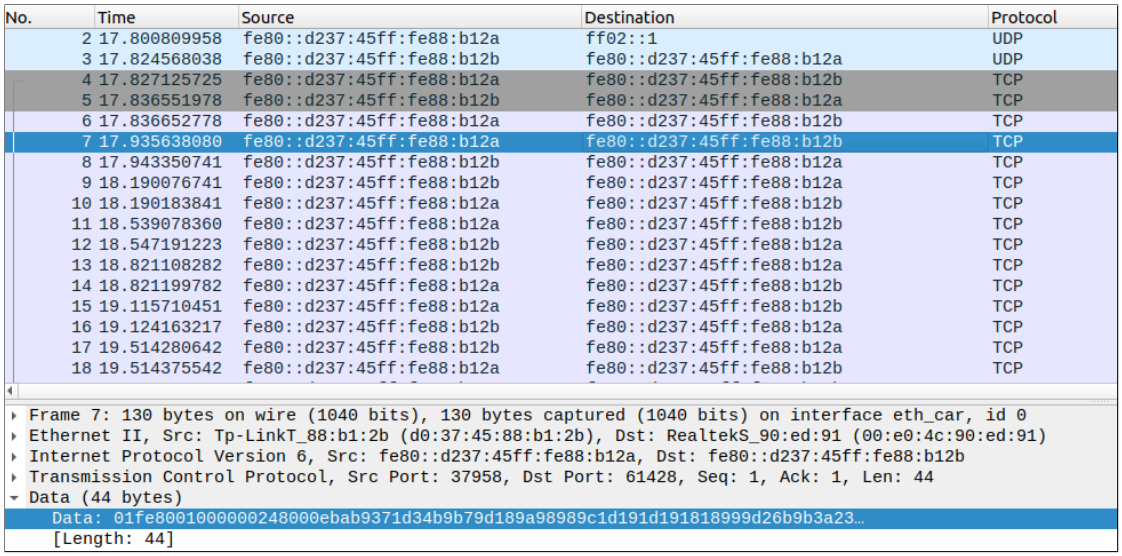


Figure 2.5 Wireshark captured communication – without V2GTP disector

EVCC, SECC and the connected Ethernet interfaces.

According to my original findings Wireshark does not recognize V2GTP by default, and therefore the transmitted communication cannot be decoded and read (can be seen on Figure 2.5. For the "Data:" label it can be seen that the communication is not decoded and the data is displayed in bytes (hex stream). For this reason it was important to find a way to decode the communication. It is not only about decoding the EXI messages, but also the correct parsing and interpretation of the V2GTP PDUs, i.e. header and payload.

For these reasons I needed to find some automated way to correctly parse, interpret and decode this V2GTP communication. I found several solutions, one of them is "V2Gdecoder" ([74]), which is only used to decode from a binary EXI V2G message to a readable XML document and vice versa. This tool is not suitable for the moment, subsequently I found two dissectors designed for Wireshark. The first project was last updated in mid-2021 ([75]), while the second ([76]) is still being updated. The best solution was to use the dissector for Wireshark from [76]. This is a constantly updated project and after testing it meets all my requirements. The image (Figure 2.6) shows how the dissector can recognize V2GTP. Within V2GTP it can separate SDP and V2G messages, which it can also decode and interpret (convert from an XML document to a more readable form).

Since I needed decoding in my tool and using wireshark dissector would not be possible (not easy, suitable and fast solution), I was looking for a suitable CLI tool that can be used to encode and decode V2G messages. I already mentioned a suitable tool above, it is "V2Gdecoder" ([74], which after inserting EXI encoded data (for V2G message) can decode this data and vice versa. This tool, which was developed as part of research [67]. V2Gdecoder can serve as a CLI, but also offers the possibility to run as a web service to which requests for encode or decode are sent. It is the web service version that I use in my tool to encode and decode V2G messages.

For various EXI data I tried to use "V2Gdecoder" to check if the tool is suitable and functional for my use case. Therefore, for demonstration, I attach the EXI data captured by Wireshark and decoded using "V2Gdecoder". In the picture (Figure 2.7) you can see the EXI encoded data in more detail (automatic decoding of the EXI encoded V2G messages in Wireshark is disabled). For this EXI data I used the mentioned "V2Gdecoder" and decoded it (shown in Figure 2.8). The above is only to show how "V2Gdecoder" works; as part of my research I also did this reverse of this procedure (from XML to EXI encoded V2G message).

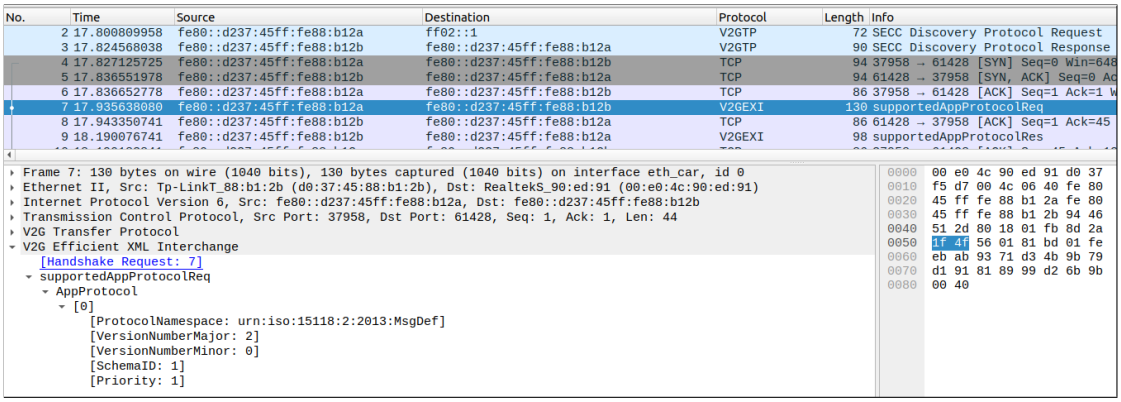


Figure 2.6 Wireshark captured commnuication – with V2GTP disector

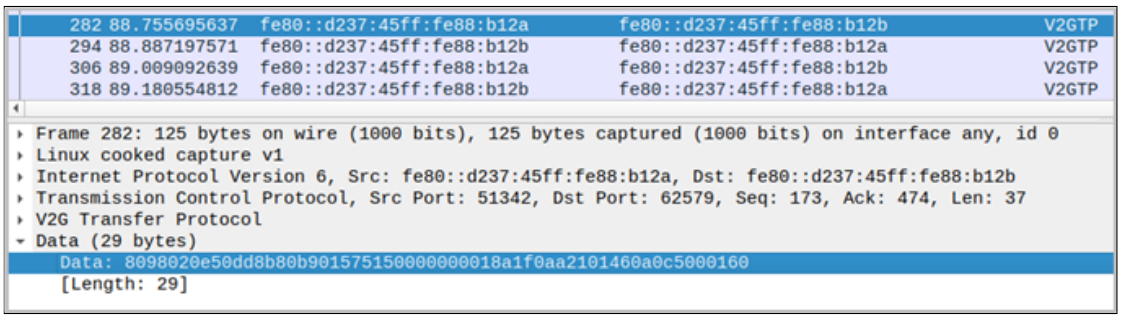


Figure 2.7 Wireshark captured commnuication – EXI encoded data

```
┌──(v2g㉿v2g-virtual-machine)-[~/V2G/repos/V2Gdecoder]
└─$ java -jar V2Gdecoder.jar -e -s '8098020e50dd8b80b90157515000000000018a1f0aa2101460a0c5000160
'
WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will impact performance.
<?xml version="1.0" encoding="UTF-8"?><ns7:V2G_Message xmlns:ns7="urn:iso:15118:2:2013:MsgDef"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns3="http://www.w3.org/2001/XMLSc
hema" xmlns:ns4="http://www.w3.org/2000/09/xmldsig#" xmlns:ns5="urn:iso:15118:2:2013:MsgBody"
xmlns:ns6="urn:iso:15118:2:2013:MsgDataTypes" xmlns:ns8="urn:iso:15118:2:2013:MsgHeader"><ns7:
Header><ns8:SessionID>3943762E02E4055D</ns8:SessionID></ns7:Header><ns7:Body><ns5:PowerDeliver
yReq><ns5:ChargeProgress>Start</ns5:ChargeProgress><ns5:SAScheduleTupleID>1</ns5:SAScheduleTup
leID><ns5:ChargingProfile><ns6:ProfileEntry><ns6:ChargingProfileEntryStart>0</ns6:ChargingProf
ileEntryStart><ns6:ChargingProfileEntryMaxPower><ns6:Multiplier>0</ns6:Multiplier><ns6:Unit>W<
/ns6:Unit><ns6:Value>11000</ns6:Value></ns6:ChargingProfileEntryMaxPower></ns6:ProfileEntry><n
s6:ProfileEntry><ns6:ChargingProfileEntryStart>86400</ns6:ChargingProfileEntryStart><ns6:Charg
ingProfileEntryMaxPower><ns6:Multiplier>0</ns6:Multiplier><ns6:Unit>W</ns6:Unit><ns6:Value>0</
ns6:Value></ns6:ChargingProfileEntryMaxPower></ns6:ProfileEntry></ns5:ChargingProfile></ns5:Po
werDeliveryReq></ns7:Body></ns7:V2G_Message>
```

Figure 2.8 V2Gdecoder – Decoded EXI V2G message

2.3 Summary

I use this V2G setup further in the implementation of the tool (i.e. in the following chapters). This chapter describes: everything needed in terms of the necessary hardware to ensure PLC communication according to ISO 15118-3; all necessary configurations to successfully enable communication between SECC and EVCC using HPGP devices; the necessary Wireshark configuration for HLC analysis; how the V2Gdecoder works, which I will use for the implementation of my tool. Thanks to this, I only build on it in the next chapter, and the things mentioned are a necessary prerequisite for the following chapters. Therefore, from a tool implementation point of view, I only focus on ISO 15118-2.

Chapter 3

Building the tool

This chapter briefly describes and explains the implementation process of the tool. The steps taken during the implementation are presented. The documentation comments mainly describe individual source files rather than this text. Therefore, this is more of an overall view of how the tool evolved. A summary of the tool's modules and features is provided at the end of this chapter.

To develop the tool, which I later named "V2GEvil", I built upon the previous V2G setup. It is important to note that testing EVCC requires the assumption that EVCC intends to initiate communication. As previously mentioned, the SECC behaves like a server. In contrast to [67], in my setup, the EVCC and the SECC are directly connected using PLC modules. Another assumption is that HPGP devices (PLC modules) have successfully established AVLNs. The SECC is under the control of my tool, which means that I am directly connected to the tested EVCC. Therefore, I do not attempt to enter any AVLN network. In this case, I use an AVLN that is automatically configured for my two HPGP devices. Another prerequisite is the use of a static IPv6 setup. This setup has already been tested and proven to work for EVCC and SECC implementations. Therefore, I do not take into consideration other IPv6 setup options and work with IPv6 addresses as specified in the V2G setup.

Python version 3.10.12 was chosen to implement V2GEvil due to its capabilities in working with packets, particularly with the Scapy library. The language is also considered programmer-friendly. In addition, many security testing tools are written in Python. Virtual environments also simplify the distribution of such a tool.

There is a similar tool ("V2Gdecoder"), but according to its github documentation and [67] it mainly focuses on sniffing, parsing and injecting v2gtp packets. It is also mentioned that the next work will be to add some pre-developed attacking functions during interception. However, these functionalities have not yet been added to the tool. Therefore, my tool is different in that it includes automatic enumeration and fuzzing. Additionally, a better possibility for configuring V2G messages and their content is implemented.

The individual sections are in chronological order as I progressed through the development of "V2GEvil". I start with the development of the V2GTP support and the sniffer functionality. Then I implemented the "station" part, where I focused on the implementation of UDP and TCP servers. The following step involved converting the message description and specification from XML Schemas to Python classes. As a consequence, additional functionality was necessary to add to "v2gtp" in the form of the "V2GTPMessage" class, which provides request processing and creation of the corresponding response for V2GTP messages (SDP, V2G message). At the point where the SECC functionality was implemented, I could focus on the testing and enumeration modules. The SECC functionality is handled by a combination of the "station", "v2gtp" and

"messages" modules. Following the operational SECC (tested against the reference EVCC), I implemented the "enumerator" module and then the "fuzzer" module. Each module has its own entry point for the CLI. The functionality of a CLI module is always available through its "cli tools", for example, the "station" module has "station-tools" in the "cli" module. Individual modules for security testing are accessible via "modules-tools". The exact structure can be found in the source code. A more detailed description of the individual logical parts of the implementation is given in the following sections.

A summary of "V2GEvil" of what is implemented in the tool is provided at the end of this chapter. A description of the main building blocks of the tool is given in the individual sections of this chapter.

3.1 V2GTP

The initial step was to investigate the communication between the SECC and the EVCC. This was done by using Wireshark and studying [34]. The results of this research were then used in my implementation of V2GTP in the "v2gtp" module.

In the module I first implemented methods that were responsible for parsing packets. The implemented methods were designed to detect: whether the packet is a TPC/UDP packet; whether the packet has an IPv6 layer; whether the packet is a V2GTP packet. For a V2GTP packet, the implemented methods are used to recognize the payload type, i.e. whether the payload is an SDP request, SDP response or V2G message. The SDP request and SDP response decoding is also implemented. The module includes the implementation of V2G EXI message decoding using V2Gdecoder as a web service.

The original version of this module, which was later extended (see below), can be used standalone with the "v2gtp-tools" command for "V2GEvil". The user can access two functionalities, decode and extract, through "v2gtp-tools". The extract command allows to extract V2GTP packets from the provided pcap file. The "decode" command provides decoding of a V2GTP packet from the provided pcap file, the desired packet is defined by the packet number (within the pcap file). The decoding functionality for the V2G EXI message requires a running web service called V2Gdecoder. An example of decoding is shown in the sniffer module, which uses this functionality for live decoding.

The figure (Figure 3.1) shows a sample call and output of the 'v2gtp-tools' for the 'extract' command. The figure's output is truncated due to its size.

Thanks to testing the above functionalities, I have verified that all implemented methods work as expected. The verification was conducted by comparing the module's output with that of the Wireshark dissector. Furthermore, comparing with [34], if the messages make sense and if all V2GTP specifications (in the module) are conforming to the standard.

In the later phase, the creation of responses to requests from EVCC was added to this module for the "station" module. For this purpose "v2gtp" uses the "messages" module, in which V2G messages and their structure according to [34] are defined. The creation of V2TP packets is handled by class "V2GTPMessage" in this module. This class contains the necessary functionality to parse V2GTP messages and create responses (for both SDP and V2G messages). The class uses the "messages" module. This class is then used in the "station" module: to parse SPD request/response and V2G EXI messages; decode V2G EXI messages; create responses. In a more simplified way, it handles everything within V2GTP.

In addition, it should be noted that the implementation of this class was dependent on the implementation of the station module because an object approach was better to use than a functional one. The above-described functional approach was adequate for the basic implementation of V2GTP and its use in the sniffer module.

The best way to verify the implementation of V2GTP for live communication was to use it for the sniffer module, which is discussed in the next section. It is worth noting that during

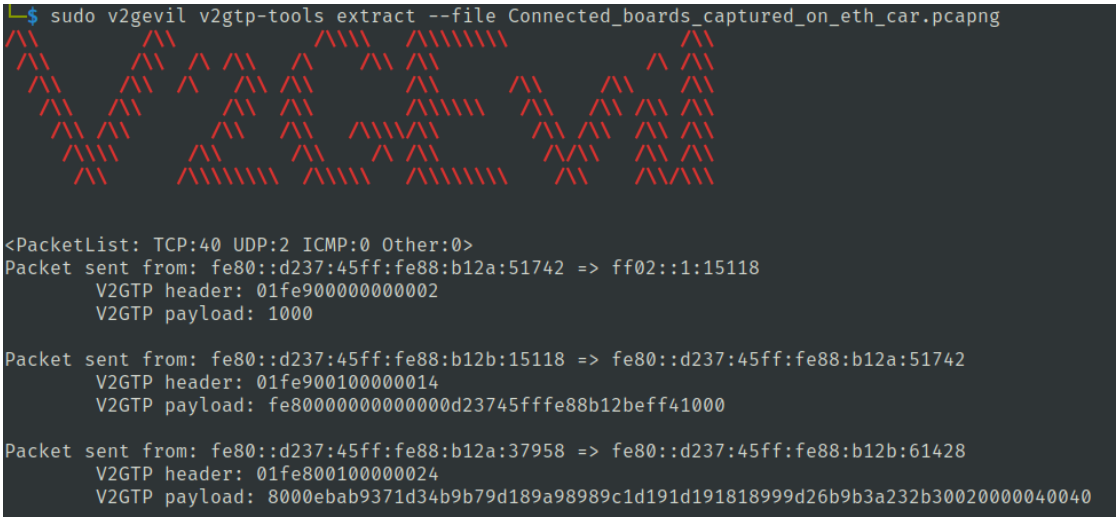


Figure 3.1 V2GEvil – v2gtp-tools extract example

the live communication testing, V2GTPMessage class was not yet implemented. Therefore, a functional approach was used that involved utilising all the currently implemented methods.

3.2 Sniffer

This module was added to the tool to test whether V2G communication can be intercepted, decoded, and interpreted. The sniffer module therefore adds additional functionality in the form of sniffing live communication with decoding. The output and functionality can be customized depending on the arguments provided via cli.

The sniffer module includes methods for both static sniffing (i.e. pcap file analysis) and live sniffing. Therefore, the functionality available through 'v2gtp-tools' for static analysis/decoding is integrated and extended within this module. The reason for this is to keep the corresponding functions within the same module. So, the following functions are defined in the sniffer module: sniff, analyze, live_sniff, inspect.

The "sniff" and "inspect" methods are accessible to the user via the CLI within the "sniffer-tools" command (shown in the Figure 3.2 figure). The "sniff" method differentiates between a live sniff and a pcap file. To work with a live sniff, call the "live_sniff" method, and to work with a pcap file, call the "analyze" method. The "inspect" method handles the inspection of a particular packet, identified by the packet number in the pcap file. Details for the specific command of my tool are available via the "–help" option. The specifics of a particular implementation are in the source code.

In the sniffer module I use the methods implemented in the "v2gtp" module. I also use the Scapy library, for example, to load a pcap file or the scapy sniff method. Live sniffing is performed using the Scapy library's sniff method. The scapy sniff method processes the packet and prints the output based on a function passed through the "prn" parameter. The passed function is implemented in the v2gtp module

An example of how live sniff with decoding works is shown in the video: "2gevil_live_decode_flag.mp4". An example of a live sniff with V2GTP recognition and splitting into header and payload can be found in the video titled "v2gevil_live_sniff_v2gtp_flag.mp4". Only a partial output of the tool is illustrated here (Figure 3.3).

By testing with the "v2gtp" a "sniff" modules, I have verified the proper functionality of my implementation of V2GTP, and that it can be used as a base for the SECC (station module)

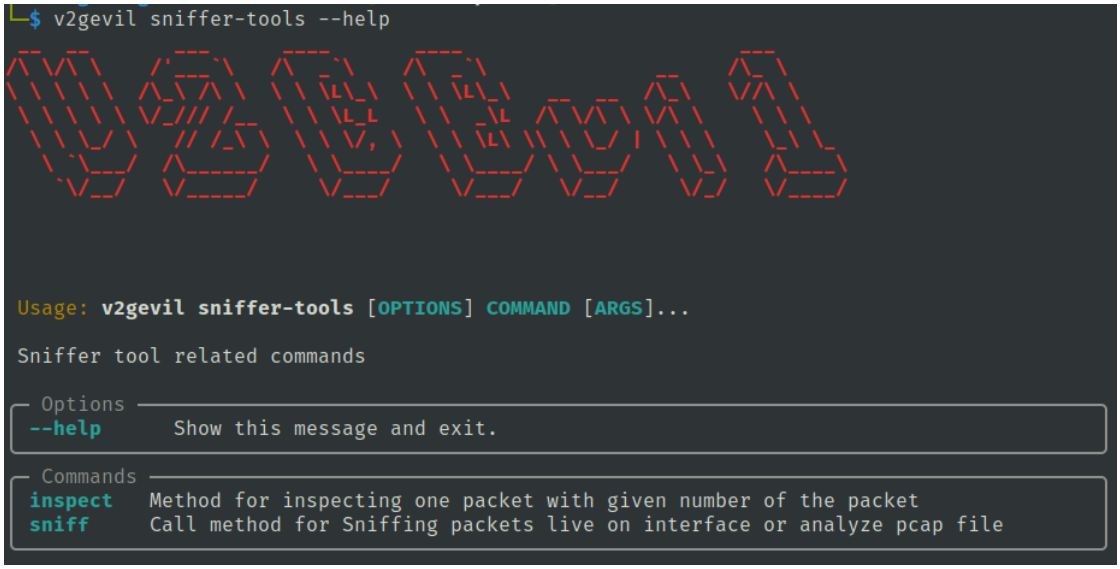


Figure 3.2 V2GEvil – sniffer-tools

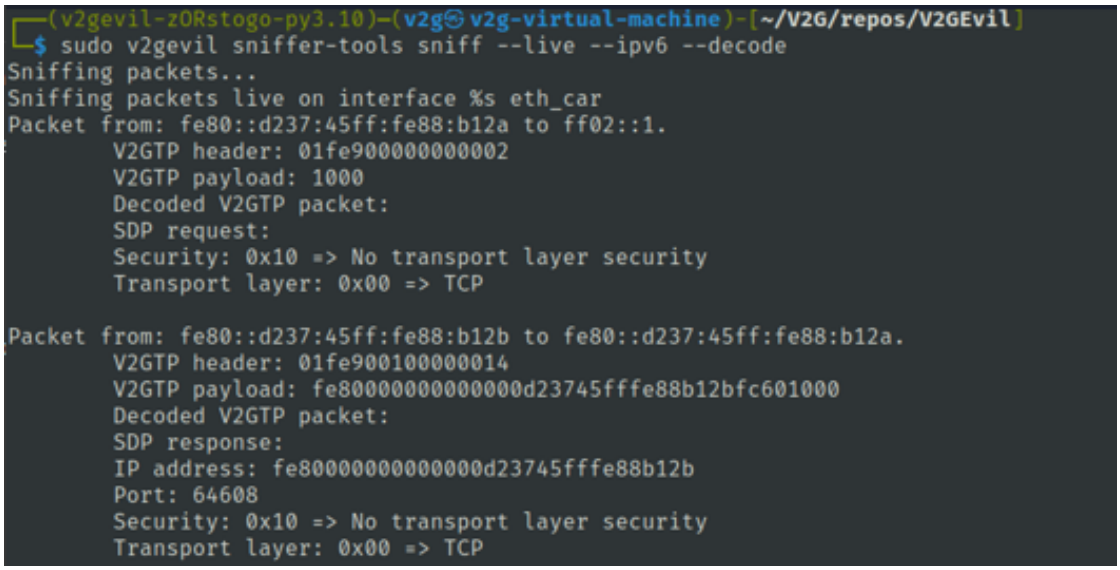


Figure 3.3 V2GEvil – sniffer with SDP request/response decoded

implementation. The sniffer module can be used as an alternative to Wireshark for monitoring V2G communication. This eliminates the need to search for Wireshark dissectors and configure Wireshark. It is important to note that TLS is not currently being used for communication, as it was necessary to establish communication between EVCC and SECC first. TLS support was added only after successful exchange of V2G messages (after I implemented SECC).

3.3 Station

This section includes the main module named "station", which mainly uses the "v2gtp" and "messages" modules. The "messages" module is described later in this section, as it is closely related to the SECC functionality. The core of SECC is implemented within station module. The implementation process is described first, followed by a brief summary of how this module works.

Considering the information about SECC implementation that comes from [34], I decided to use "asyncio" library to manage UDP and TCP servers. In this module I implemented "ServerManager" class that contains asynchronous methods.

The communication between EVCC and SECC starts through the SECC discovery protocol (part of V2GTP) and UDP is used for this purpose. The first step in the implementation of the station was a UDP server that supports SDP requests. This server sends a proper response to the SDP request according to [34]. I used the "socket" library to implement the UDP server. It was important to set the server to listen on the IPv6 multicast address on port 15118, according to [34].

Additionally, I added the option to set TLS to the UPD server implementation. This setting concerns whether the TLS byte should correspond to the TLS byte received in the SDP request. Alternatively, whether TLS or plain TCP (no TLS) should be enforced using SDP response. The "V2GTPMessage" class from the "v2gtp" module is used to create the SDP response. Later, during the development of the EVCC evaluation modules, I added a check for the "Enumerator" module to the UDP server implementation.

The EVCC attempts to connect to a TCP server based on the SDP response, requiring the implementation of such a server. The socket library was used to gain control over the interface, IPv6 link local address, and the port on which the TCP server listens. The chosen port is from the range specified in [34]. If it was agreed within SDP that the connection should be encrypted using TLS, then SECC uses a TLS connection instead of plain TCP. The communication will be encrypted using TLS with the help of the "ssl" library. This library enables the use of the same "server_socket" for both plain TCP and TLS connections. To establish a TLS connection, only this socket needs to be wrapped with "wrap_socket" using SSLContext. Once the TCP or TLS connection has been successfully established, V2G communication is managed through the use of the "v2gtp_comm_handler" method.

The method uses the established connection to process V2G communication and receives and sends data. The received V2GTP message is processed using the "V2GTPMessage" class from the "v2gtp" module. A representation of the request is created using an instance of the "V2GTPMessage" class for the received bytes. This instance includes a method for generating a response to the message, correctly associating the request with the requested response. The most important method from this perspective is "create_v2gtp_exi_msg_response", which uses "message_dict" and methods and classes from the "messages" module.

The mentioned dictionary contains the names of V2G requests and their corresponding V2G response names, along with the response data. This dictionary is passed to the constructor of the "ServerManager" class, if it is not passed, then the default dictionary is loaded. The dictionary can be generated by the user using the generator, or the user can edit the dictionary file and set custom values for various parameters.

The dictionary solution seems to be the most ideal way, because live data regarding charging cannot be obtained otherwise, because it is not a real charging station (it does not supply any

Code listing 1 Example of default_dict_AC.json (truncated part of dictionary)

```
{
   "supportedAppProtocolReq":{
      "supportedAppProtocolRes":{
         "ResponseCode":"OK_SuccessfulNegotiation",
         "SchemaID":"1"
      }
   },
   "SessionSetupReq":{
      "SessionSetupRes":{
         "ResponseCode":"OK",
         "EVSEID":"FRA23E45B78C",
         "EVSETimeStamp":1700593914
      }
   },
   "ServiceDiscoveryReq":{
      "ServiceDiscoveryRes":{
         "ResponseCode":"OK",
         "PaymentOptionList":{
            "PaymentOption":[
               "Contract",
               "ExternalPayment"
            ]
         }
      }
   }
}
```

current). To illustrate, I attach a sample of a part of the dictionary (Code listing 1). The complete dictionary can be found in the "default_dictionaries" folder in the "messages" module. There is also a runtime option to generate a response to "supportedAppProtocolReq", if the message is not in the dictionary, then the runtime response is used. This response identifies the same protocol namespace in the request defined in "v2gtp_enums".

This leads me to the "messages" module, which implements everything related to V2G EXI messages. That contains following: default dictionaries (AC and DC) for the "station" module; dictionary generator; parsing and conversion from XML to Python dict and vice versa; conversion from XML to EXI (using V2Gdecoder) and vice versa; conversion from Python dict to class instance and vice versa; conversion from XML to class instance and vice versa.

For the messages module it is most important to mention that this was one of the longest processes in the implementation. This is because it was necessary to convert all the V2G message definitions from XML Schema (from [34]) to Python classes. I used the same file splitting as the standard XML Schema splitting, i.e. "MsgDef.py", "MsgHeader.py", "MsgBody.py", "MsgDataTypes.py", "AppProtocol.py".

To create an instance from a dictionary, I sought a suitable solution and found the 'pydantic' library to be the most appropriate for my needs. It also allows to convert a class instance to a dictionary, which I used for example when converting a V2G message instance to an EXI V2G message. This conversion specifically: instance => dict data => xml data => EXI data. Another reason why I use "pydantic" is that this library allows to define schemas via Pydantic models (more in [77]). An example of the use of this library and therefore the definition of a V2G message is shown on Code listing 2. It is also shown in the "MsgBody.py" code snippet,

Code listing 2 Example of pydantic use in MsgDef.py

```
"""
Module for V2G message definition
"""
from pydantic import BaseModel, Field, ConfigDict
from .MsgHeader import Header
from .MsgBody import Body


class V2G_Message(BaseModel):
    """Base class for V2G messages.

    Attributes:
        header: The header of the V2G message.
        body: The body of the V2G message.
    """

    header: Header = Field(..., alias="Header")
    body: Body = Field(..., alias="Body")
```

specifically ServiceDiscoveryReq/Res is shown on Code listing 3.

After implementing all XML schemas and necessary conversions as mentioned above, the "messages" module can be used successfully.

That concludes the description of the most important part of the "station" module implementation and its relevant modules. Information about the tool and its individual modules mentioned in the description can be found in Usage of the tool.

3.3.1 How it works – summary

The "station" module serves as an implementation of SECC, i.e. it contains a UDP server, a TCP server and supports TLS. It also contains "v2gtp_comm_handler", which is used to process V2G messages within a V2G communication session. This handler processes the request and sends the corresponding response. The "ServerManager" class comprises all of these parts.

This class defines the "start" method which initiates the server. All communication-related methods are defined as "async". The "asyncio" library was used for starting UDP and TCP servers and for switching control between them. For example, because the UDP server is no longer needed after the TCP connection is established, and the UDP server should be available again after the TCP connection is terminated. Further details on TCP and UDP servers can be found in [34]. Therefore, I use the "asyncio" library and its methods to ensure asynchronous server operation.

The class "ServerManager" instance is created after calling the "start_async" method. This method passes the necessary configuration parameters and then calls the asynchronous method of the instance (called "start") using the "asyncio" library and the "run" method. The UDP server is then started on port 15118 and waits for SDP requests. In addition, a TCP server runs asynchronously waiting for TCP connections. If the UDP server receives an SDP request then it responds with information about the TCP server (IP, port, …). If the EVCC establishes a connection with the TCP server, the UDP server does not continue to run. V2G Communication between EVCC and SECC within the TCP connection is handled by "v2gtp_comm_handler". After the V2G communication is terminated, the UDP server is available again.

Code listing 3 Example of pydantic use in MsgBody.py – ServiceDiscoveryReq/Res

```
class ServiceDiscoveryReq(BodyBaseType):
    """Representation of V2GMessage ServiceDiscoveryReq."""

    # serviceScopeType, xs:string, maxLength 64, minOccurs 0 => not required
    service_scope: str = Field(default=None, alias="ServiceScope")
    # ServiceCategoryType, xs:string, enum values, minOccurs 0 => not required
    service_category: serviceCategoryType = Field(
        default=None, alias="ServiceCategory"
    )


class ServiceDiscoveryRes(BodyBaseType):
    """Representation of V2GMessage ServiceDiscoveryRes."""

    # responseCodeType, enum values, minOccurs 1 => required
    response_code: responseCodeType = Field(..., alias="ResponseCode")
    # PayementOptionListType, minOccurs 1, maxOccurs 1
    payement_option_list: PaymentOptionListType = Field(
        ..., alias="PaymentOptionList"
    )
    # ChargeServiceType, minOccurs 1 => required
    charge_service: ChargeServiceType = Field(..., alias="ChargeService")
    # ServiceListType, minOccurs 0 => not required
    service_list: ServiceListType = Field(default=None, alias="ServiceList")
```

Starting SECC (station) is supported by the mentioned “start_async” method, which serves as “entry point”. This method is used in “station-tools”, which allows user to start station via CLI. The method is also used in other modules (fuzzer and enumerator). These modules use working SECC implementation and only pass certain parameters that modify the behaviour of the station.

Thanks to this, EVCC testing is enabled using the “station” module and so it is not necessary to reimplement SECC logic in test modules ((fuzzer and enumerator). During module implementation, only minor modifications were required to the SECC source code to enable easy use of the modules as described above.

This briefly describes how the station module works in my tool.

3.4 Enumerator

This module is used for enumeration of EVCC. To use this module with the CLI, “modules-tools” is used. All modules used for testing (enumeration, evaluation, etc.) are available via the CLI via “modules-tools”. This is to provide better clarity on which modules are relevant for EVCC security testing and which are not (also for future modules).

The most important thing was to cleverly incorporate the use of this module into my existing implementation of “station”. Therefore, I chose to pass an instance of the “EVEnumerator” class to the method that runs SECC as a suitable solution. This instance is created before the start of the SECC (station) and after its end (even forcibly), depending on the enumeration mode selected, the information obtained (stored in the instance) is printed to the user.

Creating a class instance and then calling “station.start_async” is in the “enumerate_ev” method (in the “enumerate_ev.py” file), which is called from “modules_tools” for “enumerate-EV”. For a better understanding I attach the source code of this method (Code listing 4), shown without comments due to length. However, the main logic for the enumerator is in the “EVEnumerator” class, for which it is not appropriate to include a code listing due to the number of lines.

The enumerator implementation is modular and it is easy to add additional methods for enumeration.

The actual implementation of the “EVEnumerator” class is in the “enumerator.py” file. I implemented functionality for the class that allows: enumeration of supported EVCC protocols; checking whether TLS is required or not; checking the TLS version used. The enumerator implementation is modular and it is easy to add additional methods for enumeration.

As part of the implementation of this module I had to slightly modify the implementation of the “station” module. Specifically the UDP server, TCP server and “v2gtp_comm_handler”. This modification consisted of adding checks to see if an EVEnumerator instance is passed. If the instance has been passed, then the required data is stored in the enumerator instance based on the enumerator mode and possibly the properties of the instance variables. For example, the “msgs_for_enum” class variable defines which requests should be added to the enumerator instance.

All information is stored in the EVEnumerator instance and is printed out to the user after the end of the station run. A sample usage is explained in the chapter Usage of the tool, where examples of the tool output will be included.

3.5 Fuzzer

Another part of “modules-tools” is the “fuzzer” module. This module is used to test the processing of V2G messages in EVCC. According to various settings (see below), it sends invalid or malicious data in responses to V2G requests. I will first briefly describe how the module works and then focus more on the steps during implementation.

Code listing 4 V2GEvil – enumerate_ev method

```
def enumerate_ev(
    interface: str = EVSEDetails.INTERFACE.value,
    accept_security: bool = True,
    enum_mode: EVEnumMode = EVEnumMode.ALL,
):

    ev_enumerator = EVEnumerator()

    tls_flag = False
    match enum_mode:
        case None:
            print("EVSE is not connected")
            return
        case EVEnumMode.ALL:
            ev_enumerator.add_all()
            tls_flag = True
        case EVEnumMode.SUPPORTED_PROTOCOLS:
            ev_enumerator.add_supported_protocols_check()
        case EVEnumMode.TLS_CHECK:
            ev_enumerator.add_tls_check()
        case EVEnumMode.TLS_ENUM:
            ev_enumerator.add_tls_enum()
            tls_flag = True


    station.start_async(
        interface=interface,
        accept_security=accept_security,
        ev_enumerator=ev_enumerator,
        tls_flag=tls_flag,
    )

    match enum_mode:
        case EVEnumMode.ALL:
            ev_enumerator.print_all()
            return
        case EVEnumMode.SUPPORTED_PROTOCOLS:
            ev_enumerator.print_supported_protocols()
            print("EVSE is not connected")
            return
        case EVEnumMode.TLS_CHECK:
            ev_enumerator.print_tls_check_result()
            return
        case EVEnumMode.TLS_ENUM:
            ev_enumerator.print_tls_enum_result()
            return
```

As I am testing the client-side and not the server-side, this fuzzer is somewhat specific. For the "station" module, the response generation for the corresponding request is facilitated by the dictionary (Code listing 1). The basic principle of working with the dictionary has already been described in Station. The Fuzzer module modifies the mentioned dictionary according to the CLI arguments or the configuration file and passes them to the "station.start_async" method, which starts the station.

Fuzzer generally generates a default dictionary (or copies an existing one) and then modifies it. This modification consists of modifying and filling it with malicious and non-valid data for certain V2G responses. When the dictionary modification is successfully completed, the SECC (station) is run with that dictionary. Subsequently, when the client sends a request, a corresponding response is sent for the request with a non-valid response. Errors on the EVCC side are hard to detect on the SECC side in general and thus within this tool. The best way in the real case is to connect via CAN to the CCU handling the charging process and use CAN messages to detect the status. If this is not enabled, then the indication to the tool user (tester) may be that the communication ends unexpectedly. This is simplified functionality of this fuzzer.

This module has been modified several times, so I only mention the steps that led to the final implementation. Again, as in the case of enumerator, I tried for modularity and a simple solution, i.e. not to interfere with the existing implementation of the "station" module. This was possible by the previous design of the response generation, i.e. the use of a dictionary for this purpose. The fuzzer logic can be executed in its entirety before starting the station, and only then passing the resulting product (dictionary) to the station. The station operates with a dictionary that may contain non-valid data in some or all of the V2G responses. This is used to evaluate the EVCC implementation.

This module, as well as the "enumerator" module, is accessible from the CLI via "modules-tools" and the "fuzz-EV" command. I implemented the main logic for EV fuzzing in the "EVFuzzer" class in the "fuzz.py" file. Within "modules-tools" this class is used to create a class instance and then call the "fuzz" method (from "EVFuzzer" class). In this method I implemented a mode-based selection for fuzzing. I have attached a sample of the "fuzz" method (Code listing 5).

Based on the mode specified by the user via the CLI, the corresponding method is called. This method performs dictionary generation for fuzzing. After the dictionary generation method finishes, the "fuzz" method passes the dictionary to "station.start_async", which starts the SECC.

I have implemented 4 fuzzing modes. The first mode "ALL" performs fuzzing for all possible parameters of all messages, and then stores them in the dictionary (fuzzing_dict). This mode does not use a config file, it just calls the fuzzing methods for each message. Within each fuzzing method, fuzzing is performed for all parameters, and the values are chosen randomly. I attach a sample implementation of the method that is called for the "ALL" mode (Code listing 6).

For the "MESSAGE" mode, a method is called to perform fuzzing exclusively on the designated message. The fuzzing of specific paremeters of a given message is based on the configuration file for the fuzzer. Only the configuration information for the selected message is extracted from the configuration file. Based on the configuration information, fuzzing of this message is performed. The user can specify in the configuration file which parameters will be fuzzed and how. The default configuration file is located in the "config" folder in the "fuzzer" module and is named "ev_fuzzer_config_default.toml". The configuration file is loaded if the fuzzing method uses it, which is the case here. A demonstration of the fuzzing method that is called for "MESSAGE" mode is in Code listing 7. The "CUSTOM" mode only calls a method that, based on the provided filename (path), reads and uses this file as "fuzzing_dict". This dictionary should use the same format as the default dictionary for the "station" module. The responsibility is on the user; if some messages are missing then SECC may not send a response to some requests.

Last, I implemented a mode that is based only on the configuration file provided for the fuzzer, this mode is called "CONFIG". This mode is handled by the "fuzz_config_based" method and its

Code listing 5 V2GEvil – fuzzer core logic, fuzz method

```
def fuzz(self, message_name: str = ""):
    match self.mode:
        case EVFuzzMode.ALL:
            self.fuzz_all()
        case EVFuzzMode.CUSTOM:
            self.fuzz_custom()
        case EVFuzzMode.MESSAGE:
            if message_name == "":
                logger.error(
                    "Message name is not set."
                    "Cannot fuzz only one message."
                )
                raise ValueError(
                    "Message name is not set."
                    "Cannot fuzz only one message."
                )
            self.fuzz_message(message_name=message_name)
        case EVFuzzMode.CONFIG:
            self.fuzz_config_based()
        case _:
            logger.error("Invalid fuzzing mode: %s", self.mode)
            raise ValueError(f"Invalid fuzzing mode: {self.mode}")

    station.start_async(
        interface=self.interface,
        charging_mode=self.charging_mode,
        req_res_map=self.fuzzing_dict,
        validate=False,
    )
```

Code listing 6 V2GEvil – fuzzer, fuzz_all method

```
def fuzz_all(self):

    # If None value is set for params, all possible params will be fuzzed
    self.fuzz_supported_app_protocol_res(msg_config=None)
    self.fuzz_session_setup_res(msg_config=None)
    self.fuzz_service_discovery_res(msg_config=None)
    self.fuzz_service_detail_res(msg_config=None)
    self.fuzz_payment_service_selection_res(msg_config=None)
    self.fuzz_payment_details_res(msg_config=None)
    self.fuzz_authorization_res(msg_config=None)
    self.fuzz_charge_parameter_discovery_res(msg_config=None)
    self.fuzz_power_delivery_res(msg_config=None)
    self.fuzz_metering_receipt_res(msg_config=None)
    self.fuzz_session_stop_res(msg_config=None)
    self.fuzz_certificate_update_res(msg_config=None)
    self.fuzz_certificate_installation_res(msg_config=None)

    # Differ between charging modes
    if self.charging_mode == EVSEChargingMode.AC:
        self.fuzz_charging_status_res(msg_config=None)
    elif self.charging_mode == EVSEChargingMode.DC:
        self.fuzz_cable_check_res(msg_config=None)
        self.fuzz_pre_charge_res(msg_config=None)
        self.fuzz_current_demand_res(msg_config=None)
        self.fuzz_welding_detection_res(msg_config=None)
    else:
        # Should never happen
        logger.error("Invalid charging mode: %s", self.charging_mode)
        raise ValueError(f"Invalid charging mode: {self.charging_mode}")
```

Code listing 7 V2GEvil – fuzzer, fuzz_message method

```
def fuzz_message(self, message_name: str):

    # Need conversion to enum, because user input is string
    try:
        message_name = MessageName(message_name)
    except ValueError as exc:
        logger.error("Invalid message name: %s", message_name)
        raise ValueError(f"Invalid message name: {message_name}") from exc

    # Load fuzzer config file in binary mode
    with open(self.config_filename, "rb") as config_file:
        config_data = tomli.load(config_file)

    # Check if message name is in config file
    if message_name not in config_data.keys():
        logger.error(
            "Message name: %s is not in fuzzer config file.", message_name
        )
        logger.error("Fuzzer config file: %s", self.config_filename)
        raise ValueError(
            f"Message name: {message_name} is not in fuzzer config file"
        )

    # Pick only config for specified message
    msg_config = config_data[message_name]

    # Differ in "match case", which fuzz method will be called
    # based on the message name
    match message_name:
        case MessageName.SUPPORTED_APP_PROTOCOL_RES:
            self.fuzz_supported_app_protocol_res(msg_config=msg_config)
    # shortened part of the code, other cases of message names continue.
```

Code listing 8 V2GEvil – fuzzer, fuzz_config_based method

```
def fuzz_config_based(self):

    # Load config file in binary mode
    with open(self.config_filename, "rb") as config_file:
        config_data = tomli.load(config_file)

    # Get all messages names from config file, which should be fuzzed
    message_names = config_data.keys()

    # Pairs message name and fuzzing method
    # NOTE: some lines had to be split into multiple lines
    #   because of space in the code listing.
    pairs_message_fuzz_method = {
        MessageName.SUPPORTED_APP_PROTOCOL_RES:
            self.fuzz_supported_app_protocol_res,
        MessageName.SESSION_SETUP_RES: self.fuzz_session_setup_res,
        MessageName.SERVICE_DISCOVERY_RES: self.fuzz_service_discovery_res,
        MessageName.SERVICE_DETAIL_RES: self.fuzz_service_detail_res,
        MessageName.PAYMENT_SERVICE_SELECTION_RES:
            self.fuzz_payment_service_selection_res,
        MessageName.PAYMENT_DETAILS_RES: self.fuzz_payment_details_res,
        MessageName.AUTHORIZATION_RES: self.fuzz_authorization_res,
        MessageName.CHARGE_PARAMETER_DISCOVERY_RES:
            self.fuzz_charge_parameter_discovery_res,
        MessageName.POWER_DELIVERY_RES: self.fuzz_power_delivery_res,
        MessageName.METERING_RECEIPT_RES: self.fuzz_metering_receipt_res,
        MessageName.SESSION_STOP_RES: self.fuzz_session_stop_res,
        MessageName.CERTIFICATE_UPDATE_RES: self.fuzz_certificate_update_res,
        MessageName.CERTIFICATE_INSTALLATION_RES:
            self.fuzz_certificate_installation_res,
        MessageName.CHARGING_STATUS_RES: self.fuzz_charging_status_res,
        MessageName.CABLE_CHECK_RES: self.fuzz_cable_check_res,
        MessageName.PRE_CHARGE_RES: self.fuzz_pre_charge_res,
        MessageName.CURRENT_DEMAND_RES: self.fuzz_current_demand_res,
        MessageName.WELDING_DETECTION_RES: self.fuzz_welding_detection_res,
    }

    # Iterate over all message names in config file and
    # call appropriate fuzzing method
    for name in message_names:
        # Check if message name is valid
        try:
            name = MessageName(name)
        except ValueError as exc:
            logger.error("Invalid message name: %s in fuzzer config", name)
            raise ValueError(f"Invalid message name: {name}") from exc
        # Call appropriate fuzzing method
        pairs_message_fuzz_method[name](msg_config=config_data[name])
```

Code listing 9 V2GEvil – fuzzer, fuzz_service_discovery_res method

```
def fuzz_service_discovery_res(self, msg_config: Optional[dict] = None):

    req_key = MessageName.SERVICE_DISCOVERY_REQ
    res_key = MessageName.SERVICE_DISCOVERY_RES

    msg_dict_to_fuzz = self.fuzzing_dict[req_key][res_key]
    # Keep default values for all params in the message
    msg_default_dict = self.default_dict[req_key][res_key]

    msg_fuzzer = FuzzerServiceDiscoveryRes(
        msg_config=msg_config,
        msg_fuzz_dict=msg_dict_to_fuzz,
        msg_default_dict=msg_default_dict,
    )

    # Replace message in fuzzing_dict with fuzzed one (msg_dict_to_fuzz)
    self.fuzzing_dict[req_key][res_key] = msg_fuzzer.fuzz()
```

implementation is in Code listing 8. In this case, only messages located in the fuzzer configuration file will be fuzzed. The configuration file is in toml format. For each message in this file, there are specified parameters to be fuzz. Furthermore, for each parameter "Mode" is specified, that is, the way the parameter will be fuzzed.

I explain which methods are used for fuzzing a particular message by using the "ServiceDiscoveryRes" message as an example. This message contains parameters of complex data types and therefore all types of implemented methods can be shown in it. I assume that the message mentioned is defined in the configuration file and should be fuzzed. For each message I implemented its own fuzz method within the "EVFuzzer" class. The method for a specific message is in Code listing 9. This method creates an instance of the class that is intended as a fuzzer for this specific message. Then fuzzing of this message is done using class method "fuzz" and the result is stored in "fuzzing_dict".

Next, I briefly introduce the fuzzer class for a specific message, especially the "fuzz" class method (Code listing 10). The method must specify all parameters for the message. Specifically, define which parameters are mandatory according to [34]. For all these parameters it was necessary to specify which method should be used for fuzzing the parameter ("pairs_name_method"). This is necessary because I use similar class methods (fuzz) for all messages. One general method for fuzzing messages is used in these class methods ("general_msg_fuzzing_method"). This method is driven by "msg_config" and calls methods based on "pairs_name_methods" for fuzzing individual parameters. The source code of this method is not shown here due to its length.

Due to the use of the same data types (defined in XML Schema in [34] in multiple V2G messages, it was appropriate to implement a custom fuzzer method for each data type. There can be simple data type parameters or complex data type parameters in a message. Complex data types contain additional parameters (of different data types – simple/complex). Within complex data types, "general_datatype_fuzzing_methods" is always called. An example of a complex parameter is "PaymentOptionList", which contains additional parameters, and "fuzz_payment_option_list" is used as the fuzz method (shown in Code listing 11).

The "fuzz_payment_option_list" method uses the general method for data types. In this method, fuzz methods are called again for specific parameters (according to the dictionary

Code listing 10 V2GEvil – fuzzer, fuzz method in class FuzzerServiceDiscoveryRes

```
def fuzz(
        self,
    ) -> dict:
    """Fuzz the message"""

    # Pairs of parameter/field name and fuzzing method
    pairs_name_method = {
        "ResponseCode": fuzz_response_code,
        "PaymentOptionList": fuzz_payment_option_list,
        "ChargeService": fuzz_charge_service,
        "ServiceList": fuzz_service_list,
    }
    # Required fields define in the standard
    required_fields = ["ResponseCode","PaymentOptionList","ChargeService"]

    # All possible fields (required/optional) define in the standard
    all_fields = list(pairs_name_method.keys())

    return general_msg_fuzzing_method(
        required_fields=required_fields,
        all_fields=all_fields,
        msg_config=self.msg_config,
        msg_fuzz_dict=self.msg_fuzz_dict,
        msg_default_dict=self.msg_default_dict,
        pairs_name_method=pairs_name_method,
        class_name=self.__class__.__name__,
    )
```

Code listing 11 V2GEvil – fuzzer, fuzz_payment_option_list method

```
def fuzz_payment_option_list(
    attr_conf: Optional[dict] = None, valid_values: Optional[dict] = None
) -> dict:
    """Fuzz payment option list"""

    # This is not end parameter,
    # so there is passing None/{} to each sub parameter

    pairs_name_method = {"PaymentOption": fuzz_payment_option}
    required_fields = ["PaymentOption"]
    all_fields = ["PaymentOption"]

    res_dict = {}
    # Call general method for fuzzing complexType
    res_dict = general_datatype_fuzzing_method(
        required_fields=required_fields,
        all_fields=all_fields,
        attr_conf=attr_conf,
        valid_values=valid_values,
        pairs_name_method=pairs_name_method,
    )

    # PaymentOption is not list, but it is in PaymentOptionListType
    res_dict["PaymentOption"] = [res_dict["PaymentOption"]]

    return res_dict
```

Code listing 12 V2GEvil – fuzzer, fuzz_payment_option method

```
def fuzz_payment_option(
    attr_conf: Optional[dict] = None, valid_values: Optional[str] = None
):

    return fuzz_general_enum_type(
        attr_conf=attr_conf,
        valid_values=valid_values,
        enum_list=list(paymentOptionType),
    )
```

"pairs_name_methods"). If it is a simple data type parameter, then the general method for the specific simple data type is called in its fuzz method.

For example, in this case, a complex parameter of type "PaymentOptionList" is defined to contain a list of simple data type "PaymentOption". The "PaymentOption" data type is an enum, so the fuzzing method calls the general method for fuzzing enum data type and passes it the necessary parameters. This is shown in Code listing 12. All methods for fuzzing data types, whether complex or simple, are contained in a file named "fuzz_datatypes.py" within the "fuzzer" module.

In simple terms, the "CONFIG" fuzzer implementation allows the user to specify: what messages will be fuzzed; what parameters will be fuzzed; what mode will be used for each parameter; whether or not to fuzz mandatory message parameters. More specific information on how to use the configuration file is in the Usage of the tool chapter.

This is a summary of the most important implementation details. The "EVFuzzer" class was implemented to provide the main logic for the fuzzer and call the appropriate methods based on the user's selected modes. Within the 'EVFuzzer' class, I have implemented a method for each V2G response message. In the case of fuzzing a given message, this method utilises fuzzer classes that are specific to each V2G response.

To enhance modularity and clarity, each V2G response has its own fuzzing class and method. These methods use a general method for fuzzing messages ("general_msg_fuzzing_method"). This generic method performs fuzzing of a given message and follows the passed arguments that are specific to each message. The message contains different parameters and therefore I have implemented its own fuzzing method for each parameter. These methods again use general fuzzing methods, this time they are general methods for data type based fuzzing. For all complex data types, "general_datatype_fuzzing_method" is used. For simple data types, there are fuzzing methods for each type, such as for enum ("fuzz_general_enum_type") or long ("gen_invalid_long"). The important thing for these general fuzzing methods is passing the correct arguments to modify the behavior of the methods so that the output matches the specific parameter.

3.6 Summary

During the development of the tool I implemented the following modules. The "v2gtp" module, which contains everything necessary to support V2GTP.

The V2GTP module has the following characteristics:

- is used for parsing V2GTP PDU,
- is used to check packets – if they are relevant for V2GTP,

- allows extracting V2GTP packets from the pcap file„
- is used for decoding V2G EXI messages (using V2Gdecoder),
- is used to interpret SDP request and response,
- allows the creation of V2GTP messages,
- in combinations with the “messages” module allows creating V2GTP response for received requests.

Another module “cli” is used to define the interface for the tool. It allows access to the individual functionalities of the V2GEvil tool.

The “sniffer” module allows live sniffing of V2G communication and its reading from the pcap file. It offers various options to filter and decode this communication using the “v2gtp” module.

An important module to allow my tool to communicate within V2GTP is a module called “messages”. This module implements XML schemas (from [34]) by Python class using pydantic models. Therefore, the module contains complete definitions of V2G messages (request, response). Furthermore, this module implements the conversion: between the V2G message class instance and the dictionary (json format); between the dictionary and XML; XML and EXI format. Conversion in the opposite direction is also implied. This module allows the user to generate a default dictionary for a specific chagring mode (AC or DC), which is used by the station module. This dictionary contains the V2G request and V2G response pairs and the data for the V2G response is also included. According to the user’s requirements, this dictionary can be modified to suit a specific use case. This module in combination with the “v2gtp” module is used to correctly generate V2GTP responses to received requests.

Another module is “station”, which is my implementation of SECC. In this module I have implemented SECC according to the requirements in [34], it should be mentioned that I have intentionally ignored some requirements. This is because the tool is not intended to be a real implementation of the SECC, but to enable testing of the EVCC. This is what I am providing in this implementation and therefore some specifications could be omitted. So in this module I implement: SDP server to support SECC Discovery Protocol; TCP (TLS) server for V2G communication; ensuring the correct exchange of V2G messages using “v2gtp” and “messages” modules.

The module for enumeration and partial evaluation is “enumerator”. This module allows for example: enumeration of application protocols supported by EVCC; it also provides a check if TLS is enforced by EVCC or not; within TLS it allows checking if the cipher suite used is compliant with [34]. It is easy to add additional requirements for enumeration because the enumerator is implemented as a class, and its instance is passed to the method that provides SECC execution.

Another module for EVCC security testing is “fuzzer”. There is an extensive description of this module in Fuzzer, so I just briefly summarize it here. This module provides the user with four fuzzing modes for EVCC testing. Within these modes, the user can choose: to fuzz all messages and all parameters; or to fuzz only one specific message and parameters defined in the configuration file; or the all messages and parameters for fuzzer is selected based on the configuration file; or the user edits the dictionary containing V2G request and response and pass it as an argument via CLI. The fuzzer provides several modes for each parameter and data type, and fuzzing is performed based on the mode.

Chapter 4

Usage of the tool

This chapter describes how to use the tool. First, I describe how to install/deploy the tool. Then, for each important module, I show what options it provides and how to use them. All the essential functionalities that the tool contains are mentioned. Parts of this chapter also refer to the content of the attached media, which contains videos with examples that are described in this chapter.

Unfortunately, this tool has not been tested on a real CCU that would have EVCC implemented. Given the knowledge from the theoretical part, it is important to mention that to work with a real EVCC, a PWM would have to be implemented to guarantee HLC enforcement. The HPGP module and therefore SLAC should work as expected according to the theoretical part.

Considering the previous paragraph, it is only verified to work for the setup and configuration described in V2G setup. This setup assumes HLC enforcement and uses an HPGP device to handle the physical and data link layer. For testing the tool, the implementation from [73] was used as the real EVCC.

4.1 How to deploy the tool

First of all, it is important to mention that the tool assumes the same V2G setup and configuration as mentioned and explained in V2G setup. It is necessary to have "root" permissions to use all functionality of the tool, because the tool uses low-level access (for example for live sniffing). Root permissions are not required for other modules. So if the user wants to use the sniffing functionality, it is necessary to install the tool using pip and use "sudo" for this module. If not, then the user just needs the setup below using poetry and pyenv. The tool has been developed for Python version 3.10.12, its functionality has been verified for this version, so I recommend using the same Python version or newer.

The tool will be published as a repository on GitHub, so it will be possible to download the source code or just ".whl" package and install it using pip. For the purposes of this thesis, the entire repository is located on the attached media, specifically in the "tool" folder.

For the tool to work properly, it is necessary to have "V2Gdecoder" running in web service mode (available from [74]). Therefore, it is necessary to clone the repository and download the release as "V2Gdecoder.jar" file. This file needs to be placed in the cloned repository to have access to the XML schemas. Once this is completed, it is then possible to use "V2Gdecoder". To run in web service mode, the "-w" option (Code listing 13) is used.

First, I will describe what steps to take if the user wants to modify the tool (or does not need to use the sniffer module). I describe my development setup below. The individual steps are as

Code listing 13 V2Gdecoder – Run as a web service

```
# In repository with V2Gdecoder.jar
java -jar V2Gdecoder.jar -w
```

Code listing 14 V2GEvil – pyenv commands

```
# In V2GEvil folder (repo)
VERSION="3.10.12"
pyenv install "$VERSION"
pyenv local $VERSION
```

follows:

1. copy V2GEvil directory from tool directory / clone V2GEvil repository,
2. use cd to V2GEvil folder,
3. use pyenv to manage python versions (follow commands in Code listing 14),
4. install poetry,
5. use poetry install to reads the pyproject.toml file, resolves the dependencies, and installs them (Code listing 14),
6. use poetry shell to create the virtual environment and spawn the shell within that environment (Code listing 15).

Once the user has this development setup ready, a simple script "build_and_deploy.sh" can be used to build and deploy the tool. I created this script to speed up the process of creating a ".whl" package and then installing it with "root" privileges. I also created a script to uninstall the old version "V2GEvil". All the necessary scripts are in the root directory of the tool in the "scripts" folder.

In case the user only wants to use the tool (not develop), it is enough to have the mentioned Python version, the V2GEvil repository and install the ".whl" package from the dist folder. The necessary commands are shown in Code listing 16. As I mentioned previously, for the tool to work properly, V2Gdecoder must be running as a web service in the default settings (localhost, port 9000). Subsequently, the tool is ready for use. With these settings, the tool can only be used for plain TCP connections.

In order to use TLS connection it is necessary to supply a certificate to the "station" module, which will be successfully verified by EVCC. A more detailed explanation of the PKI and its

Code listing 15 V2GEvil – poetry commands

```
# In V2GEvil folder (repo)
poetry install
poetry shell
```

Code listing 16 V2GEvil – installation

```
# In V2GEvil folder
cd dist
sudo pip3 install *.whl
# Next user can use the tool
sudo v2gevil --help
```

Code listing 17 V2GEvil – station, SSL context, load_cert_chain

```
if self.tls_flag:
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(
        certfile=self.certfile_path,
        keyfile=self.keyfile_path,
        password="CHANGE_THIS_BASED_ON_YOUR_PASSWORD",
    )
```

hierarchy is given in [78]. Probably the most important part of understanding why and what certificates need to be used for TLS is explained in the following description.

"During this TLS handshake, the charging station will present its set of digital certificates (SECC certificate, CPO sub-CA 1 certificate, and optionally CPO sub-CA 2 certificate) to the EV in order to identify itself as a trustworthy charging station. The EV will then need verify the digital signature of all certificates - from the SECC certificate all the way up to the pre-installed V2G root CA certificate(s) - and check whether or not any of the certificates have expired. If everything is verified without issue, a TLS session is then successfully established." [78]

The aforementioned certificate set provided by SECC to EVCC is represented in "cpoCertChain.pem" for [73] implementation. "seccLeafCert.pem" is included in this set and "seccLeaf.key" is also required to successfully use the certificate for TLS server. This key corresponds to "seccLeafCert.pem". So in my implementation I use "cpoCertChain.pem" and "seccLeaf.key" to create the SSL context, specifically for the context I load the certificate and key file (it is necessary to provide the password that was used to generate the key).

To get closer to the real case, I used a script to generate certificates and keys from [73] (an option for ISO 15118-2). I generated them in the iso15118 repository folder and then copied the necessary files to the "cert" folder in the "station" module. Then, I edited the certificate and key paths and entered the password in my "station" module. Then I just edited the certificate and key paths and entered the password in my "station" module. Instead of having the password in the source code, the password can be stored in a file because the load method can also work with a file. To test the tool, I left the certificates and keys on the included media and the password used is a simple "12345". The "station" code sample where I load the certificate and SSL context key is listed in Code listing 17. The file paths are defined as enum values in "station_enums.py". The certificate and key are located in the "station" module, specifically in the "cert" folder, and the certificate and key file names match the above ("cpoCertChain.pem" and "seccLeaf.key"). Therefore, to support TLS and test TLS communication with EVCC from [73], the user only needs to use the certificate generation script in iso15118. Then copy the necessary files to the specified certificate folder in the "station" module. To test a real EVCC, it is necessary to guarantee that the EVCC can successfully verify the certificate.

It should be noted that if the tool is installed using pip (i.e. poetry is not used), then

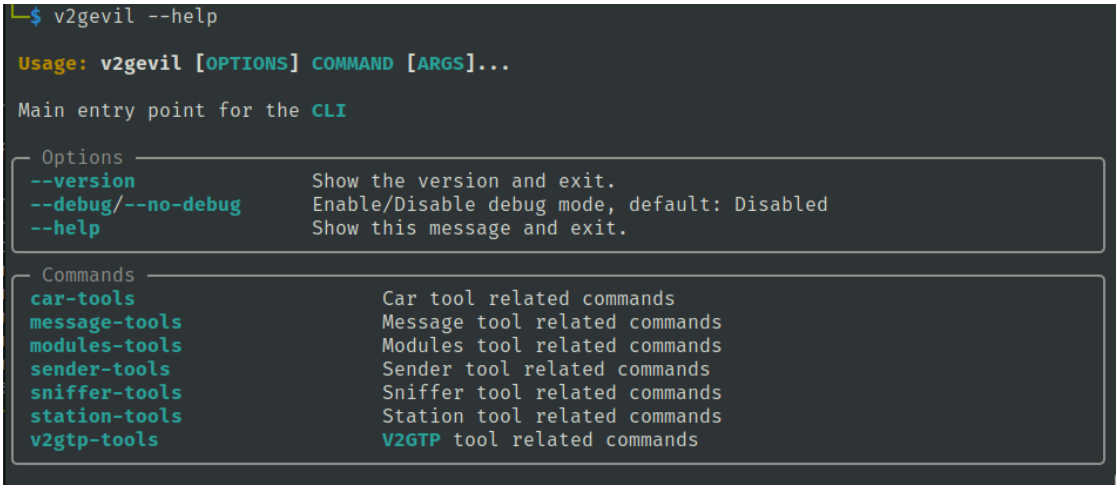


Figure 4.1 V2GEvil – option help

it is necessary to ensure that the key and certificate are in the correct folder, for example "/usr/local/lib/python3.10/dist-packages/src/v2gevil/station/certs". After these steps, the tool is ready to use and test EVCC and supports TLS communication.

4.2 Overview of functionalities

After performing the steps in the previous section, the tool is operational. It can communicate with the EVCC, or intercept and decode the communication in sniffer mode. In this section, I describe the functionalities the tool offers to the user. After this overview, the following sections focus on the main features. These sections provide a detailed description of how to use each functionality and their respective purposes.

Firstly, I present the output of the help option tool (Figure 4.1) and then explain the different commands that the tool offers. It is important to note that commands with the "-tools" suffix are always summary commands for the module. Only after the "-tools" command is called that module-specific commands are allowed.

The first command that is listed is "car-tools". This command will be used to run EVCC in the future and in conjunction with the module for EVCC it will be possible to test SECC, the exact opposite of the goal of this paper. So for this work it is an unusable command, because the logic for EVCC is not yet implemented in my tool. Another command that does not yet contain any logic is "sender-tools". This module was originally designed for sending individual responses and requests. The "sender-tools" module will probably be used for SECC testing in the future, i.e. the tool will be folded as a malicious EVCC. Then in every following section, I mention the modules that can currently be used. The descriptions of the following modules need their own section for clarity.

4.3 Messages module

The second command is "message-tools" this command provided an interface during testing to check if the individual message conversions work correctly. This involved testing, for example, the conversion of a V2G message instance to EXI format, etc. I commented out that functionality after testing.

The only and most important functionality that is accessible through the CLI "message-tools"

```
└─$ v2gevil message-tools generate-default --help

Usage: v2gevil message-tools generate-default [OPTIONS]

Generate messages

 Options
 --charging-mode  -cm  TEXT  Charging mode of the EVSE [default: AC]
 --override-flag             Override flag for default dictionary. If set to True, default
                             dictionary will be overwritten. Default dictionaries are in
                             module called "messages", path to dictionaries:
                             - AC: default_dictionaries/default_dict_AC.json
                             - DC: default_dictionaries/default_dict_DC.json
 --help                      Show this message and exit.
```

Figure 4.2 V2GEvil – message-tools generate-default

is the generation of default dictionaries for the "station" module. These are the dictionaries on the basis of which the pairing of received V2G request with V2G response works. In the given dictionary for each V2G response data is also generated. A sample of part of the dictionary content is in Code listing 1. There are two dictionaries, one for AC charging and one for DC charging, and they are located in the "messages" module in the "default_dictionaries" folder. Unless otherwise specified, these dictionaries are used for normal SECC operation. In addition, these dictionaries are used in the "fuzzer" module.

The user can edit or delete these default dictionaries at will. Therefore, it is important to give him the opportunity to generate them again as they were originally generated. The default dictionary generator is configured so that the default charging mode is AC. Further, by default, dictionaries are not overwritten if they already exist in the folder. The user must select the override-flag, this is to prevent the file from being overwritten unintentionally.

Information on how to use the tool is illustrated in the figure (Figure 4.2). The use of the dictionary generation is intuitive and therefore the help for the "generate-default" command is sufficient to describe it.

4.4 Station module

This module is mainly intended for use in combination with the "enumerator" and "fuzzer" test modules. Since it is desirable to use SECC without these modules in certain cases, I have enabled running SECC via the "station-tools" command.

To start the SECC (station), the start command is used within "station-tools" (Figure 4.3). The user of the tool can configure the SECC using options. The first option is to configure the Ethernet interface to be used by the SECC for communication. This configuration is done using the "**interface**" option followed by the name of the Ethernet interface, which can be obtained for example by the Linux command "ifconfig". If a setup other than the setup specified in V2G setup is used, then it is up to the user to guarantee the correct IPv6 setup for that interface.

Another option is to set the SECC to use or not use TLS as requested in the SDP request. If the user uses the "**accept-security**" option, then the SECC will use the transport layer security that was specified in the SDP request. If the option is "**no-accept-security**", then the SECC uses the opposite. For example, EVCC sends a request in SDP that it requires TLS, then SECC responds that it does not support it and only provides a plain TCP connection.

Another possibility is again related to transport layer security. Here the user directly defines whether TLS should be used or not. If this option and the previous one are used at the same time, then "**tls/no-tls**" takes precedence and the setting "**accept-security/no-accept-security**" does not apply. In case of "**tls**" the SECC sends a TLS indication in the SDP response and then waits for the EVCC to establish TLS connection. Otherwise, it sends an SDP response

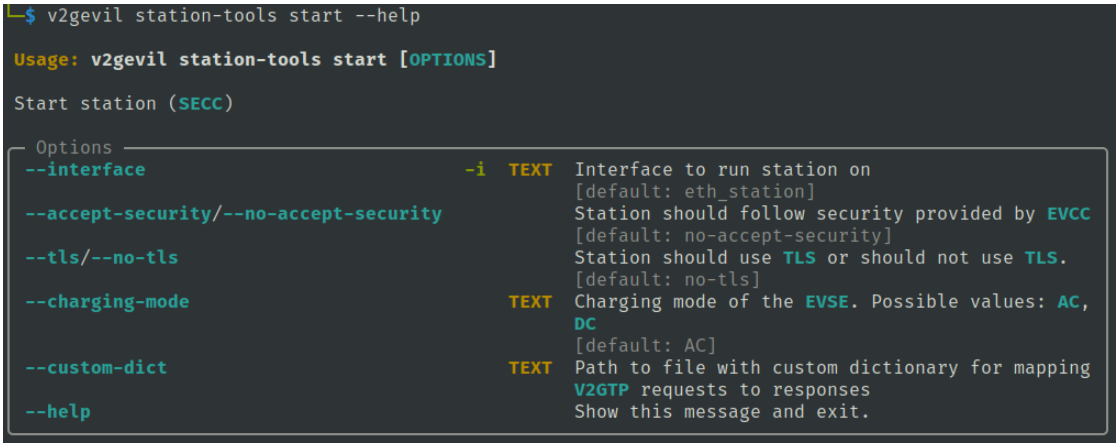


Figure 4.3 V2GEvil – station-tools start

indicating that it does not support TLS and only offers a plain TCP connection and waits to see if EVCC accept it and tries to connect or not.

SECC (station) is set to support AC charging (i.e. AC message set) in the default configuration. If the user wants to change this behavior, then he can use the "**charging-mode**" option. With this option it is possible to specify whether AC or DC charging mode should be used.

The last option is to define a dictionary that the SECC will use to match the V2G request with the V2G response. If the user specifies another dictionary file using "**custom-dict**", then the station will attempt to load and use that dictionary for communication. This option gives the user control over what dictionary will be used for communication. The user thus has full control over the data sent in the responses. The user is responsible for providing all pairs and the necessary parameter values for the V2G responses.

It is important to note that in order to support a runtime response based on the received data for the "supportedAppProtocolReq/Res" V2G response, it is necessary to delete this pair from the dictionary. If this pair is not listed in the dictionary, then the SECC selects a specific SchemaID defined in the request from the EVCC. Otherwise, it uses the data that is specified in the dictionary.

4.5 V2GTP module

This module contains the core logic for parsing, decoding, interpreting and creating Vehicle to Grid Transport Protocol (V2GTP) messages. It is an implementation of V2GTP. The main purpose of this module is to provide the ability to work with V2GTP to other modules, therefore only limited functionality is available through the CLI. Through the CLI the functionality is available from "v2gtp-tools". During the development of the tool the CLI access was also used to verify the correctness of my V2GTP implementation.

The module offers two functionalities via CLI: "**extract**" and "**decode**" (Figure 4.4). Both commands work with "pcap" file, so this is "offline" usage.

The first command allows to decode a V2GTP packet from a pcap file. As shown in the figure (Figure 4.5), the options for this command are the pcap file specification and the packet number in that file. A pcap file is provided for the default settings, to test the tool without the need to obtain a pcap file.

An example of using the "**decode**" command is shown in the figure (Figure 4.6). The default pcap file was used and the packet number was specified using the "packet-num" option. As can be seen, this is an SDP response. The command outputs a hex stream for the V2GTP PDU (header

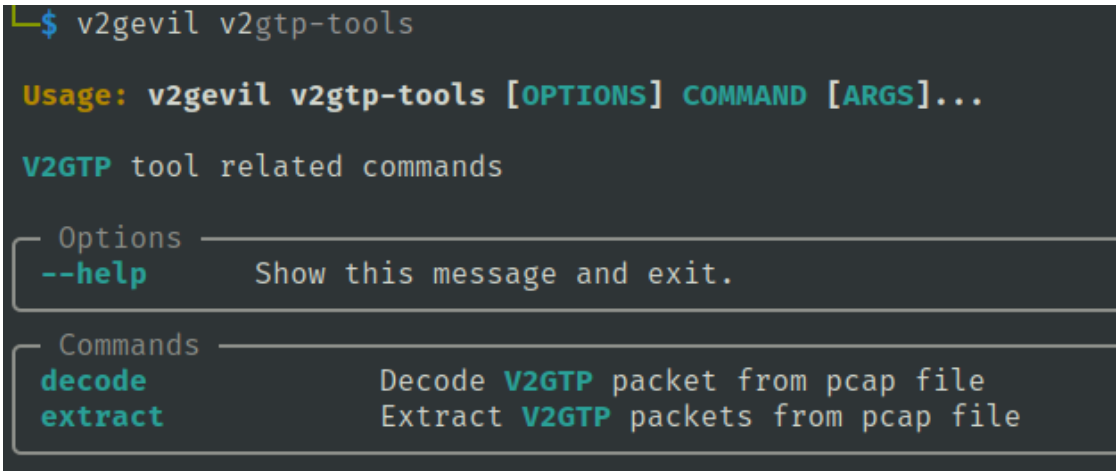


Figure 4.4 V2GEvil – v2gtp-tools commands

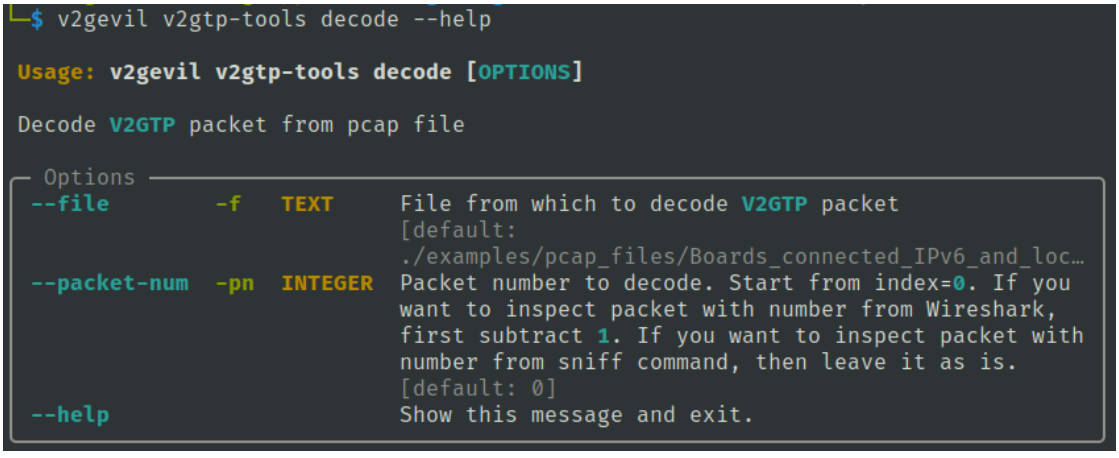


Figure 4.5 V2GEvil – v2gtp-tools decode

```
└─$ v2gevil v2gtp-tools decode --packet-num 121

Trying to decode following raw data as V2GTP packet:
01 FE 90 01 00 00 00 14 FE 80 00 00 00 00 00 00 D2 37 45 FF FE 88 B1 2B F4 73 10 00  ...............
..7E....+.s..
-----------------------------------------------------------------------------------------------
Packet sent from: fe80::d237:45ff:fe88:b12b:15118 => fe80::d237:45ff:fe88:b12a:59791
        V2GTP header: 01fe900100000014
        V2GTP payload: fe80000000000000d23745fffe88b12bf4731000

Payload type is: sdp_response
Decoded V2GTP packet:
 SDP response:
        IP address: fe8000000000000d23745fffe88b12b
        Port: 62579
        Security: 0x10 => No transport layer security
        Transport layer: 0x00 => TCP
-----------------------------------------------------------------------------------------------
```

Figure 4.6 V2GEvil – v2gtp-tools decode SDP response

```
└─$ v2gevil v2gtp-tools decode --packet-num 133

Trying to decode following raw data as V2GTP packet:
01 FE 80 01 00 00 00 24 80 00 EB AB 93 71 D3 4B 9B 79 D1 89 A9 89 89 C1 D1 91 D1 91 81 89 99 D2 6B 9
B 3A 23 2B 30 02 00 00 04 00 40  .......$.....q.K.y.............k.:#+0.....@
-----------------------------------------------------------------------------------------------
Packet sent from: fe80::d237:45ff:fe88:b12a:51342 => fe80::d237:45ff:fe88:b12b:62579
        V2GTP header: 01fe800100000024
        V2GTP payload: 8000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b3a232b30020000040040

Payload type is: exi_message
Decoded V2GTP packet:
 <?xml version="1.0" encoding="UTF-8"?><ns4:supportedAppProtocolReq xmlns:ns4="urn:iso:15118:2:2010:
AppProtocol" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns3="http://www.w3.org/2001
/XMLSchema"><AppProtocol><ProtocolNamespace>urn:iso:15118:2:2013:MsgDef</ProtocolNamespace><VersionN
umberMajor>2</VersionNumberMajor><VersionNumberMinor>0</VersionNumberMinor><SchemaID>1</SchemaID><Pr
iority>1</Priority></AppProtocol></ns4:supportedAppProtocolReq>
-----------------------------------------------------------------------------------------------
```

Figure 4.7 V2GEvil – v2gtp-tools decode V2G EXI message

and payload). It recognised the V2GTP payload as an SDP response based on the header. Then decoded and interpreted parts of the payload in the output. For demonstration I also attach an image (Figure 4.7) for the packet where the V2G message is captured in EXI format.

The other functionality that is available for this module via "v2gtp-tools" is "**extract**". With this command, packets that have a V2GTP layer are extracted from the provided file. For such packets, a hex stream of bytes for the V2GTP header and body is listed separately in the output. The output and use of "**extract**" is shown in the Building the tool chapter, specifically in the figure (Figure 3.1). The "**extract**" command offers no other CLI options besides the file specification.

These two functionalities also served as a predecessor for the following module "sniffer", which slightly improves the functionalities and allows their use for live sniffing and decoding entire V2G communication.

4.6 Sniffer module

This module is accessible from the CLI using the "sniffer-tools" command and allows sniffing and decoding of V2G communication. The sniffer module allows both offline analysis (from a pcap file) and working with live communication. The commands are basically divided into "**inspect**" and "**sniff**".

```
└─$ v2gevil sniffer-tools inspect -p 132 --decode
Analyzing packets from file %s ./examples/pcap_files/Boards_connected_IPv6_and_localhost.pcapng
Inspecting packet number: 132
###[ cooked linux ]###
  pkttype   = sent-by-us
  lladdrtype= 0x1
  lladdrlen = 6
  src       = '\\xd07E\\x88\\xb1+'
  proto     = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 147694
     plen      = 76
     nh        = TCP
     hlim      = 64
     src       = fe80::d237:45ff:fe88:b12a
     dst       = fe80::d237:45ff:fe88:b12b
###[ TCP ]###
        sport     = 51342
        dport     = 62579
        seq       = 3119412828
        ack       = 1049597150
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 507
        chksum    = 0x8d2a
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (802336115, 502401227))]
###[ Raw ]###
           load      = '\x01\\xfe\\x80\x01\x00\x00\x00$\\x80\x00팕 q\\xd3K\\x9byщ\\xa9\\x89\\x89\\xc1
ëë\\x81\\x89\\x99\\xd2k\\x9b:#+0\x02\x00\x00\x04\x00@'
```

Figure 4.8 V2GEvil – sniffer-tools, inspect command (all layers)

4.6.1 Command inspect

In the first case, the command is similar to "**decode**" for "v2gtp-tools". Command called "**inspect**" allows you to select a packet from a given file based on its packet number.

For this selected packet, it provides the "show" option to select which layers of the packet should be displayed to the user. There are 4 choices of values for the "show" option: raw, tcp, ipv6, all. For the "all" option, all layers contained in the packet are listed, as can be seen in the figure (Figure 4.8). For the other options (ipv6, tcp, raw), the layers for the packet are displayed starting from the layer the user specifies. For example, for "tcp" the layers from TCP layer and higher are printed from the ISO/OSI model perspective. In the picture (Figure 4.8) the higher layers are listed downward.

The command also allows decoding of V2GTP packets. If the user wants to decode the packets he must use the "decode" option. For the "decode" option, it is still necessary to specify the "show" option with the value "raw". Then "**inspect**" works the same as "**decode**" for the "v2gtp" module.

If this is not specified by "show" option with value "raw" then only layer listing is performed. This is also applicable if the user specifies that the packet has a raw layer, but the packet is not actually V2GTP. So in both cases, only the layers for the packet are listed and no decoding is done.

An example of how the command can be used is shown in the figure (Figure 4.9). The packet number is 132, I specify to use the raw layer and require decoding. The decoding is successfully performed because the packet does indeed contain a raw layer in the form of a V2GTP layer.

This command is an extension of "**decode**" from "v2gtp-tools".

```
└─$ v2gevil sniffer-tools inspect -p 132 --show raw --decode
Analyzing packets from file %s ./examples/pcap_files/Boards_connected_IPv6_and_localhost.pcapng
Inspecting packet number: 132

Trying to decode following raw data as V2GTP packet:
01 FE 80 01 00 00 00 24 80 00 EB AB 93 71 D3 4B 9B 79 D1 89 A9 89 89 C1 D1 91 D1 91 81 89 99 D2 6B 9
B 3A 23 2B 30 02 00 00 04 00 40  .......$.....q.K.y..............k.:#+0.....@
-------------------------------------------------------------------------------------------------
Packet sent from: fe80::d237:45ff:fe88:b12a:51342 => fe80::d237:45ff:fe88:b12b:62579
        V2GTP header: 01fe800100000024
        V2GTP payload: 8000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b3a232b30020000040040

Payload type is: exi_message
Decoded V2GTP packet:
 <?xml version="1.0" encoding="UTF-8"?><ns4:supportedAppProtocolReq xmlns:ns4="urn:iso:15118:2:2010:
AppProtocol" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns3="http://www.w3.org/2001
/XMLSchema"><AppProtocol><ProtocolNamespace>urn:iso:15118:2:2013:MsgDef</ProtocolNamespace><VersionN
umberMajor>2</VersionNumberMajor><VersionNumberMinor>0</VersionNumberMinor><SchemaID>1</SchemaID><Pr
iority>1</Priority></AppProtocol></ns4:supportedAppProtocolReq>
-------------------------------------------------------------------------------------------------
```

Figure 4.9 V2GEvil – sniffer-tools, inspect command with decoding

4.6.2 Command sniff

Within the "**sniff**" command, the user can decide whether to parse the pcap file or start live sniffing. The user specifies this decision using the options "pcap" or "live".

As I already mentioned, the "**extract**" command offers no further output customization besides V2GTP packet extraction and pcap file selection. For this reason, I have improved the pcap file handling and included this functionality in the "sniffer" module. The functionality is accesible in this module by using the "**sniff**" command with the "pcap" option. The options are the same as live sniffing except the source of communication is different, thus this offline analysis is included in sniff command.

The "file" option can only be used for offline sniffing and the "interface" option can only be used for live sniffing. The former option specifies from which pcap file the data should be read. In the latter case, the user specifies the Ethernet interface on which the sniffer will listen for communication.

Other options that the user can use for this command are "ipv6", "v2gtp", "decode". These options are common for both live sniffing ("live" option) and for processing packets from a pcap file ("pcap" option). The meaning of these options is therefore the same for both types of sniffing.

This command using "pcap" offers options compared to "**extract**" to decode packets and choose which packets are processed: all packets; or IPv6 packets; or only V2GTP packets. This is an improvement over "**extract**" of "v2gtp-tools".

The user can select packets by using the "ipv6/no-only-ipv6" and "v2gtp/no-only-v2gtp" switches. The default setting is IPv6 packet sniffing for both "pcap" and "live".

If the user also wants to perform decoding, then the "decode" option is used for this purpose. If this option is specified, then decoding is performed for V2GTP packets. For live sniffing, decoding is done sequentially as individual packets are captured. For the default setting, decoding is disabled. If some packets are not V2GTP, then the tool detects this fact for a particular packet and prints it.

The use of the live sniffing tool with IPv6 packet capture and simultaneous V2GTP packet decoding is illustrated in the figures (Figure 4.10 and Figure 4.11). This V2G communication takes place between EVCC and SECC from [73].

On the attached media, I have placed records in the "videos" folder showing how the tool works. For this module these are videos with the following names:

1. "v2gevil_sniffer_live_with_decoding.mp4",
2. "v2gevil_live_sniff_v2gtp_flag.mp4",

```
└─$ sudo v2gevil sniffer-tools sniff --live --ipv6 --decode
Sniffing packets...
Sniffing packets live on interface %s eth_car
Packet from: fe80::d237:45ff:fe88:b12a to ff02::1.
        V2GTP header: 01fe900000000002
        V2GTP payload: 1000
        Decoded V2GTP packet:
        SDP request:
        Security: 0x10 => No transport layer security
        Transport layer: 0x00 => TCP

Packet from: fe80::d237:45ff:fe88:b12b to fe80::d237:45ff:fe88:b12a.
        V2GTP header: 01fe900100000014
        V2GTP payload: fe80000000000000d23745fffe88b12bcdd21000
        Decoded V2GTP packet:
        SDP response:
        IP address: fe80000000000000d23745fffe88b12b
        Port: 52690
        Security: 0x10 => No transport layer security
        Transport layer: 0x00 => TCP

No VTGTP layer for this packet:
        Ether / IPv6 / TCP fe80::d237:45ff:fe88:b12a:56122 > fe80::d237:45ff:fe88:b12b:52690 S
```

Figure 4.10 V2GEvil – sniffer-tools, sniff command with decoding 1

```
No VTGTP layer for this packet:
        Ether / IPv6 / TCP fe80::d237:45ff:fe88:b12b:52690 > fe80::d237:45ff:fe88:b12a:56122 SA

No VTGTP layer for this packet:
        Ether / IPv6 / TCP fe80::d237:45ff:fe88:b12a:56122 > fe80::d237:45ff:fe88:b12b:52690 A

Packet from: fe80::d237:45ff:fe88:b12a to fe80::d237:45ff:fe88:b12b.
        V2GTP header: 01fe800100000024
        V2GTP payload: 8000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b3a232b30020000040040
        Decoded V2GTP packet:
        <?xml version="1.0" encoding="UTF-8"?><ns4:supportedAppProtocolReq xmlns:ns4="urn:iso:15
118:2:2010:AppProtocol" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ns3="http://
www.w3.org/2001/XMLSchema"><AppProtocol><ProtocolNamespace>urn:iso:15118:2:2013:MsgDef</Protocol
Namespace><VersionNumberMajor>2</VersionNumberMajor><VersionNumberMinor>0</VersionNumberMinor><S
chemaID>1</SchemaID><Priority>1</Priority></AppProtocol></ns4:supportedAppProtocolReq>
```

Figure 4.11 V2GEvil – sniffer-tools, sniff command with decoding 2

3. “v2gevil_live_decode_flag.mp4”.

These videos show how the tool works depending on the settings of each parameter. The mentioned folder also contains videos to show how other modules work.

4.7 Enumerator module

This module is used to enumerate basic data about EVCC. This includes the following data: supported application protocols and their versions; TLS version, supported cipher suites, used cipher suites for TLS communication session; TLS check.

For users, the module is accessible via “modules-tools”, specifically using the “**enumerate-EV**” command. This module works by first setting the enumeration mode and then starting the station (SECC). During communication between the SECC and the EVCC, the enumerator collects information based on the enumeration mode. After the user chooses to terminate the communication, the collected data is printed out.

Since SECC is run within this module, the user has the option to set which interface SECC will run on using the “interface” option.

```
└─$ v2gevil modules-tools enumerate-EV --mode all                                130 ×
Station configuration:
Interface: eth_station
IPv6 address: fe80::d237:45ff:fe88:b12b
Protocol: b'\x00'
SDP port: 15118
TCP port: 53313
TLS flag: True
Accept security: True
Charging mode: AC
Validate flag for model_dump/construct: True
Cert PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/cpoCertChain.pem
Keyfile PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/seccLeaf.key
SDP server started
SDP server is running in while loop
TLS, Connected by:  ('fe80::d237:45ff:fe88:b12a', 60132, 0, 10)
Negotiated Cipher Suite: ('ECDHE-ECDSA-AES128-SHA256', 'TLSv1.2', 128)
Negotiated TLS Version: TLSv1.2
TCP server loop ended after connection established
TCP server connection established
--------------------------------------------------------------------------------
Received from client: 01fe8001000000248000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b3a232b3002
0000040040
{'AppProtocol': [{'ProtocolNamespace': 'urn:iso:15118:2:2013:MsgDef', 'VersionNumberMajor': 2, 'Vers
ionNumberMinor': 0, 'SchemaID': '1', 'Priority': 1}]}

Created response object:
{'ResponseCode': 'OK_SuccessfulNegotiation', 'SchemaID': '1'}
--------------------------------------------------------------------------------
```

Figure 4.12 V2GEvil – modules-tools, enumerate-EV command with mode “all”, part 1

Furthermore, the user has the possibility to select the enumeration mode using the “mode” option. The options for enumeration modes are as follows: “supported_protocols”, “tls_check”, “tls_enum”, “all”.

If the user chooses the first mentioned mode, then only the information about supported application protocols is saved. This information is sent from EVCC in the V2G message “supportedAppProtocolReq”. In the case of “tls_check”, a check is performed to see if EVCC requires and supports TLS communication. Another mode related to TLS is used to enumerate: TLS versions; cipher suites used for TLS communication; cipher suites offered. The last mode contains all the previous modes.

In order to be able to obtain data for “tls_enum” mode, it is necessary that the communication between SECC and EVCC takes place using TLS. Therefore, the enumerator module sets the tls flag based on the enumeration mode. This flag is then passed to the station.

If the mode requires TLS communication to collect the necessary data, then the TLS flag set to “true” is passed to “station.start_async”. This is the case, for example, when the “tls_enum” mode is used.

The output of this tool for “all” mode is attached in the picture. TLS communication is used during the communication between SECC and EVCC. To support TLS in EVCC, it is necessary to modify its configuration file referenced from “.env”. In this “json” file, the value of “useTls” must be changed from false to true.

I recorded the work with the “enumerator” module tool and the videos are available on the attached media in the “videos” folder. These are the following recordings:

1. “v2gevil_enumerator_mode_all_TLS_used.mp4”,
2. “v2gevil_enumerator_mode_tls_check_NO_TLS.mp4”.

In the first video the enumeration mode “all” and SECC is used, so it will require TLS communication. I have set TLS support using the configuration file in EVCC as well. Otherwise the communication would not proceed and the output of the tool would not contain all the required information. In the video it can be seen that first the station configuration (SECC) is

```
--------------------------------------------------------------------------------
EV enumeration results:
--------------------------------------------------------------------------------
Supported App protocols result:
{'AppProtocol': [{'ProtocolNamespace': 'urn:iso:15118:2:2013:MsgDef', 'VersionNumberMajor': 2,
'VersionNumberMinor': 0, 'SchemaID': '1', 'Priority': 1}]}

Supported protocols by EV:
Number of supported protocols: 1
ProtocolNamespace: urn:iso:15118:2:2013:MsgDef, VersionMajor: 2, VersionMinor: 0, SchemaID: 1,
Priority: 1
--------------------------------------------------------------------------------
TLS check result:
EV requested security: TLS and as a transport protocol: TCP for communication
--------------------------------------------------------------------------------
TLS enumeration result:
TLS negotiated version: TLSv1.2
TLS negotiated cipher suite: ('ECDHE-ECDSA-AES128-SHA256', 'TLSv1.2', 128)
TLS shared ciphers: [('ECDHE-ECDSA-AES128-SHA256', 'TLSv1.2', 128)].
Shared ciphers are ciphers available in both the EV and the EVSE.
```

Figure 4.13 V2GEvil – modules-tools, enumerate-EV command with mode "all", part 2

listed, and then the communication is performed. Received and sent messages are output (within the station module). After the user ends the communication (CTRL+C), SECC is terminated and then the enumerated data is printed. In the second video, TLS communication is not used and only TLS check is performed.

4.8 Fuzzer module

The last module of my tool is "fuzzer". This module is used to test the EVCC implementation, specifically the processing of V2G messages. This testing is done by sending various non-valid values for parameters in V2G messages.

For the user, the functionality of this module is accessible via "modules-tools", as well as "enumerator". Within "modules-tools", the fuzzer is accessible using "**fuzz-EV**".

Fuzzer first generates/edits the dictionary based on the passed parameters. This dictionary is then passed as an argument to the method that runs SECC. Therefore, the fuzzer again has a "interface" option to specify which interface SECC will run on.

The next option determines which message set is used, whether for AC or DC charging. The default mode is AC and the user can change it using the "charging-mode" option. Charging mode specifies which dictionary the fuzzer will use and modify. Both charging modes have their specific V2G messages, their resolution is defined in [34].

The most important option for the "**fuzz-EV**" command is "mode". Based on mode, it is specified how the fuzzer should work. I describe the different modes below.

4.8.1 Mode "all"

This mode means that the fuzzer modifies all messages and all parameter values of these messages (except application protocol messages). This is the most basic and simplest mode for fuzzing.

The message can have simple data type parameters and complex data type parameters. Complex data type parameters are those that contain additional parameters. Each complex data type parameter and each simple data type parameter has its own fuzz method. In basic terms, it can be described as follows: for a complex data type parameter, a fuzz method is called for each parameter contained in that complex parameter. If it is a simple data type parameter, then fuzzing of the value is performed based on "ParamFuzzMode". If this mode is not specified (or does not exist), the mode is selected randomly from the modes that exist for that simple

```
└─$ v2gevil modules-tools fuzz-EV --mode all
********************************************************************************
FUZZING METHOD START for FuzzerSessionSetupRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'EVSEID': 'FRA23E45B78C', 'EVSETimeStamp': 1700593914}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': -7723727986904441704, 'EVSEID': -8389756374973661238, 'EVSETimeStamp': '
TjaqlKgNAzGvYMVqEOCddeUghgZXjauApJnTRCuFTybtVEGvqCrCSnHeFUZOWKBhmGiNDpeK'}
FUZZING METHOD END for FuzzerSessionSetupRes
********************************************************************************
```

Figure 4.14 V2GEvil – fuzz-EV mode “all”, generated messges part 1

```
********************************************************************************
FUZZING METHOD START for FuzzerServiceDiscoveryRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'PaymentOptionList': {'PaymentOption': ['Contract', 'ExternalPayme
nt']}, 'ChargeService': {'ServiceID': 1, 'ServiceName': 'AC_DC_Charging', 'ServiceCategory
': 'EVCharging', 'FreeService': True, 'SupportedEnergyTransferMode': {'EnergyTransferMode'
: ['AC_three_phase_core', 'DC_extended']}}, 'ServiceList': {'Service': [{'ServiceID': 3, '
ServiceName': 'Fast Internet', 'ServiceCategory': 'Internet', 'FreeService': True}, {'Serv
iceID': 2, 'ServiceName': 'Certificate', 'ServiceCategory': 'ContractCertificate', 'FreeSe
rvice': True}]}}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': 4331552912365351140, 'PaymentOptionList': {'PaymentOption': ['sOVhGgxBAg
pBxgpGggAGxTOHxqgTFgYCCyVJofVkGFgvfnhMqKqsfZa']}, 'ChargeService': {'ServiceID': 'KgCSWzck
avcnFcMbAcwhsIHEDtCxABPZAKZWARiEkGiTyCqskDdjtavYHVIkdqMiaHyePlkjrCMnOYrMnVYjaquMzkWcRSPYWN
NL', 'ServiceName': -7749399962964235051, 'ServiceCategory': -6502281958587080966, 'Servic
eScope': 'XsXDeQKfLqNiUYdkaQbotElTzoxNYGtwyVecWeibAJZgycxcNQEgGqhPiULMafHkEBcpaSXmdAxrV',
'FreeService': 'vXQftCCLmcUJIzWRCetcRvUgHSPHyduXEnciyIc', 'SupportedEnergyTransferMode': {
'EnergyTransferMode': [4434500343658101378]}}, 'ServiceList': {'Service': [{'ServiceID': 2
.5500147264638l4e+18, 'ServiceName': -6763399041126583398, 'ServiceCategory': -28308539128
32565010, 'ServiceScope': -4353568443856179010, 'FreeService': 'MctkTjKhDZujZOLeqAuEkWLVNR
xuRPBGGeWwzbAcRo'}]}}
FUZZING METHOD END for FuzzerServiceDiscoveryRes
```

Figure 4.15 V2GEvil – fuzz-EV mode “all”, generated messges part 2

type. The nesting is done until all simple data type parameters are fuzzed. The same logic is also used in other modes (config mode and message mode), but in these modes it is possible, for example: to specify a specific fuzz method for a parameter based on a configuration file; or to specify which parameters are to keep their original values and which parameters will be fuzzed.

In this fuzzer mode a random mode is always used for all parameters. This mode is set as default because it does not need any user interaction or any configuration file. So this mode does not use the configuration file that is in the “config” folder in the “fuzzer” module. Neither the custom dictionary specification (option “custom-dict”) nor the message name specification (option “message-name”) is relevant for this mode. This mode may generate an inappropriate message format because random data is used for all parameters and it may happen that the V2G response cannot be assembled. This mode is mainly for demonstration purposes. Moreover, due to the fact that fuzzing is for all messages, the communication usually fails already for the first V2G message exchange. Complete control on fuzzing is provided by “config” mode, because the fuzzer is controlled based on the configuration file.

A sample of the generated messages for this mode is shown in the pictures (Figure 4.14 and Figure 4.15). This images show the output of the fuzzer tool before the communication starts. The output of the tool is much longer, for illustration I attach the parts where only the first two messages are shown.

How the fuzzer works with this mode is shown in the video “v2gevil_fuzzer_all.mp4”. The video shows how the fuzzer created “SessionSetupRes” and then sent it.
An “java.lang.RuntimeException” error occurred in EVCC due to a malicious V2GTP message. For demonstration I attach here parts of the output of the “fuzzer” module (Figure 4.16 and

```
└─$ v2gevil modules-tools fuzz-EV --mode all
WARNING  2024-01-09 23:58:03,125 src.v2gevil.fuzzer.gen_types: No valid    gen_types.py:75
         value specified for xs:short, using valid value randomly
         generated.Disclaimer: Generated value - meets the conditions for
         length and type but may not meet the valid value for particular
         parameter.
Station configuration:
Interface: eth_station
IPv6 address: fe80::d237:45ff:fe88:b12b
Protocol: b'\x00'
SDP port: 15118
TCP port: 61233
TLS flag: False
Accept security: True
Charging mode: AC
Validate flag for model_dump/construct: False
Cert PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/cpoCertChain.pem
Keyfile PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/seccLeaf.key
SDP server started
SDP server is running in while loop
Plain TCP, Connected by:  ('fe80::d237:45ff:fe88:b12a', 60638, 0, 10)
TCP server loop ended after connection established
TCP server connection established
--------------------------------------------------------------------------------
```

Figure 4.16 V2GEvil – modules-tools, fuzz-EV command with mode “all”, part 1

```
--------------------------------------------------------------------------------
Received from client: 01fe80010000000248000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b
3a232b30020000040040
False
{'AppProtocol': [{'ProtocolNamespace': 'urn:iso:15118:2:2013:MsgDef', 'VersionNumberMajor'
: 2, 'VersionNumberMinor': 0, 'SchemaID': '1', 'Priority': 1}]}

Created response object:
{'ResponseCode': 'OK_SuccessfulNegotiation', 'SchemaID': '1'}
--------------------------------------------------------------------------------
Received from client: 01fe80010000000e8098004011d01b40dd1622c4ac00
False
{'Header': {'SessionID': '00'}, 'Body': {'SessionSetupReq': {'EVCCID': 'D0374588B12B'}}}
Created response object:
{'Header': {'SessionID': '00'}, 'Body': {'SessionSetupRes': {'ResponseCode': 6958597786855
330308, 'EVSEID': 'hQnjsRdXDUrfaUyXJxtcrfCauNZwjqipOpaQQVyzPnzQAqdNIoNGWUUxbQYLcYtkfVUKJrj
dPIXIkzlqsRuuLfvTGtZm', 'EVSETimeStamp': 1700593914, 'EVSETimestamp': -2.0376020479938703e
+18}}}
Key EVSETimestamp doesn't belong to any namespace
TCP connection closed
SDP server is running in while loop
^CStopping SDP server by KeyboardInterrupt
SDP server stopped
```

Figure 4.17 V2GEvil – modules-tools, fuzz-EV command with mode “all”, part 2

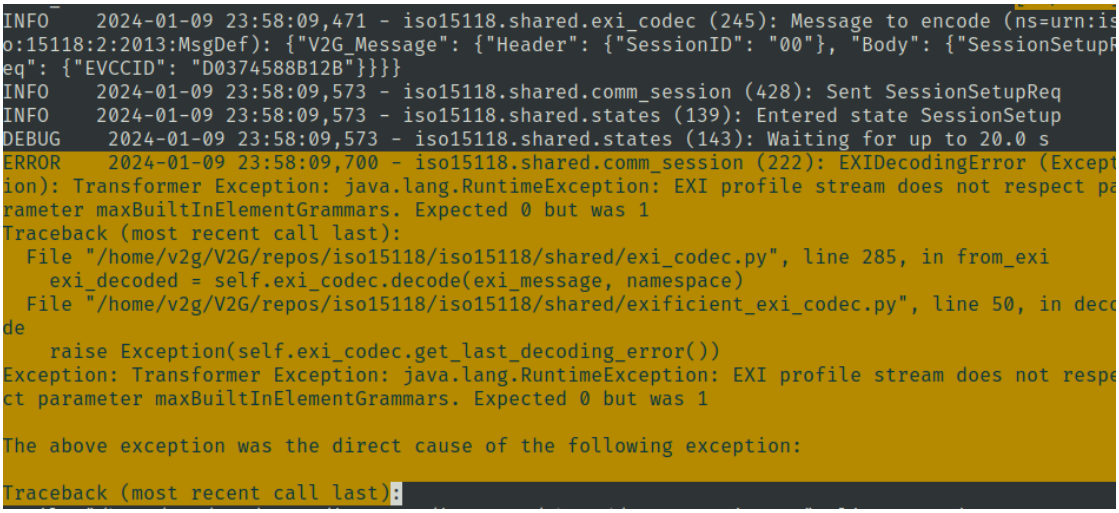


Figure 4.18 Failure in EVCC side – error caused by fuzz, mode “all”

Figure 4.17). Next, the figure (Figure 4.18) shows what fuzzer can cause on the EVCC side (the error always depends on the specific EVCC implementation).

4.8.2 Mode “custom”

This mode is described only briefly because it is not essentially a fuzzer function. When specifying this option, it is also required to pass a file using the “custom-dict” option. Passing a file is defining a path to that file. This mode assumes that the user has already customised this dictionary. This dictionary is then used by the charging station for handling communication, that is, to send a corresponding V2G response to a V2G request based on the dictionary.

This mode does not use any other options. Like the previous one, it does not use any configuration file.

4.8.3 Mode “config”

This mode allows the user to control the fuzzer using a configuration file. The user can specify message names, parameter names and their fuzz mode in the configuration file. An explanation of the configuration file and how each file entry is interpreted for the fuzzer is given below. The configuration file can be specified by the user by using the “config-filename” option and providing a filename. This file must be located in the fuzzer module, specifically in the “config” folder.

The parameters that are specified for the message in the configuration file are fuzzed. Regardless of the specification of “RequiredParams”, all parameters that are specified for the message in the configuration file will be fuzzed.

If the user only wants to specify which parameters to fuzz, but no longer wants to specify how, then he uses “RequiredParams”. This specifies which parameters are required by the user for fuzzing. Further, if no specific mode is specified for the parameters in the file, then it will be chosen randomly.

The user can use “RequiredParams” in combination with specifying modes for specific parameters. The fuzzer will include all parameters specified for a given message. It also includes the parameters specified in “RequiredParams” for the message. If there is no entry in the configuration file for the parameter specified in “RequiredParams”, then the random fuzz mode selection is used for that parameter. If the parameter is defined both individually and in “RequiredParams”

for the message, then the setting that the user specified for the particular parameter is used. The parameter does not need to be specified in "RequiredParams", it is enough if it has its own entry for the message, then it will be fuzzed.

If it is a parameter (complex data type) that contains other parameters, then the same procedure is used as for configuring the message. Thus, the user can specify "RequiredParams" and/or specific parameters. In case it is an end parameter (simple data type), then user specifies only the mode for the parameter (no "RequiredParams").

If the user defines the message in the configuration file, but does not specify anything else, then fuzzing is performed for the mandatory parameters (for the message defined in [34]). The mode for the parameters is chosen randomly.

The same approach is used for parameters. If the user just provides a parameter name and no other specification, then fuzzing of the parameter is performed. If it is a complex data type parameter, then only mandatory subparameters are fuzzed (mandatory according to [34]). If it is an end parameter (simple data type), then only a random fuzzing mode is selected for it.

If the message name is not in the configuration file, then fuzzing of the message is not performed. This is not true for parameters, so for further explanation I assume that the parameter is not listed in the configuration file separately. Once the message name is already included in the configuration file, then the fuzzer performs fuzzing of the parameters that are mandatory. Mandatory according to [34] or according to "RequiredParams". If the user specifies "RequiredParams", then only those parameters contained in ("RequiredParams") are considered mandatory. In case the user does not specify these "RequiredParams", the tool will warn him about it and random fuzz mode is selected for such parameters.

All mandatory parameters (according to [34]) for each message are provided regardless of the configuration file. This means that the fuzzer automatically selects valid values from the default dictionary for parameters that are not intended for fuzzing based on the configuration file. All parameters and their values that have not been changed by fuzzing remain the same as in the default dictionary.

The user can also specify (in the configuration file) whether the value should be valid for the end parameter. In this case, the value is either taken from the default dictionary or a valid value is generated. The generation of a valid value is based on a specific data type. Generating a valid value only satisfies that the data type of the value is correct.

For a better understanding of the configuration file and how "fuzzer" works with it, I attach examples of the contents of the configuration file and the associated fuzzer output. I use these examples to describe what the information in the configuration file means. I demonstrate this description with one specific message. The principle is the same for other messages.

The following examples show the relation between the contents of the configuration file and the messages generated by the fuzzer. Thus, the examples do not include subsequent communication with EVCC. A demonstration of what such generated messages can do is shown in the following section. The following section describes another mode. The "message" mode also uses a configuration file. so the demonstration of communication using malicious messages is there.

4.8.3.1 Example 1

The first example of using fuzzer with a configuration file is the simplest. In the configuration file, the user specifies only the names of the messages he wants to modify. The names are given without any further specification of parameters, etc. The configuration file may look like this (Code listing 18).

Based on this configuration file, the fuzzer will perform fuzzing only for the mentioned messages. The other messages will remain the same as in the default dictionary for SECC. The fuzzer will only fuzz the parameters that are mandatory for the message according to [34]. For these mandatory parameters, the specific mode for fuzzing that parameter is chosen randomly.

The fuzzer output is shown in the figure (Figure 4.19). The dictionary for messages before

Code listing 18 V2GEvil – fuzzer config file, example 1

```
# ev_fuzzer_example_1.toml
# Configuration for fuzzing messages
# Only the messages listed here will be fuzzed
[supportedAppProtocolRes]
# Mandatory parameters according to ISO15118 are: ResponseCode.

[SessionSetupRes]
# Mandatory parameters according to ISO15118 are: ResponseCode, EVSEID
```

```
└─$ v2gevil modules-tools fuzz-EV --mode config --config-filename ev_fuzzer_example_1.toml

********************************************************************************
FUZZING METHOD START for FuzzerSupportedAppProtocolRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK_SuccessfulNegotiation', 'SchemaID': '1'}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': 4483016746495683834, 'SchemaID': '1'}
FUZZING METHOD END for FuzzerSupportedAppProtocolRes
********************************************************************************
FUZZING METHOD START for FuzzerSessionSetupRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'EVSEID': 'FRA23E45B78C', 'EVSETimeStamp': 1700593914}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': 'FQgFOeTcBNnsOnzvHfrYzDVswkVBBPDMWcizmqEzsQOSkTlIQmIDNtsJDUDAahEOwzxIFNc
AcccJEMiqDKHnx', 'EVSEID': 'W', 'EVSETimeStamp': 1700593914}
FUZZING METHOD END for FuzzerSessionSetupRes
********************************************************************************
```

Figure 4.19 V2GEvil – fuzz-EV config mode, example 1

and after fuzzing can be observed in the output. It can also be seen that the fuzzer used only the mandatory parameters for the message and left the others as they originally were. A sample is also shown in the video “v2g_evil_fuzzer_config_example_1.mp4”.

4.8.3.2 Example 2

In the second example, the user made the following settings. The messages to be fuzzed are “supportedAppProtocolRes” and “SessionSetupRes”.

The user sets “ResponseCode” and “SchemaID” as mandatory parameters for the first message. The user specified the fuzzing mode only for the first parameter, and therefore the fuzzing mode for the second parameter is chosen randomly by the fuzzer. The user is informed of this via a warning.

For the second message, the user wants to fuzz the following parameters: “ResponseCode” and “EVSETimeStamp”. For both parameters, the user has specified the mode. For the first parameter, the valid value will be selected from the default dictionary for SECC if the dictionary contains a parameter with the value. If not, then the value is generated based on valid values for the parameter. For the second parameter the user that wants to generate a random float number. The parameter (EVSEID) that the user did not mention but is mandatory for the message will have a valid value from the SECC default dictionary.

The configuration file I described is listed in Code listing 19. A sample of the generated messages based on the mentioned configuration file is shown in the figure (Figure 4.20). The demonstration is recorded in the video “v2gevil_fuzzer_config_example_2.mp4”.

Code listing 19 V2GEvil – fuzzer config file, example 2

```
# ev_fuzzer_example_2.toml
# Configuration for fuzzing messages
# Only the messages listed here will be fuzzed
[supportedAppProtocolRes]
# Mandatory parameters according to ISO15118 are: ResponseCode.
# User defined the required parameters for fuzzing: ResponseCode, SchemaID
RequiredParams = ["ResponseCode", "SchemaID"]

# ResponseCode is enum, so it is a string from the list of possible values
[supportedAppProtocolRes.ResponseCode]
Mode = "base64"
# User does not specify mode for SchemaID, so it will be random mode

[SessionSetupRes]
# Mandatory parameters according to ISO15118 are: ResponseCode, EVSEID.
# User defined that required parameters for fuzzing are:
#   ResponseCode, EVSETimeStamp
RequiredParams = ["ResponseCode", "EVSETimeStamp"]

# User defined that parameter should be chosen
# from default dict or generated
[SessionSetupRes.ResponseCode]
Mode = "valid"

# EVSETimestamp is type: long according to ISO15118
[SessionSetupRes.EVSETimeStamp]
Mode = "float"
```

```
└─$ v2gevil modules-tools fuzz-EV --mode config --config-filename ev_fuzzer_example_2.toml
********************************************************************************
FUZZING METHOD START for FuzzerSupportedAppProtocolRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK_SuccessfulNegotiation', 'SchemaID': '1'}
WARNING  2024-01-10 13:21:54,155                                  fuzz_msg_general.py:72
         src.v2gevil.fuzzer.fuzz_msg_general: Required parameter
         SchemaID is not specified in config for fuzzing class
         FuzzerSupportedAppProtocolRes.
WARNING  2024-01-10 13:21:54,159                                  fuzz_msg_general.py:77
         src.v2gevil.fuzzer.fuzz_msg_general: Fuzzing with random
         mode
WARNING  2024-01-10 13:21:54,161 src.v2gevil.fuzzer.fuzz_datatypes:   fuzz_datatypes.py:97
         Invalid fuzzing mode for parameter with type
         responseCodeType, using random mode.
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': -6173253531517357357, 'SchemaID': -1.8172303438773658e+17}
FUZZING METHOD END for FuzzerSupportedAppProtocolRes
********************************************************************************
FUZZING METHOD START for FuzzerSessionSetupRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'EVSEID': 'FRA23E45B78C', 'EVSETimeStamp': 1700593914}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': 'OK', 'EVSEID': 'FRA23E45B78C', 'EVSETimeStamp': 7.002287080783344e+18}
FUZZING METHOD END for FuzzerSessionSetupRes
********************************************************************************
```

Figure 4.20 V2GEvil – fuzz-EV config mode, example 2

4.8.3.3 Example 3

The last sample is to demonstrate how to work with and define fuzzing messages that contain complex data types parameters. For this demonstration I have selected a message that contains these parameters, namely “ServiceDiscoveryRes”.

The content of the configuration file that contains the complex parameters is in Code listing 20. The user can specify which subparameters will be fuzzed using “RequiredParams” for the complex parameter. The complex parameters in the example are “PaymentOptionList” and “ChargeService”. For the former, the mode for the subparameter is specifically defined (“PaymentOption”).

The second mentioned complex parameter contains another complex parameter “SupportedEnergyTransferMode” and only has a simple type subparameter. So, the user defined a fuzzing mode for this simple type parameter. For other parameters in “RequiredParams” for “ChargeService” the user has not specified additional information and, therefore, all mentioned parameters will be fuzzed. For complex data-type parameters all mandatory subparameters will be fuzzed. For simple data-type parameters, the fuzzing mode will be selected randomly. The output of the generated message is shown in the picture (Figure 4.21). The demonstration is also captured in a video called “v2gevil_fuzzer_config_example_3.mp4”.

For parameters that contain other parameters, the user can specify only “end parameters”. Thus, the user does not need to specify a sequential hierarchy. An example of such a file is shown in Code listing 21. In this file, I have excluded the “ServiceID” and “ServiceName” parameters, so the tool does not print a warning. The output of the tool that used this configuration file for the fuzzer is shown in the picture (Figure 4.22).

4.8.4 Mode “message”

The message mode allows the user to use an existing configuration file, but at the same time to select a single message. This is advantageous, for example, because if user wants to test only a specific message, he does not have to create a new configuration file.

Code listing 20 V2GEvil – fuzzer config file, example 3

```
# ev_fuzzer_example_3.toml
# Configuration for fuzzing messages
# Only the messages listed here will be fuzzed


[ServiceDiscoveryRes]
RequiredParams = ["ResponseCode", "PaymentOptionList", "ChargeService"]

[ServiceDiscoveryRes.ResponseCode]
Mode = "random"

# not the end parameter, no Mode
[ServiceDiscoveryRes.PaymentOptionList]
RequiredParams = ["PaymentOption"]

# end parameter => Mode
[ServiceDiscoveryRes.PaymentOptionList.PaymentOption]
Mode = "string"

[ServiceDiscoveryRes.ChargeService]
RequiredParams = ["ServiceID", "ServiceName", "SupportedEnergyTransferMode"]

[ServiceDiscoveryRes.ChargeService.SupportedEnergyTransferMode]
RequiredParams = ["EnergyTransferMode"]

# split into two lines due to latex layout
# must be one line to use
[ServiceDiscoveryRes.ChargeService.SupportedEnergyTransferMode.
EnergyTransferMode]
Mode = "random"
```

```
└─$ v2gevil modules-tools fuzz-EV --mode config --config-filename ev_fuzzer_example_3.toml
********************************************************************************
FUZZING METHOD START for FuzzerServiceDiscoveryRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'PaymentOptionList': {'PaymentOption': ['Contract', 'ExternalPayment']}, 'ChargeServ
ice': {'ServiceID': 1, 'ServiceName': 'AC_DC_Charging', 'ServiceCategory': 'EVCharging', 'FreeService': True
, 'SupportedEnergyTransferMode': {'EnergyTransferMode': ['AC_three_phase_core', 'DC_extended']}}, 'ServiceLi
st': {'Service': [{'ServiceID': 3, 'ServiceName': 'Fast Internet', 'ServiceCategory': 'Internet', 'FreeServi
ce': True}, {'ServiceID': 2, 'ServiceName': 'Certificate', 'ServiceCategory': 'ContractCertificate', 'FreeSe
rvice': True}]}}
WARNING  2024-01-10 14:33:02,846 src.v2gevil.fuzzer.fuzz_datatypes: Required parameter fuzz_datatypes.py:160
         ServiceID is not specified in config for fuzzing method .
WARNING  2024-01-10 14:33:02,850 src.v2gevil.fuzzer.fuzz_datatypes: Fuzzing with       fuzz_datatypes.py:165
         random mode
WARNING  2024-01-10 14:33:02,851 src.v2gevil.fuzzer.fuzz_datatypes: Required parameter fuzz_datatypes.py:160
         ServiceName is not specified in config for fuzzing method .
WARNING  2024-01-10 14:33:02,852 src.v2gevil.fuzzer.fuzz_datatypes: Fuzzing with       fuzz_datatypes.py:165
         random mode
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': -1245684878168754478, 'PaymentOptionList': {'PaymentOption': ['lBMVqtVepzyKlwmyGYgdYbNvtqL
lLQHkqPMwNpaSHwsnrcDqQEZbnwTmNAHypgFdMrBNxppTWKGLrfgqqazzKxEMviEtsu']}, 'ChargeService': {'SupportedEnergyTr
ansferMode': {'EnergyTransferMode': [-6147219206623117840]}, 'ServiceID': 26657.3914348841, 'ServiceName': 3
111733653098094503}, 'ServiceList': {'Service': [{'ServiceID': 3, 'ServiceName': 'Fast Internet', 'ServiceCa
tegory': 'Internet', 'FreeService': True}, {'ServiceID': 2, 'ServiceName': 'Certificate', 'ServiceCategory':
 'ContractCertificate', 'FreeService': True}]}}
FUZZING METHOD END for FuzzerServiceDiscoveryRes
```

Figure 4.21 V2GEvil – fuzz-EV config mode, example 3

Code listing 21 V2GEvil – fuzzer config file, only end parameters

```
# ev_fuzzer_example_3.toml
# Configuration for fuzzing messages
# Only the messages listed here will be fuzzed

[ServiceDiscoveryRes]
RequiredParams = ["ResponseCode", "PaymentOptionList", "ChargeService"]

[ServiceDiscoveryRes.ResponseCode]
Mode = "random"

# end parameter => Mode
[ServiceDiscoveryRes.PaymentOptionList.PaymentOption]
Mode = "string"

# split into two lines due to latex layout
# must be one line to use
[ServiceDiscoveryRes.ChargeService.SupportedEnergyTransferMode.
EnergyTransferMode]
Mode = "random"
```

```
└─$ v2gevil modules-tools fuzz-EV --mode config --config-filename ev_fuzzer_example_3.toml
********************************************************************************
FUZZING METHOD START for FuzzerServiceDiscoveryRes
BEFORE fuzzing msg_fuzz_dict:
 {'ResponseCode': 'OK', 'PaymentOptionList': {'PaymentOption': ['Contract', 'ExternalPayment']}, 'ChargeServ
ice': {'ServiceID': 1, 'ServiceName': 'AC_DC_Charging', 'ServiceCategory': 'EVCharging', 'FreeService': True
, 'SupportedEnergyTransferMode': {'EnergyTransferMode': ['AC_three_phase_core', 'DC_extended']}}, 'ServiceLi
st': {'Service': [{'ServiceID': 3, 'ServiceName': 'Fast Internet', 'ServiceCategory': 'Internet', 'FreeServi
ce': True}, {'ServiceID': 2, 'ServiceName': 'Certificate', 'ServiceCategory': 'ContractCertificate', 'FreeSe
rvice': True}]}}
AFTER FUZZING msg_fuzz_dict:
 {'ResponseCode': -7186660106825221381, 'PaymentOptionList': {'PaymentOption': ['JpotXyqBJqUYPNyDYflFdMvEzhn
WjrcqJChNzovNQVvYOofazfmZOLLfxglGXCFfsThNnxPbzfdXwwBxXpCiUkCehWgqp']}, 'ChargeService': {'SupportedEnergyTra
nsferMode': {'EnergyTransferMode': ['tlUkuDLHjwFwgBEVQpALZRoemcrxBarJeXBYhiNXvJSnwZermONuegDMGcMVFkJj']}, 'S
erviceID': -6.358747855209623e+18, 'ServiceCategory': 3637494057923284188, 'FreeService': 128717657232038405
}, 'ServiceList': {'Service': [{'ServiceID': 3, 'ServiceName': 'Fast Internet', 'ServiceCategory': 'Internet
', 'FreeService': True}, {'ServiceID': 2, 'ServiceName': 'Certificate', 'ServiceCategory': 'ContractCertific
ate', 'FreeService': True}]}}
FUZZING METHOD END for FuzzerServiceDiscoveryRes
********************************************************************************
```

Figure 4.22 V2GEvil – fuzzer config file, only end parameters

```
└─$ v2gevil modules-tools fuzz-EV --mode message --message-name ServiceDiscoveryRes
WARNING  2024-01-10 09:43:29,842 src.v2gevil.fuzzer.fuzz_datatypes: Not  fuzz_datatypes.py:159
         all required parameters are specified for fuzz in method .
WARNING  2024-01-10 09:43:29,846 src.v2gevil.fuzzer.fuzz_datatypes: Not  fuzz_datatypes.py:159
         all required parameters are specified for fuzz in method .
WARNING  2024-01-10 09:43:29,848 src.v2gevil.fuzzer.fuzz_datatypes: Not  fuzz_datatypes.py:159
         all required parameters are specified for fuzz in method .
WARNING  2024-01-10 09:43:29,849 src.v2gevil.fuzzer.fuzz_datatypes: Not  fuzz_datatypes.py:159
         all required parameters are specified for fuzz in method .
Station configuration:
Interface: eth_station
IPv6 address: fe80::d237:45ff:fe88:b12b
Protocol: b'\x00'
SDP port: 15118
TCP port: 54497
TLS flag: False
Accept security: True
Charging mode: AC
Validate flag for model_dump/construct: False
Cert PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/cpoCertChain.pem
Keyfile PATH: /home/v2g/V2G/repos/V2GEvil/src/v2gevil/station/certs/seccLeaf.key
SDP server started
```

Figure 4.23 V2GEvil – modules-tools, fuzz-EV with specific message “ServiceDiscoveryRes”, part 1

In this mode, the user needs to provide the name of the requested message to the tool using the “message-name” option. No default message name is set, the user must set it themselves.

In this mode, the fuzzer then finds the desired message by name in the configuration file and performs the fuzzing according to the specified configuration. The explanation of the configuration file is the same as in the previous section, so I don’t explain it here.

Below is an example of how to use fuzzing in “message” mode. First, I chose the message I want to fuzz. In this case it was “ServiceDiscoveryRes”, then I edited the default configuration file for this message. Next, I ran the fuzzer in “message” mode and passed “message-name” as “ServiceDiscoveryRes”. I didn’t specify the configuration file because I work with the default one (it is loaded automatically). The progress of the tool can be seen in the pictures (Figure 4.23, Figure 4.24 and Figure 4.25). The caused error in EVCC can be seen in the picture (Figure 4.26). The first figure (Figure 4.23) shows the start of the tool, where the SECC configuration is listed and the user is informed by a warning that he did not specify all mandatory parameters. The normal V2G communication (without fuzzed messages) can be seen in the second picture (Figure 4.24). The content of the fuzzed message and the detection of the communication termination can be seen in the third figure (Figure 4.25)).

Fuzzing “ServiceDiscoveryRes” caused an error on the EVCC side when processing this mes-

```
SDP server started
SDP server is running in while loop
Plain TCP, Connected by:  ('fe80::d237:45ff:fe88:b12a', 38792, 0, 12)
TCP server loop ended after connection established
TCP server connection established
--------------------------------------------------------------------------------
Received from client: 01fe8001000000248000ebab9371d34b9b79d189a98989c1d191d191818999d26b9b3a23
2b30020000040040
{'AppProtocol': [{'ProtocolNamespace': 'urn:iso:15118:2:2013:MsgDef', 'VersionNumberMajor': 2,
 'VersionNumberMinor': 0, 'SchemaID': '1', 'Priority': 1}]}

Created response object:
{'ResponseCode': 'OK_SuccessfulNegotiation', 'SchemaID': '1'}
--------------------------------------------------------------------------------
Received from client: 01fe80010000000e8098004011d01b40dd1622c4ac00
{'Header': {'SessionID': '00'}, 'Body': {'SessionSetupReq': {'EVCCID': 'D0374588B12B'}}}
Created response object:
{'Header': {'SessionID': '00'}, 'Body': {'SessionSetupRes': {'ResponseCode': 'OK', 'EVSEID': '
FRA23E45B78C', 'EVSETimeStamp': 1700593914}}}
--------------------------------------------------------------------------------
```

Figure 4.24 V2GEvil – modules-tools, fuzz-EV with specific message “ServiceDiscoveryRes”, part 2

```
--------------------------------------------------------------------------------
Received from client: 01fe800100000000068098004011b8
{'Header': {'SessionID': '00'}, 'Body': {'ServiceDiscoveryReq': {}}}
Created response object:
{'Header': {'SessionID': '00'}, 'Body': {'ServiceDiscoveryRes': {'ResponseCode': -474379419508
8843863, 'PaymentOptionList': {'PaymentOption': [2557011587430418181]}, 'ChargeService': {'Ser
viceID': 1261313794055509673, 'ServiceName': -631972600500673573, 'ServiceCategory': 'YYhpZimb
bvQAkakMIhaavNkAJVpAAQWbAzjBjuWNEXbPGOgpdMRCaODNIWeSEWETxoLUsltmbzAfXgGvSTxPsKuCyKbwgvJZi', 'F
reeService': 'wcjuqlnQxnXkFHbedUfjSkPUnhhGc', 'SupportedEnergyTransferMode': {'EnergyTransferM
ode': ['CYZQyPpaMfUyUzIhgGzxmPAnGNmCjkkfIaPmHKqhuhXoCskkDmesCNNqmPfSUSuvDEvhtRGdD']}}, 'Servic
eList': {'Service': [{'ServiceID': 6.078804982723869e+18, 'ServiceName': 'FvVKbfXbJXSrdiTXQKPE
CeIAMHXXGSlZyMpqCTXsTYyodoReAhbfb', 'ServiceCategory': 503402615467551945, 'FreeService': -588
9312028921686995}]}}}}
TCP connection closed
SDP server is running in while loop
```

Figure 4.25 V2GEvil – modules-tools, fuzz-EV with specific message “ServiceDiscoveryRes”, part 3

```
INFO     2024-01-10 09:47:25,683 - iso15118.shared.exi_codec (245): Message to encode (ns=urn:is
o:15118:2:2013:MsgDef): {"V2G_Message": {"Header": {"SessionID": "00"}, "Body": {"SessionSetupR
eq": {"EVCCID": "D0374588B12B"}}}}
INFO     2024-01-10 09:47:25,791 - iso15118.shared.comm_session (428): Sent SessionSetupReq
INFO     2024-01-10 09:47:25,791 - iso15118.shared.states (139): Entered state SessionSetup
DEBUG     2024-01-10 09:47:25,791 - iso15118.shared.states (143): Waiting for up to 20.0 s
INFO     2024-01-10 09:47:25,941 - iso15118.shared.exi_codec (299): Decoded message (ns=urn:iso:
15118:2:2013:MsgDef): {"V2G_Message":{"Header":{"SessionID":"00"},"Body":{"SessionSetupRes":{"R
esponseCode":"OK","EVSEID":"FRA23E45B78C","EVSETimeStamp":1700593914}}}}
INFO     2024-01-10 09:47:25,941 - iso15118.shared.comm_session (235): SessionSetupRes received
INFO     2024-01-10 09:47:25,942 - iso15118.shared.exi_codec (245): Message to encode (ns=urn:is
o:15118:2:2013:MsgDef): {"V2G_Message": {"Header": {"SessionID": "00"}, "Body": {"ServiceDiscov
eryReq": {}}}}
INFO     2024-01-10 09:47:26,036 - iso15118.shared.comm_session (428): Sent ServiceDiscoveryReq
INFO     2024-01-10 09:47:26,036 - iso15118.shared.states (139): Entered state ServiceDiscovery
DEBUG     2024-01-10 09:47:26,036 - iso15118.shared.states (143): Waiting for up to 2.0 s
ERROR     2024-01-10 09:47:26,194 - iso15118.shared.comm_session (222): EXIDecodingError (Except
ion): Exception: javax.xml.bind.UnmarshalException
Traceback (most recent call last):
  File "/home/v2g/V2G/repos/iso15118/iso15118/shared/exi_codec.py", line 285, in from_exi
    exi_decoded = self.exi_codec.decode(exi_message, namespace)
```

Figure 4.26 EVCC error cause by fuzzed response message

sage. Again, this was an error that occurred during the decoding of the EXI data of the received message. However, this time it was a different error, namely
"javax.xml.bind.UnmarshalException" (shown in Figure 4.26.

"Since a javax.xml.bind.Unmarshaller parses XML and does not support any flags for disabling XXE, it's imperative to parse the untrusted XML through a configurable secure parser first, generate a source object as a result, and pass the source object to the Unmarshaller." [79]

It implies that if EVCC has this parsing configured incorrectly, then it is possible for XXE to abuse this setting for Unmarshaller. However, this detailed information can only be obtained if the attacker has access to the EVCC logs.

If the user does not have access to the tested EVCC, then it can only suggest that the communication was terminated after sending a malicious message. Another indicator that an error occurred in the EVCC while processing the malicious response is that the communication was terminated without using "SessionStopReq". From the attacker's point of view, this information is important because then the attacker knows which message and possibly parameters to focus on. If the attacker (tester) also has access to the EVCC, then he can investigate the vulnerability and try to exploit it.

The above description is recorded on video and is available on the attached media in the "video" folder under name "v2gevil_fuzz_message.mp4".

Chapter 5
Conclusion

In this thesis I have focused on the security issues of modern electric vehicles, in particular the security aspects associated with the On-Board Charging (OBC) port. The most important goal was to implement a tool that allows the enumeration and partial evaluation of the on board charging port on the European electric vehicle.

Sub-objectives which led to the final implementation were as follows. Examine existing research in the areas of OBC, OBC safety and advanced vehicle architecture. Analyze the functionality of the OBC interface and determine the scope and network layer protocols that the security tool should target.

This was followed by the implementation of the tool, which was based on the research conducted in the theoretical part. The last objective, which is related to the implementation of the tool, is to prepare documentation on the usage of the tool. In this case, this paper is considered as documentation. Therefore, in this thesis I describe in detail the steps to make the tool operational and examples of how to use the tool. I have thus summarized the objectives of the thesis and will further outline how I proceeded in achieving them.

In this thesis focusing on electric vehicles, I first started with a research in which the main element is the electric vehicle and its charging. At the beginning I focused on the in-vehicle network in general. I then expanded this topic by exploring the in-vehicle network for electric vehicles and conducted a discussion regarding how an EV differs from a conventional modern combustion engine vehicle.

I further built on the foundations of the in-vehicle network description in the section on attack vectors for vehicles. In this section, I surveyed the attack surface for a conventional vehicle and for an electric vehicle. The most important thing that results from the attack surface exploration is that the EV offers a publicly accessible interface in the form of a charging inlet.

In the next section I cover charging and all the aspects that the charging process entails. This includes charging methods and types, charging interfaces used, and standards relevant to the EVs charging process. A complete summary of the use of standards and charging interfaces and other information related to charging is defined in the CCS (Combined Charging System). The CCS also aims to unify the charging standards used. The CCS also defines the type of vehicle inlet to be used in the EU. For on board charging this is specifically the Type 2 and Combo 2 inlet, both inputs are compatible with the Type 2 connector (designed for AC charging). It is important to mention here that on board charging is just another term for AC charging. This term is used because the conversion from AC to DC is done inside the vehicle using a device called an on board charger.

Since the aim of the tool is to target an electric vehicle available on the EU market, I focused mainly on EU relevant standards in the charging section. Based on this analysis regarding the standards used, the supervisor and I agreed that the core standard for the development of the

tool is ISO 15118.

This standard defines all the requirements for high level communication during which complex data is exchanged. The supervisor and I agreed that the tool should focus on 3-7. ISO/OSI layer. This decision was based on the fact that there are devices that will provide the first two layers. The higher layers are described in ISO 15118-2 and therefore I followed this standard during implementation. The result is that the tool allows communication according to ISO 15118-2 with EVCC. In terms of the standard, the SECC part is implemented in the tool, this controller is usually located in the EVSE.

Later in the theoretical part I described how the communication between EV and EVSE takes place during charging. I explained this communication from the point where the EV is connected to the EVSE by the charging cable. The mentioned HLC is possible for 3 mode and mandatory for 4 charging mode. I list the modes and their properties in the chapter called Charging. Here it is sufficient to notice that 3 mode is AC charging and 4 mode is DC charging. Within ISO 15118 the HLC communication is defined in the same way for both modes, only some messages differ. This allows my tool to be used for the off board charging port (DC charging) as long as the EVCC communicates according to ISO 15118.

At the end of the theoretical part, I explained the concept of OBC and discuss possible security vulnerabilities that may occur during OBC communication between EV and EVSE. Following the whole theoretical part and the knowledge gained regarding European EV charging, I summarised important information for the implementation of the tool. Specifically, I provide a reason to justify the choice of standards and the selection of ISO/OSI layers.

The theoretical part is followed by a description of the necessary V2G setup. In this description, the hardware and firmware needed to simulate a real EVCC and SECC are presented. Since the real V2G communication is using PLC technology to transfer data between the EVSE and the EV using a charging cable, I used a boards that provide PLC communication transfer. Furthermore, according to ISO 15118-3, I needed to provide the physical and data layers of the ISO/OSI model. In my case this is provided by hardware and firmware (HPGP boards and modules), which supports the SLAC process according to the HPGP specification. Further details are given in the chapter V2G setup. Next, I needed to provide a reference implementation of EVCC against which the tool can be tested. This is provided by the Josev software, specifically the iso15118 part. Josev also implements the SECC part, so I was able to first monitor the communication using Wireshark and compare it against the ISO 15118 standard.

Based on the acquired knowledge of how communication works on a real example, I was able to start implementing my tool. Most important of all was the implementation of V2GTP support, i.e. packet generation, processing and decoding. I implemented this functionality in the "v2gtp" module in the tool.

Then I needed to verify the implementation by capturing the real communication, so I implemented the "sniffer" module. With this module it is possible to capture and decode the V2G communication between EVCC and SECC. This, combined with a comparison with ISO 15118-2, verified that the V2GTP implementation is correct.

Next, I implemented the "messages" module, which ensures correct encoding and decoding for V2G EXI messages. In this module one of the most important parts is the implementation of XML Schema according to ISO 15118, where V2G messages are defined.

Thanks to the previous module combined with the V2GTP implementation, I could implement SECC. SECC is implemented in the "station" module and this module provides complete handling of communication with EVCC.

When I had implemented the previous core, I could implement modules dealing with enumeration and partial evaluation. The first of the modules dedicated to testing is the "enumerator" module. This module provides enumeration of EVCC, for example supported application protocols or enumeration of TLS version or TLS cipher suites.

The second module for security testing is "fuzzer". This module provides the user with the ability to create non-valid and malicious data that is sent to EVCC in V2G responses. The

V2GEvil tool is implemented to be modular, so both modules use the “station” implementation and only add functionality for the “station” module.

This was a short summary of how I implemented the tool and what functionality it contains. In addition, I focused on a detailed description of the usage of the tool in chapter called Usage of the tool. Thus, I met the requirement for detailed documentation on the usage of the tool.

Unfortunately, the implemented tool could not be tested on a real vehicle, but for the mentioned V2G testing setup it works and meets the objectives defined at the beginning of the thesis. For a real use case (EV charging port), the use of one HPGP board and the addition of a low level communication implementation using PWM to enforce HLC (according to IEC 61851) should be sufficient. Consequently, for SLAC it should be sufficient to use an HPGP board with the appropriate firmware, as I used in my V2G setup. Then, after successfully building the data link layer using the SLAC, the communication would be handled by my tool. Of course, the correct connection of the pins of the charging connector and the HPGP device, namely the CP and PE pins, is also essential. The communication is done through the CP circuit and uses PLC technology.

In future development I plan to extend the tool further, for example implementing EVCC and thus allowing SECC testing. As stated in many publications, it is the EVSE that provides more space for mistakes in impelemnation, as it generally contains more services.

The main benefit of the tool is that it provides automatic functions compared to, for example, V2GInjector. Compared to commercial tools, my tool is freely available on GitHub. In terms of available features I cannot compare with commercial tools due to their availability (price).

Bibliography

1. *Smart charging* [online]. Vector Informatik GmbH [visited on 2023-11-24]. Available from: https://www.vector.com/int/en/products/solutions/e-mobility/#c50821.
2. ISO 15118-1:2019. *Road vehicles – Vehicle to grid communication interface – Part 1: General information and use-case definition*. Geneva, CH: International Organization for Standardization, 2019-04.
3. LESJAK, Žiga. *ISO 15118 Standard (Plug & Charge Protocol)* [online]. Tridens d.o.o. [visited on 2023-12-04]. Available from: https://tridenstechnology.com/iso-15118-standard.
4. KHATRI, Narayan; SHRESTHA, Rakesh; NAM, Seung Yeob. Security Issues with In-Vehicle Networks, and Enhanced Countermeasures Based on Blockchain. *Electronics*. 2021, vol. 10, no. 8, p. 893. ISSN 2079-9292. Available from DOI: 10.3390/electronics10080893.
5. TUOHY, Shane; GLAVIN, Martin; JONES, Edward; TRIVEDI, Mohan; KILMARTIN, Liam. Next generation wired intra-vehicle networks, a review. In: *2013 IEEE Intelligent Vehicles Symposium (IV)*. 2013, pp. 777–782. Available from DOI: 10.1109/IVS.2013.6629561.
6. SAGSTETTER, Florian; LUKASIEWYCZ, Martin; STEINHORST, Sebastian; WOLF, Marko; BOUARD, Alexandre; HARRIS, William R.; JHA, Somesh; PEYRIN, Thomas; POSCHMANN, Axel; CHAKRABORTY, Samarjit. Security challenges in automotive hardware/software architecture design. In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2013, pp. 458–463. Available from DOI: 10.7873/DATE.2013.102.
7. BRUNNER, Stefan; RODER, Jurgen; KUCERA, Markus; WAAS, Thomas. Automotive E/E-architecture enhancements by usage of ethernet TSN. In: *2017 13th Workshop on Intelligent Solutions in Embedded Systems (WISES)*. 2017, pp. 9–13. Available from DOI: 10.1109/WISES.2017.7986925.
8. RISHVANTH, D.; KALIYAPERUMAL, Ganesan. Design of an in-vehicle network (Using LIN, CAN and FlexRay), gateway and its diagnostics using vector CANoe. *American Journal of Signal Processing*. 2012, vol. 1, pp. 40–45. Available from DOI: 10.5923/j.ajsp.20110102.07.
9. SCHMIDT, E. G.; ALKAN, M.; SCHMIDT, K.; YÜRÜKLÜ, E.; KARAKAYA, U. Performance evaluation of FlexRay/CAN networks interconnected by a gateway. In: *International Symposium on Industrial Embedded System (SIES)*. 2010, pp. 209–212. Available from DOI: 10.1109/SIES.2010.5551395.

10. SAGSTETTER, Florian; LUKASIEWYCZ, Martin; STEINHORST, Sebastian; WOLF, Marko; BOUARD, Alexandre; HARRIS, William R.; JHA, Somesh; PEYRIN, Thomas; POSCHMANN, Axel; CHAKRABORTY, Samarjit. *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Security challenges in automotive hardware/-software architecture design. 2013. Available from DOI: 10.7873/DATE.2013.102. Image from Fig. 1.

11. SHANKER, Shreejith. *Enhancing Automotive Embedded Systems with FPGAs - Scientific Figure on ResearchGate* [online]. [N.d.]. [visited on 2023-12-07]. Available from: https://www.researchgate.net/figure/Typical-in-vehicle-network-architecture-in-a-modern-car_fig2_305499872. Figure 2.2: Typical in-vehicle network architecture in a modern car.

12. ZHANG, Haichun; MENG, Xu; ZHANG, Xiong; LIU, Zhenglin. CANsec: A Practical in-Vehicle Controller Area Network Security Evaluation Tool. *Sensors*. 2020, vol. 20, no. 17. ISSN 1424-8220. Available from DOI: 10.3390/s20174900. Figure 1. Architecture of the IVNs.

13. PATSAKIS, Constantinos. *External Monitoring Changes in Vehicle Hardware Profiles: Enhancing Automotive Cyber-Security - Scientific Figure on ResearchGate* [online]. [N.d.]. [visited on 2023-12-07]. Available from: https://www.researchgate.net/figure/Paradigm-of-the-automotive-network-architecture_fig1_335191619. Figure name is Paradigm of the automotive network architecture.

14. RATHORE, Rajkumar Singh; HEWAGE, Chaminda; KAIWARTYA, Omprakash; LLORET, Jaime. In-Vehicle Communication Cyber Security: Challenges and Solutions. *Sensors*. 2022, vol. 22, no. 17. ISSN 1424-8220. Available from DOI: 10.3390/s22176679. Figure 3. In-Vehicle Network Architecture with Automotive Protocols.

15. FALCH, Martin. *CAN Bus Explained - A Simple Intro [2023]* [online]. CSS Electronics [visited on 2023-12-07]. Available from: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial.

16. *The CAN Bus Protocol Tutorial* [online]. Kvaser [visited on 2023-12-07]. Available from: https://www.kvaser.com/can-protocol-tutorial/.

17. FALCH, Martin. *LIN Bus Explained - A Simple Intro [2023]* [online]. CSS Electronics [visited on 2023-12-07]. Available from: https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial.

18. *Introduction to the LIN bus* [online]. Kvaser [visited on 2023-12-07]. Available from: https://www.kvaser.com/about-can/can-standards/linbus/. Figure 3: Example of LIN frame.

19. GRZEMBA, Andreas et al. *MOST - The Automotive Multimedia Network*. 2nd ed. Franzis Verlag GmbH, 2011. ISBN 978-3-645-65061-8. Available also from: https://www.mostcooperation.com/specifications/.

20. ZHAO, Lin; HE, Feng; LI, Ershuai; LU, Jun. Comparison of Time Sensitive Networking (TSN) and TTEthernet. In: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. 2018, pp. 1–7. Available from DOI: 10.1109/DASC.2018.8569454.

21. HANK, Peter; MÜLLER, Steffen; VERMESAN, Ovidiu; VAN DEN KEYBUS, Jeroen. Automotive Ethernet: In-vehicle networking and smart mobility. In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2013, pp. 1735–1739. Available from DOI: 10.7873/DATE.2013.349.

22. HANK, Peter; MÜLLER, Steffen; VERMESAN, Ovidiu; VAN DEN KEYBUS, Jeroen. Automotive Ethernet: In-vehicle networking and smart mobility. 2013, pp. 1735–1739. Available from DOI: 10.7873/DATE.2013.349. Fig. 2 Ethernet backbone in domain architecture.

23. DIMITRIADOU, Konstantina et al. Current Trends in Electric Vehicle Charging Infrastructure; Opportunities and Challenges in Wireless Charging Integration. *Energies*. 2023, vol. 16, no. 4. ISSN 1996-1073. Available from DOI: 10.3390/en16042057. Figure 1. General schemes for (a) DC charging and (b) AC charging.

24. AROLE, Sreeraj. *Electric Vehicle Charging System: A bittersweet entry from Tesla* [online]. Telematics Wire, 2023-01-20 [visited on 2023-12-19]. Available from: https://www.telematicswire.net/electric-vehicle-charging-system-a-bittersweet-entry-from-tesla/.

25. *Vehicle Control Unit (VCU)* [online]. STMicroelectronics [visited on 2023-12-15]. Available from: https://www.st.com/en/applications/chassis-and-safety/vehicle-control-unit-vcu.html.

26. *Exploring AC Level 1 and Level 2 Chargers for Efficient EV Charging* [online]. EV Builders Guide [visited on 2023-12-15]. Available from: https://www.evbuildersguide.com/exploring-ac-level-1-and-level-2-chargers-for-efficient-ev-charging/.

27. LESERER, Jonas; NORTHEY, Joel. *E-Mobility – It's all about the Charging* [online]. Vector Informatik GmbH, 2019 [visited on 2023-12-15]. Available from: https://cdn.vector.com/cms/content/events/2019/VU/VU_C19_Files/021_Workshop_E-Mobility.pdf.

28. JOHNSON, John; GUPTA, Sachin. *How specialized MCUs meet on-board charger design needs* [online]. Embedded by AspenCore, 2022 [visited on 2023-12-15]. Available from: https://www.embedded.com/how-specialized-mcus-meet-on-board-charger-design-needs/.

29. PLAPPERT, Christian; ZELLE, Daniel; GADACZ, Henry; RIEKE, Roland; SCHEUERMANN, Dirk; KRAUSS, Christoph. Attack Surface Assessment for Cybersecurity Engineering in the Automotive Domain. In: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2021, pp. 266–275. Available from DOI: 10.1109/PDP52278.2021.00050. Fig. 2. Generic reference architecture for risk analysis.

30. ELKHAIL, Abdulrahman Abu; REFAT, Rafi Ud Daula; HABRE, Ricardo; HAFEEZ, Azeem; BACHA, Anys; MALIK, Hafiz. Vehicle Security: A Survey of Security Issues and Vulnerabilities, Malware Attacks and Defenses. *IEEE Access*. 2021, vol. 9, pp. 162401–162437. Available from DOI: 10.1109/ACCESS.2021.3130495.

31. KHATRI, Narayan; SHRESTHA, Rakesh; NAM, Seung Yeob. Security Issues with In-Vehicle Networks, and Enhanced Countermeasures Based on Blockchain. *Electronics*. 2021, vol. 10, no. 8. ISSN 2079-9292. Available from DOI: 10.3390/electronics10080893. Figure 3. CAN bus attack interfaces.

32. MILLER, Charlie; VALASEK, Chris. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*. 2015, vol. 2015, no. S 91, pp. 1–91.

33. CAI, Zhiqiang; WANG, Aohui; ZHANG, Wenkai; GRUFFKE, Michael; SCHWEPPE, Hendrick. 0-days & mitigations: roadways to exploit and secure connected BMW cars. *Black Hat USA*. 2019, vol. 2019, no. 39, p. 6.

34. ISO 15118-2:2014. *Road vehicles – Vehicle-to-grid communication interface – Part 2: Network and application protocol requirements*. Geneva, CH: International Organization for Standardization, 2014-04.

35. AROLE, Sreeraj. *Electric Vehicle Charging System: A bittersweet entry from Tesla* [online]. Telematics Wire, 2023-01-20 [visited on 2023-12-19]. Available from: https://www.telematicswire.net/electric-vehicle-charging-system-a-bittersweet-entry-from-tesla/. Figure 1: EV Charging System High Level Overview.

36. *The European Green Deal* [online]. European Commission [visited on 2023-12-20]. Available from: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal_en.
37. GROSSMANN, Dirk; HILD, Heiner. Smart Charging – A Key to Successful E Mobility. *Elektronik automotive*. 2014. Available also from: https://cdn.vector.com/cms/content/know-how/_technical-articles/EMobility_Smart_Charging_ElektronikAutomotive_201407_PressArticle_EN.pdf.
38. *Communication Protocols* [online]. Vector Informatik GmbH [visited on 2023-11-24]. Available from: https://www.vector.com/int/en/know-how/smart-charging/communication-protocols/#.
39. *Charging Interfaces* [online]. Vector Informatik GmbH [visited on 2023-12-20]. Available from: https://www.vector.com/int/en/know-how/smart-charging/charging-interfaces/#.
40. *Charging Types and Methods* [online]. Vector Informatik GmbH [visited on 2023-12-22]. Available from: https://www.vector.com/int/en/know-how/smart-charging/charging-types-and-methods/#.
41. MULLER, Joann. *Electric car battery swapping gets a reboot* [online]. Axios [visited on 2023-12-23]. Available from: https://www.axios.com/2023/05/18/electric-car-battery-swapping.
42. *AC / DC Charging* [online]. EV EXPERT s.r.o. [visited on 2023-12-22]. Available from: https://www.evexpert.eu/eshop1/knowledge-center/ac-dc-charging-electromobil-current-alternating-direct.
43. CHAUHAN, Sheeba. *Conductive Charging of Electrified Vehicles(EVs)-Challenges and Opportunities* [online]. ELE Times, 2021-09-05 [visited on 2023-12-22]. Available from: https://www.eletimes.com/conductive-charging-of-electrified-vehiclesevs-challenges-and-opportunities.
44. *Charging Interfaces* [online]. Vector Informatik GmbH [visited on 2023-12-20]. Available from: https://www.vector.com/int/en/know-how/smart-charging/charging-interfaces/#. Figure of AC and DC Charging Setup.
45. *On-Board Charger* [online]. EV EXPERT s.r.o. [visited on 2023-12-22]. Available from: https://www.evexpert.eu/eshop1/knowledge-center/on-board-charger.
46. TYBEL, Michael; POPOV, Andrey; SCHUGT, Michael. Assuring Interoperability Between Conductive EV and EVSE Charging Systems. In: BÄKER, Bernard; MORAWIETZ, Lutz (eds.). *Energy Efficient Vehicles 2015*. TUDpress, 2015, pp. 150–157. ISBN 978-3-959080088. Figure 2: Charging System of EVSE and EV.
47. *Recharging systems* [online]. The European Alternative Fuels Observatory [visited on 2023-12-24]. Available from: https://alternative-fuels-observatory.ec.europa.eu/general-information/recharging-systems.
48. *DC Charging: A complete Guide to Hardware* [online]. Heliox Energy, 2022-06-04 [visited on 2023-12-24]. Available from: https://www.heliox-energy.com/blog/dc-charging-a-complete-guide-to-hardware.
49. *Combined Charging System 1.0 Specification - CCS 1.0* [online]. Carmeq GmbH, 2015. Version 1.2.1 [visited on 2023-12-26]. Available from: https://tesla.o.auroraobjects.eu/Combined_Charging_System_1_0_Specification_V1_2_1.pdf.
50. *Combined Charging System 1.0 Specification - CCS 1.0* [online]. Carmeq GmbH, 2015. Version 1.2.1 [visited on 2023-12-26]. Available from: https://tesla.o.auroraobjects.eu/Combined_Charging_System_1_0_Specification_V1_2_1.pdf. Figure 1 – Charging Interface of CCS.

51. *Design Guide for Combined Charging System V7* [online]. Charging Interface Initiative (CharIN), 2019 [visited on 2023-12-20]. Available from: https://www.charin.global/media/pages/technology/ccs-specification/42a9d61e04-1626949173/design_guide_combined_charging_system_v7.pdf.

52. RAJ, Aswinth. *Electric Vehicle On-board Chargers and Charging Stations* [online]. Circuit Digest, 2019-06-11 [visited on 2023-12-20]. Available from: https://circuitdigest.com/article/electric-vehicle-on-board-chargers-and-charging-stations.

53. *Charging Standards* [online]. Vector Informatik GmbH [visited on 2023-11-24]. Available from: https://www.vector.com/int/en/know-how/smart-charging/charging-standards/#.

54. KÜBEL, Matthias (ed.). *Design Guide for Combined Charging System* [online]. Initiative Charging Interface, 2015 [visited on 2023-12-20]. Available from: https://tesla.o.auroraobjects.eu/Design_Guide_Combined_Charging_System_V3_1_1.pdf.

55. IEC 61851-1:2010. *Electric vehicle conductive charging system – Part 1: General requirements*. Geneva, CH: International Electrotechnical Commission (IEC), 2010-11. ISBN 978-2-88912-222-6.

56. GB/T 18487.1-2023. *Electric vehicle conductive charging system—Part 1: General requirements (English Version)*. China: AQSIQ, SAC, 2023-09.

57. MÜLTIN, Marc. *What is ISO 15118?* [online]. Switch EV, 2021-10-11 [visited on 2023-12-28]. Available from: https://www.switch-ev.com/blog/what-is-iso-15118.

58. *ISO/DIS 15118-10* [online]. International Organization for Standardization [visited on 2023-12-28]. Available from: https://www.iso.org/standard/85604.html.

59. MÜLTIN, Marc. *What is ISO 15118?* [online]. Switch EV, 2021-10-11 [visited on 2023-12-28]. Available from: https://www.switch-ev.com/blog/what-is-iso-15118. The eight parts of ISO 15118 and their relation to the seven ISO/OSI layers.

60. *Interface EVSE with Combined Charging System (CCS) using OpenECU™ M560 or M580* [online]. Dana Limited [visited on 2023-12-10]. Available from: https://openecu.com/case_study/interface-evse-with-combined-charging-system-ccs-using-openecu-m560-or-m580/.

61. *ISO 15118 Protocol* [online]. Typhoon HIL Inc. [visited on 2023-12-10]. Available from: https://www.typhoon-hil.com/documentation/typhoon-hil-software-manual/References/iso15118_protocol.html#iso15118_protocol__section_rgv_zck_ctb.

62. MÜLTIN, Marc. *Webinar series - What's new in ISO 15118-20?* [online]. Switch EV, 2022-04-12 [visited on 2023-12-28]. Available from: https://www.switch-ev.com/blog/switch-to-clarity-whats-new-in-iso-15118-20.

63. ISO 15118-3:2016. *Road vehicles – Vehicle-to-grid communication interface – Part 3: Physical and data link layer requirements*. Geneva, CH: International Organization for Standardization, 2016-04.

64. *Home Plug Green PHY The Standard For In-Home Smart Grid Powerline Communications*. HomePlug Powerline Alliance, Inc., 2010. Version 1.00. Available also from: https://content.codico.com/fileadmin/media/download/datasheets/powerline-communication/plc-homeplug-green-phy/homeplug_green_phy_whitepaper.pdf.

65. MÜLTIN, Marc. ISO 15118 as the Enabler of Vehicle-to-Grid Applications. In: *2018 International Conference of Electrical and Electronic Technologies for Automotive*. 2018, pp. 1–6. Available from DOI: 10.23919/EETA.2018.8493213.

66. MÜLTIN, Marc. ISO 15118 as the Enabler of Vehicle-to-Grid Applications. In: *2018 International Conference of Electrical and Electronic Technologies for Automotive*. 2018, pp. 1–6. Available from DOI: 10.23919/EETA.2018.8493213. Fig. 2. Message sequence for an Alternating Current (AC) charging session.

67. DUDEK, Sébastien; DELAUNAY, Jean-Christophe; FARGUES, Vincent. V2G Injector: Whispering to cars and charging units through the Power-Line. In: *Proceedings of the SSTIC (Symposium sur la sécurité des technologies de l'information et des communications), Rennes, France*. 2019, pp. 5–7. Available also from: https://www.sstic.org/media/SSTIC2019/SSTIC-actes/v2g_injector_playing_with_electric_cars_and_chargi/SSTIC2019-Article-v2g_injector_playing_with_electric_cars_and_charging_stations_via_powerline-dudek.pdf.

68. *dLAN Green PHY eval board II* [online]. devolo AG [visited on 2023-05-10]. Available from: https://www.devolo.global/fileadmin/Web-Content/DE/products/bs/green-phy-eval-board-II/pictures/de/devolo_dLANGreenpHY_evalboardII_datasheets_EN.pdf.

69. *dLAN Green PHY eval board II* [online]. devolo GmbH [visited on 2023-05-10].

70. *dLAN Green PHY Module* [online]. devolo GmbH [visited on 2023-05-10]. Available from: https://www.devolo.global/fileadmin/Web-Content/DE/products/bs/green-phy-module/documents/en/devolo_dlan_green_phy_module_datasheet_EN.pdf.

71. *devolo dLAN Green PHY Module SDK* [online]. devolo [visited on 2023-05-10]. Available from: https://github.com/devolo/dlan-greenphy-sdk.

72. *dLAN Green PHY module* [online]. devolo GmbH [visited on 2023-05-10]. Available from: https://www.devolo.global/dlan-green-phy-module.

73. *ISO15118* [online]. SwitchEV [visited on 2023-05-10]. Available from: https://github.com/SwitchEV/iso15118.

74. *V2Gdecoder* [online]. FlUxIuS [visited on 2023-05-13]. Available from: https://github.com/FlUxIuS/V2Gdecoder.

75. *v2g-ws-dissectors* [online]. Argus Cyber Security Ltd. [visited on 2023-05-13]. Available from: https://github.com/geynis/v2g-ws-dissectors.

76. *wireshark-v2g - A protocol dissector for V2G communications* [online]. ChargePoint [visited on 2023-05-13]. Available from: https://github.com/ChargePoint/wireshark-v2g.

77. *Models* [online]. Pydantic [visited on 2023-07-13]. Available from: https://docs.pydantic.dev/latest/concepts/models/.

78. MÜLTIN, Marc. *The basics of Plug & Charge* [online]. Switch EV, 2020-11-15 [visited on 2023-12-28]. Available from: https://www.switch-ev.com/blog/basics-of-plug-and-charge.

79. *XML External Entity Prevention Cheat Sheet* [online]. OWASP [visited on 2023-10-11]. Available from: https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html#jaxb-unmarshaller.

Content of the attached media

- `readme.txt` ..... brief description of the contents of the attached media
- `videos` ..... directory containing demo videos of how the tool works
- `dist` ..... directory containing .whl package of the implemented tool
  - `v2gevil-1.0.0-py3-none-any.whl` ..... wheel package of the implemented tool
- `src`
  - `tool` ..... directory containing repository of developed tool
    - `V2GEvil` ..... the directory with the implemented tool
  - `thesis` ..... source thesis in LaTeX format
- `text` ..... the text of the thesis
  - `thesis.pdf` ..... text of the thesis in PDF format