Connected Car Hacking

Ignition - Automotive cybersecurity & networks bootstrap

hardwear.io

Quarkslab

Access

- Discord server: https://discord.gg/E4cqVzq2

Detail of a modern car

Same base as many decades: 4 wheels around a motor

But with significant improvements:

- **Driving assistances** (anti-collision, line detection, ESP/ABS, road sign reconnaissance, self-park…)
- **Connectivity** (GPS/LTE/Wi-Fi/Bluetooth…)
- **Onboard services** (remote control, localisation, auto-diagnostic…)
- **Autonomous driving** (Mercedes reached level 3 autonomous driving, fall 2021)

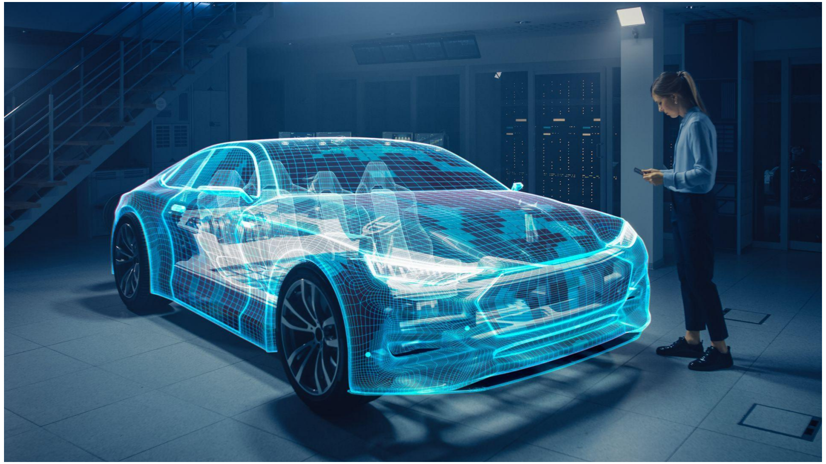


Illustration: [link]

Cores of a car: ECUs

ECU: **E**lectronic **C**ontrol **U**nit

It reads **SENSORS**, manage **ACTUATORS**, communicate with others **ECU/DEVICES/BACKEND** through wired or wireless networks. Could be one or several **MCU/SOC**.

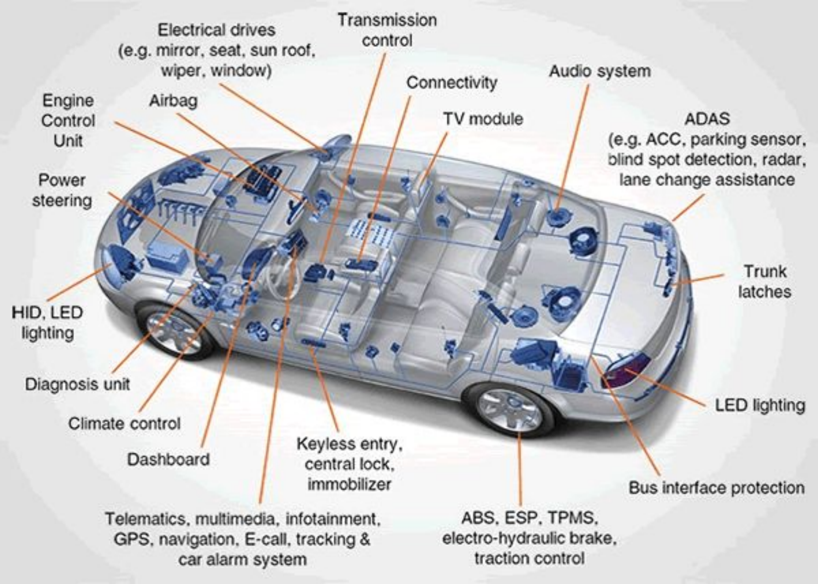


Illustration: [link] & [link]

ECUs communications: multiple internal networks

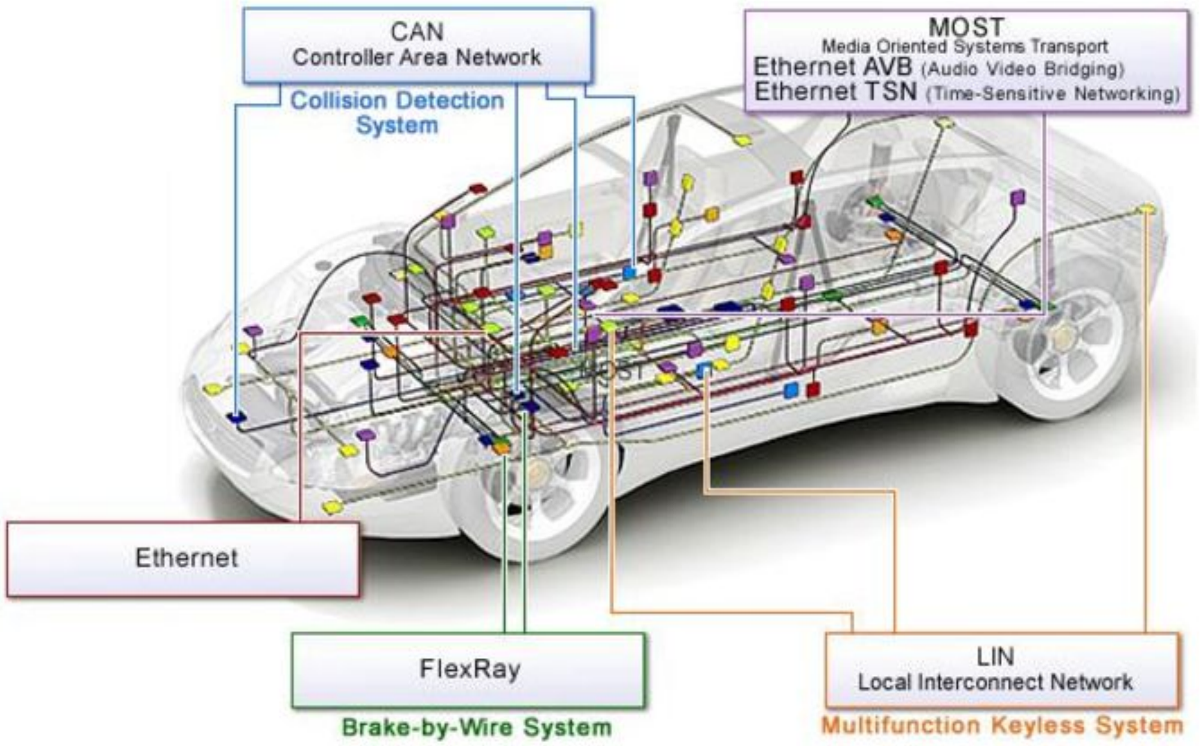


Illustration: [link]

LIN: Local Interconnect Network

- **Single wire** serial network protocol with speed up to **19.2 Kbit/s (ISO 17987)**
- **Broadcast** protocol allowing up to **16 nodes**
- **Master - slave** communication system
- **Low cost** network for **non-critical** application (locking system...)

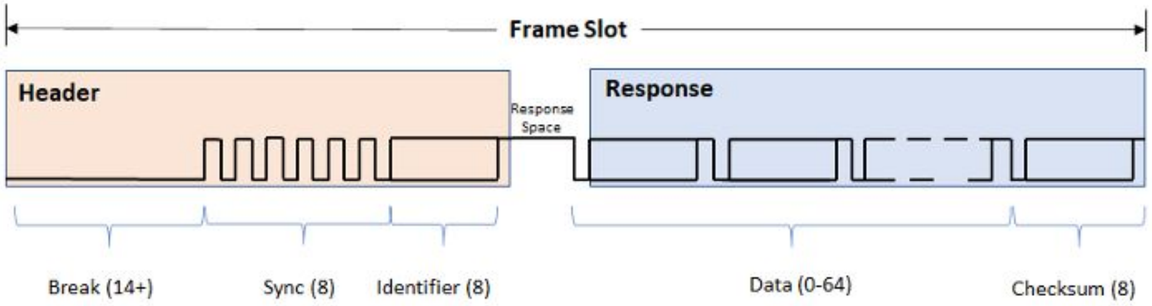


Illustration: [link]

CAN: Controller Area Network

- **Two wires** half-duplex network protocol with speed up to **1 Mbit/s (ISO 11898)**
- **Fault resistant** protocol
- **CAN-FD** (Flexible Data Rate) allows speed up to **8 Mbit/s**
- **Most commonly** used in-vehicle network to connect **ECUs**

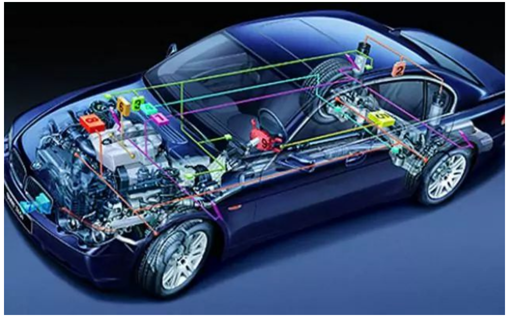


Illustration: [link]

MOST: Media Oriented System Transport

- **High speed** multimedia network with speed up to **150 Mbit/s (ISO 21806)**
- **Ring network** topology to transport audio, video, voice and data signals
- **Expensive network** using optical fibre

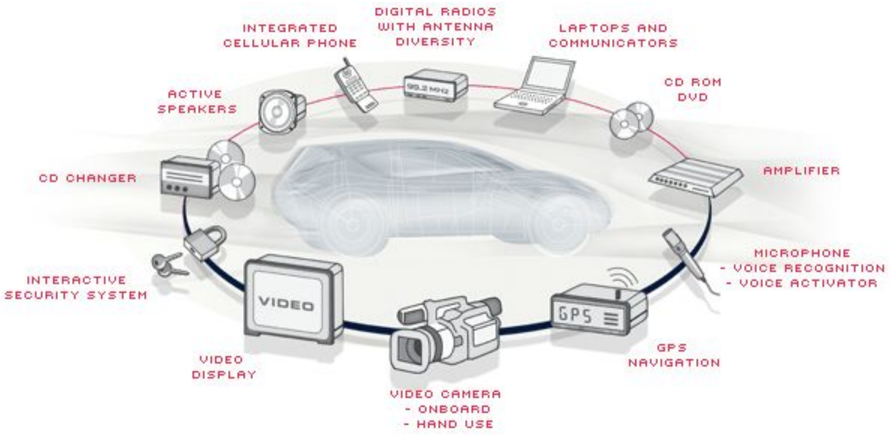


Illustration: [link]

FLEXRAY

- **High performance** 2/4 wires network protocol with speed up to **10 Mbit/s (ISO 17458)**
- **Fault resistant** and **deterministic**
- **TDMA** (Time Division Multiple Access): **Flexray** node communicates during a **scheduled time slot**
- Used for **high-performance applications** (Steer-by-wire, ADAS, …)

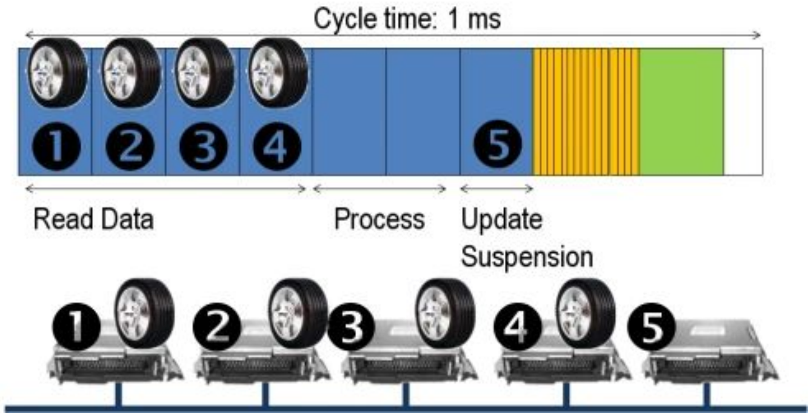


Illustration: [link]

Automotive Ethernet - 100/1000Base-T1

- **High speed** two wires network with speed up to **1 000 Mbit/s**
- **Physical layer** different from traditional ethernet (100/1000Base-T1, BroadR-Reach)
- **Less expensive** than MOST
- Handle **generic network protocols** and **automotive specifics**

Illustration: [link]

Automotive Ethernet - 10Base-T1S

- **Multidrop** ethernet network with speed up to **10 Mbit/s**
- Allows **ethernet homogeneous** network
- Use **twisted pairs** as CAN and 100/1000Base-T1 networks
- Supports at least **8 nodes** in 25m max

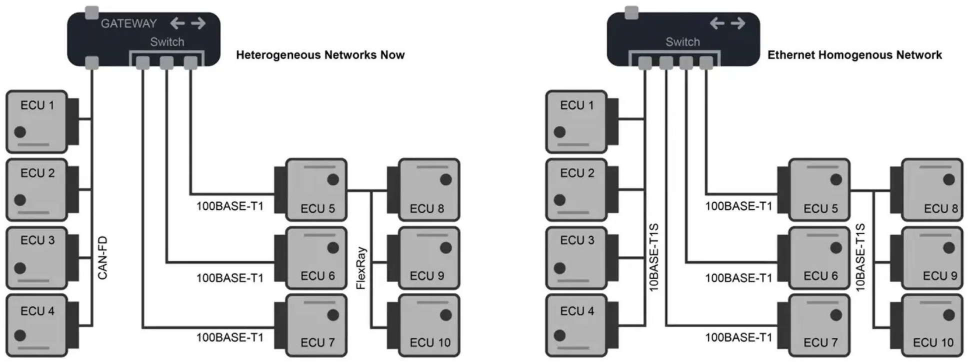


Illustration: [link]

CAN-XL

- **Latest evolution** of the CAN protocol started in 2018 **(eXtra Long)**
- Compatible with **CAN-FD,** allows speed up to **20 Mbit/s** and **2048 bytes** of payload
- Can **tunnel** Ethernet frames

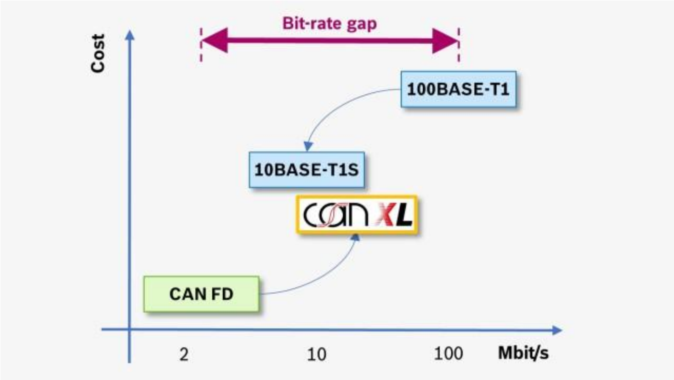


Illustration: [link]

Attack surfaces

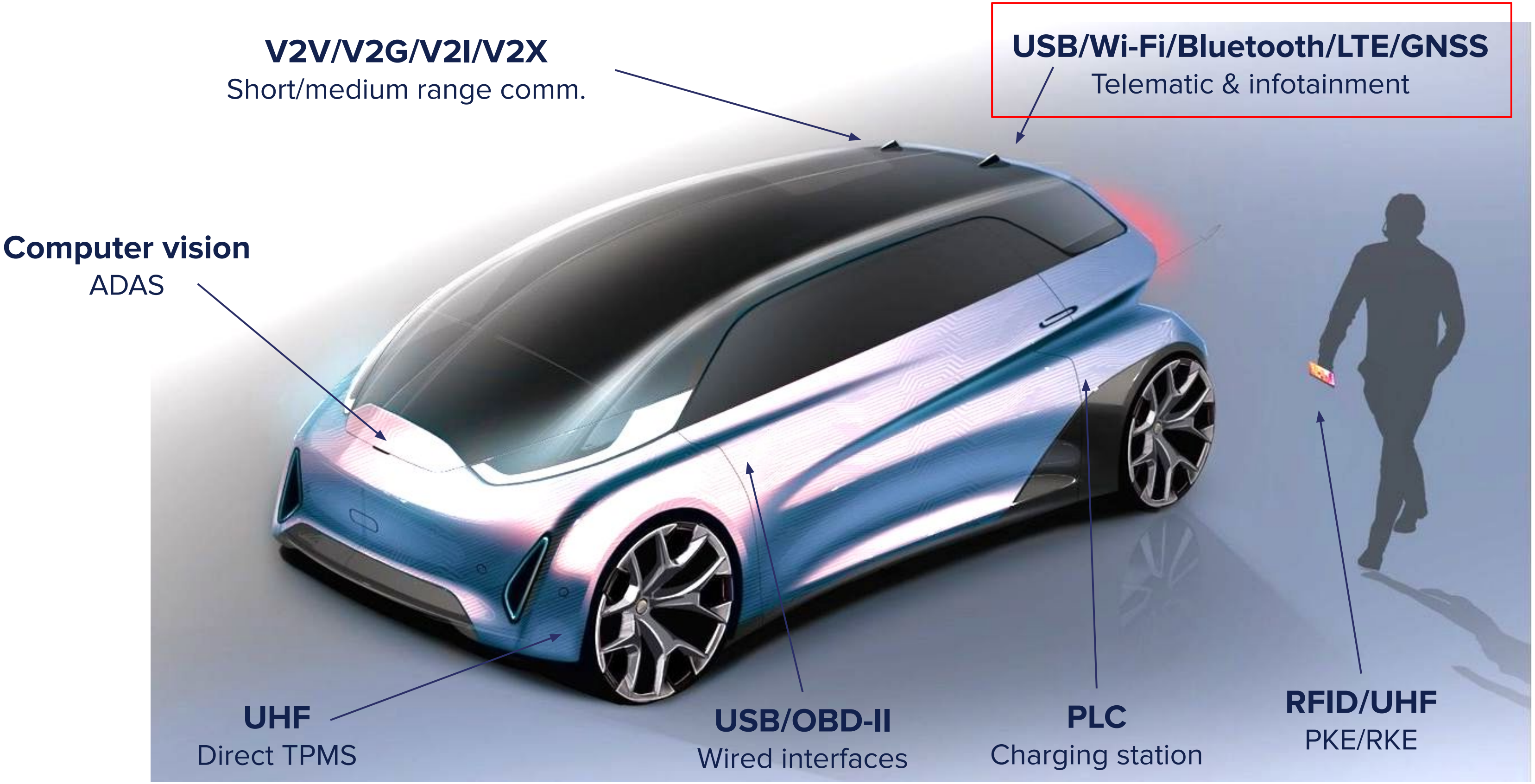


Illustration: [link]

Automotive & cybersecurity: milestone

In **2015** security researchers shown **major vulnerabilities** in a connected vehicle, causing the recall of **1.5M** of vehicles in US

Illustration: [link]

1. https://static.nhtsa.gov/odi/rcl/2015/RCRIT-15V461-7681.pdf

Network segregation

Nowadays, it is common that in-vehicle networks are **segregated** regarding their usage, a **gateway** managing a **secure bridge** between them

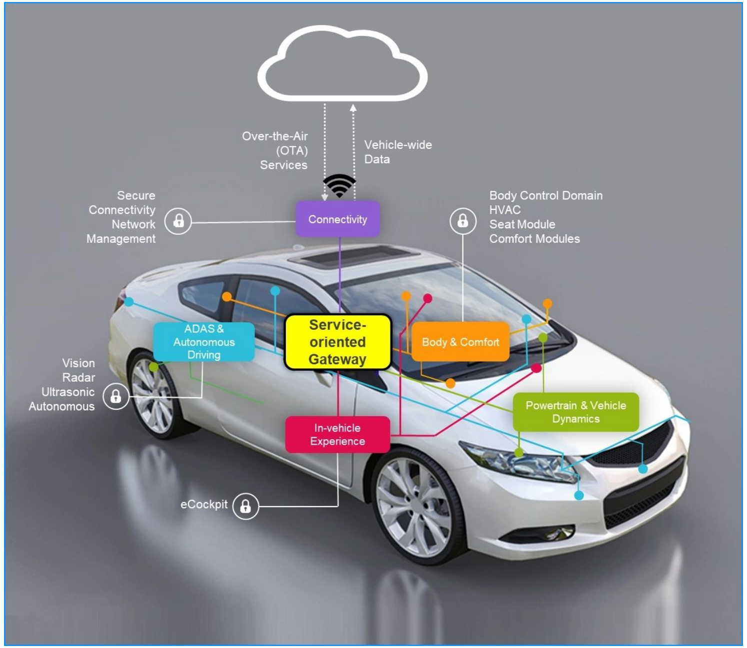


Illustration: [link]

From Domain to Zonal controller

OEMs tend to switch from **Domain architectures** to **Zonal architectures**, where a **central computer** handles data and actuators from different isolated zones, providing better scalability/reliability, **bringing software-defined vehicles**

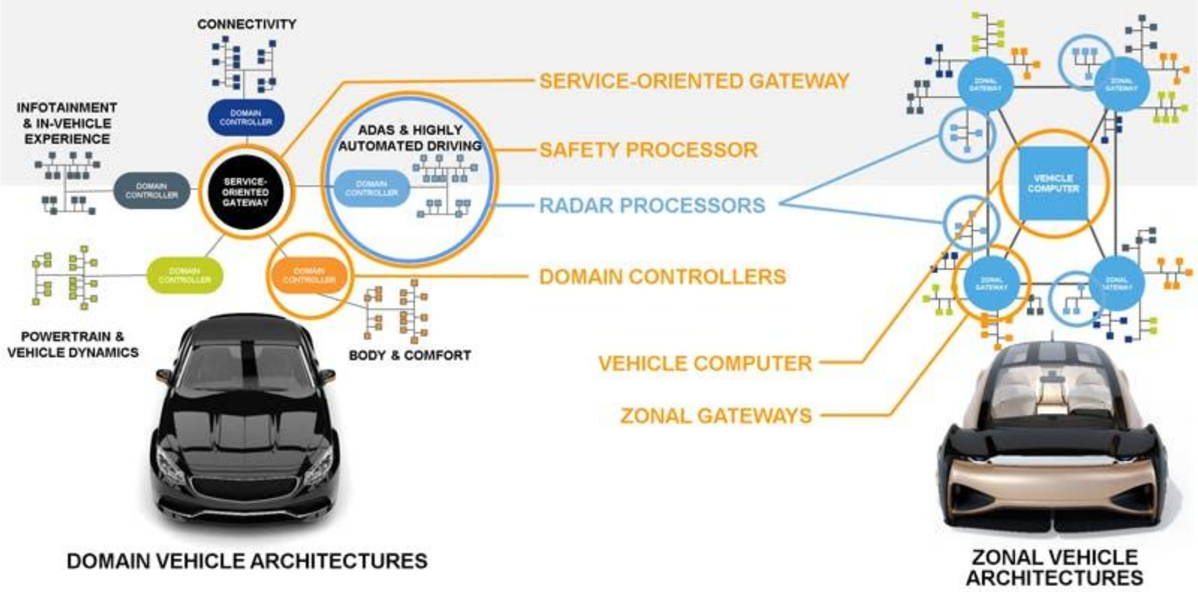
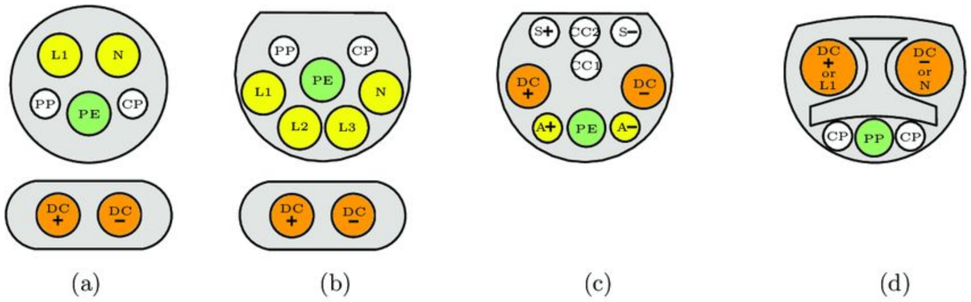


Illustration: [link]

Electrical vehicle plugs

- Electrical vehicle are able to communicate with charging station using **PLC** or **CAN**
- **PLC** communication is made via **Control Pilot** pin

CCS Type 1 & 2 | **GB/T** | **Tesla**



L1	Single-phase AC voltage	S+	Charging Communication CAN
N	Neutral	S–	0V-30V 2A
CP	Control Pilot	CC1	Charging Connection Confirmation
PP	Proximity Pilot	CC2	0V-30V 2A
PE	Protective Earth	A+	Low-voltage auxiliary power supply
L1-L2-L3	Three-phase AC voltages	A–	0V-30V 20A

Illustration: [link]

Vehicle-to-grid

- **Plug & charge**, defined by **ISO-15118**, allows drivers to simply plug their vehicle and start charging without needing to authenticate

- It provides a standardized **secure** communication protocol, relying on several certificates (OEM, MO, CPO…) using several **PKIs**

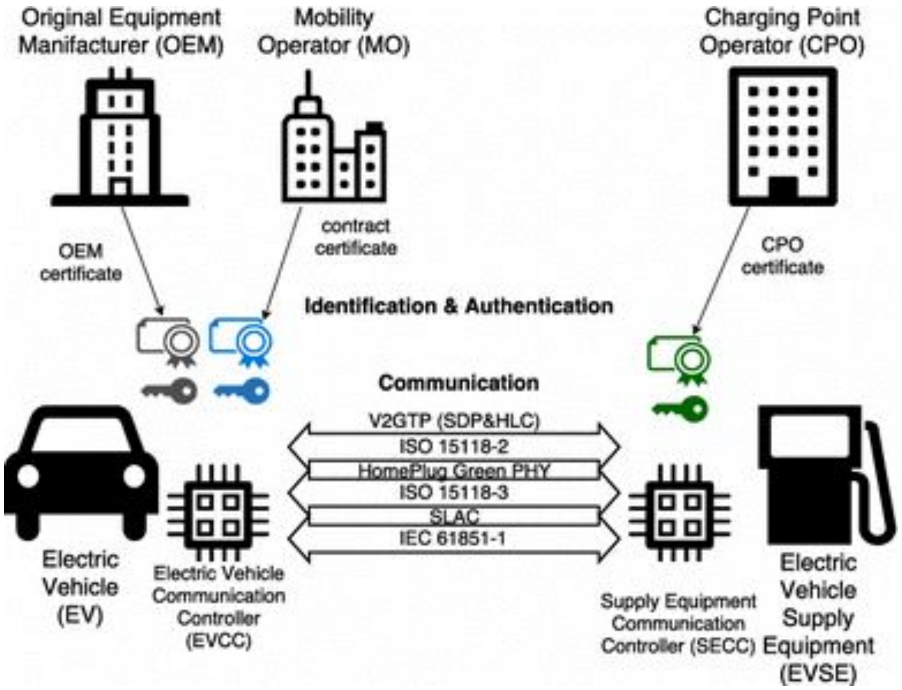


Illustration: [link]

UN/EU regulations

Security

***UNECE WP.29* -** Regulations no. 155 & 156 (annex 1958 agreement since 22 January 2021)

- Manage vehicle cybersecurity risks
- Secure vehicles by design
- Detect /respond to security incidents across a vehicle fleet
- Provide safe, secure software updates

Safety

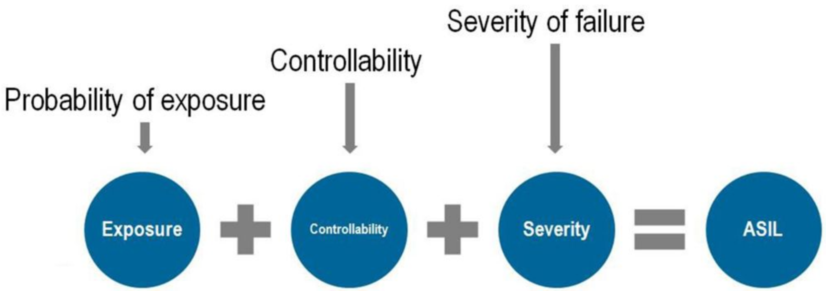
EU 2015/758 - april 2015: mandatory e-Call system in new cars

- Data/voice connection plus GNSS

UNECE R64 - 2009 & ***UNECE R141*** - 2017: mandatory tire pressure monitoring system

> **Safety** brings more attack surface and complexity, requiring more **Security** for automotive <

From ISO 26262 (ASIL)



ASIL: Automotive Safety Integrity Levels

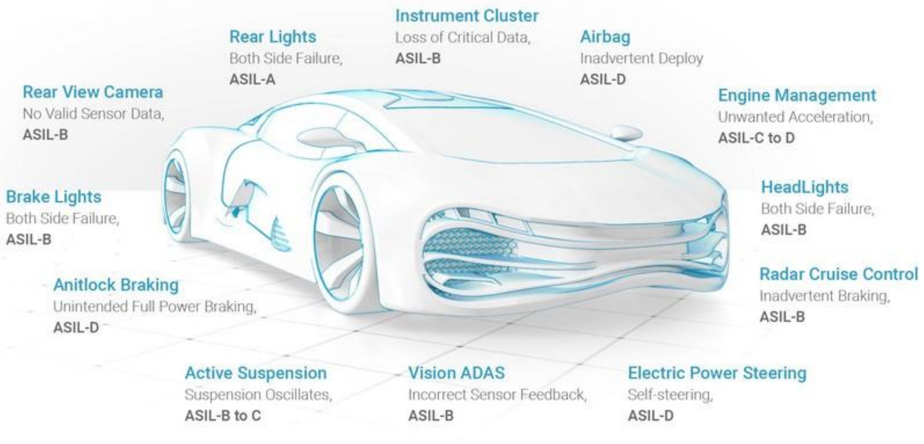


Illustration: [link]

To ISO 21434 (overview)

UN Regulations R155 & R156 refer to ISO 21434

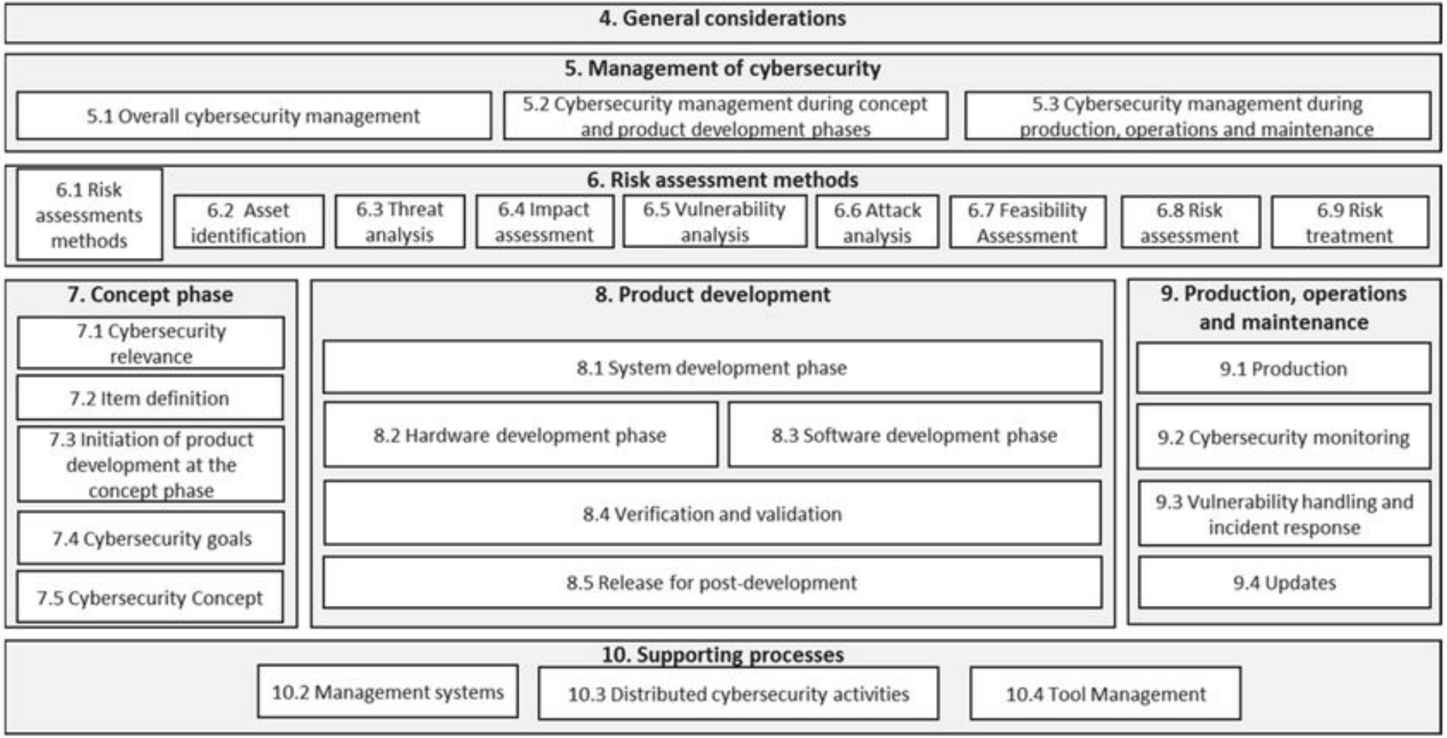


Illustration: [link]

AutoSAR

- Development partnership of **automotive parties** (manufacturer, Tier 1 suppliers) founded in 2003
- Defines a **standardized software architecture** for ECUs, **methodology** and **procedures**
- A **whole car is modelled** using an AutoSAR compliant architecture tool, then specific information for an ECU are extracted

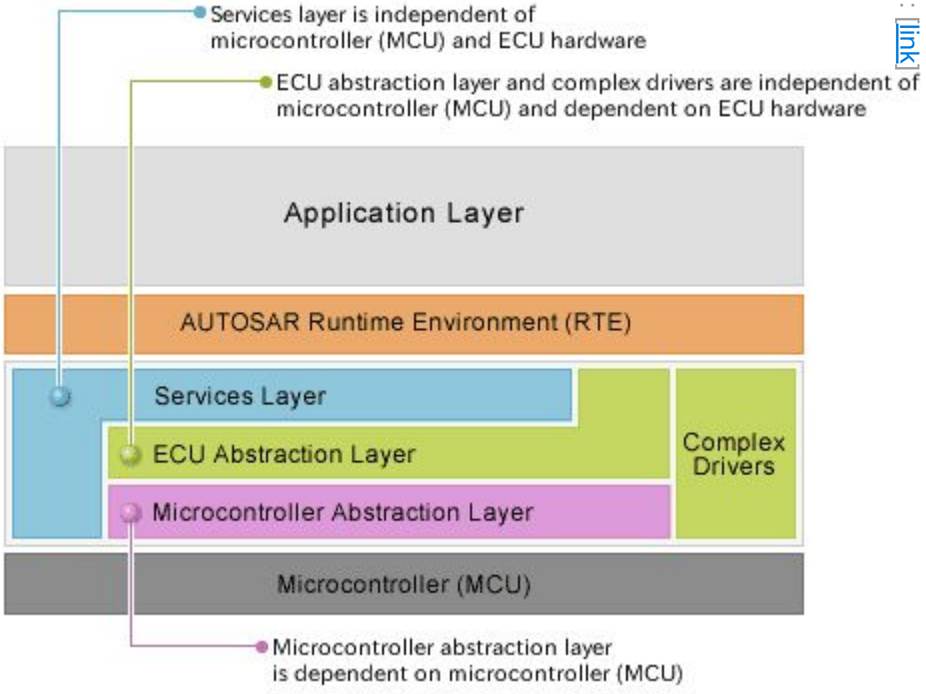


Illustration: [link]

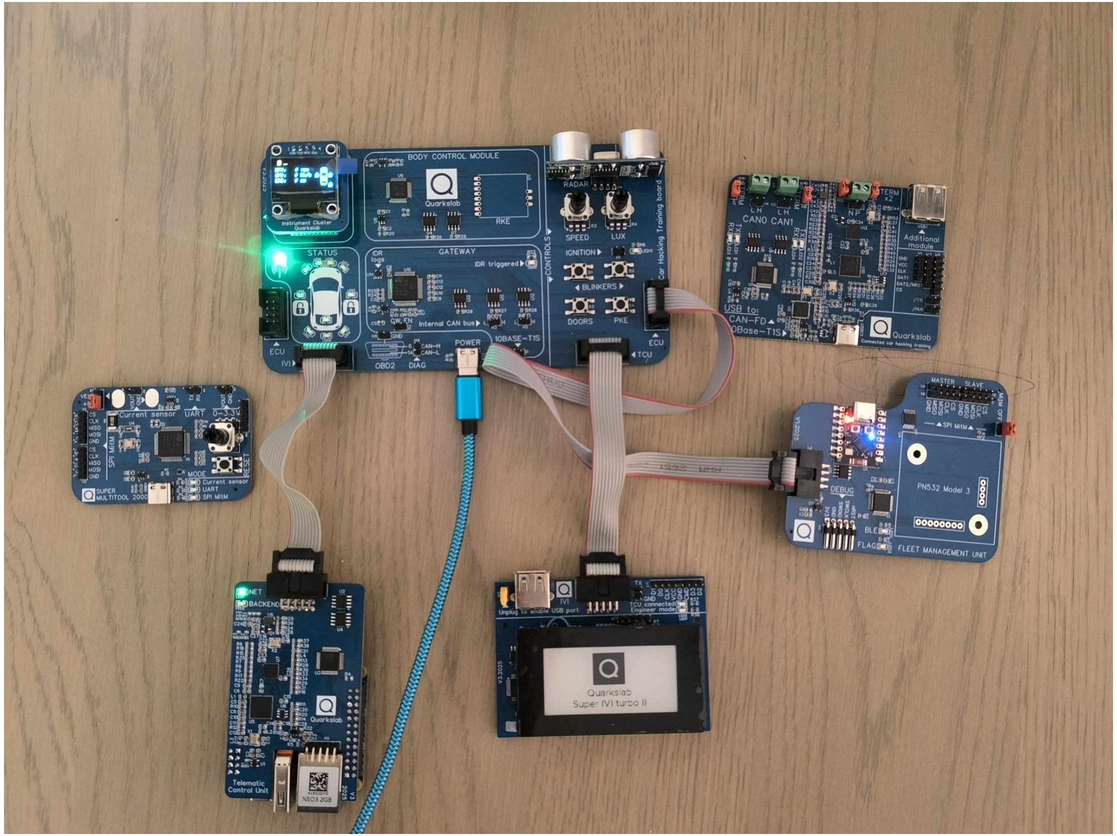
Training equipment: presentation

Quarkslab

Virtual machine

- Ubuntu based VM, including:
  - **Can-utils & Scapy**: CAN tools
  - **srsRAN & Open5GS**: LTE network emulation tools (with BladeRF support)
  - **Sysmo-isim-tools**: programmable SIM management
  - **imHex**: Hex editor
  - **Binary Ninja**: reverse engineering software
  - **Saleae Logic 2**: logic analyser
  - **JADX**: APK reverse engineering
  - **Terminator**: multi-window terminal
  - **Facedancer**: USB emulation
  - **Unicorn/Keystone/Capstone & AFL++**: software emulation and fuzzing
  - **Wireshark, Nmap, Bettercap**: network analysis
  - **WHAD**: Bluetooth Low Energy toolkit

Training ECUs and tools



Playing with real ECUs

- One of our **Car in a Box** will be available during the training

- A **Raspberry Pi** with a **PiCAN** hat is connected to one of the CAN bus, giving access at least to the **ICM**

  - Wi-FI SSID: Quarkslab_CarHacking
  - Wi-Fi passphrase: HackMyC4r!
  - IP: 192.168.11.254
  - Login: student
  - Password: canihack

CAN 101

Quarkslab

CAN bus: key concepts

- **Broadcasted** messages
- The **Arbitration ID** (11 bits: 0x000-0x7FF) and Extended Arbitration ID (29 bits: 0x1FFFFFFF) allows **priority** and **anti-collision** of CAN messages
- Payload of **8 bytes** for CAN, **64 bytes** for CAN-FD and **2048** bytes for CAN-XL

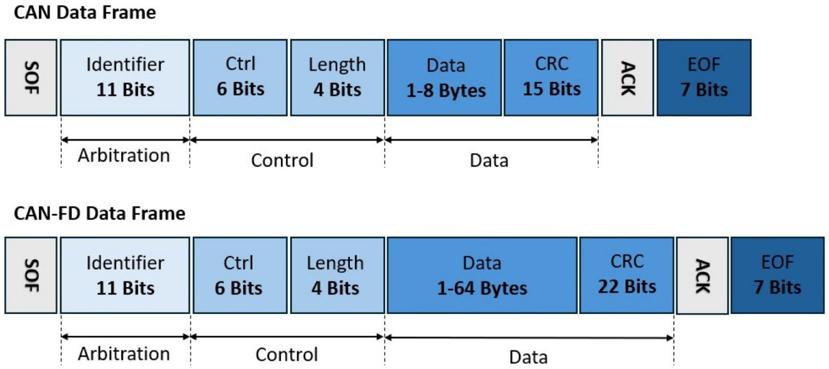


Illustration: [link]

CAN bus wiring

- **Two wires:** twisted pair with an **CAN High** and **CAN Low** wire
- The bus is terminated by a **120 ohms resistor** to prevent signal reflection

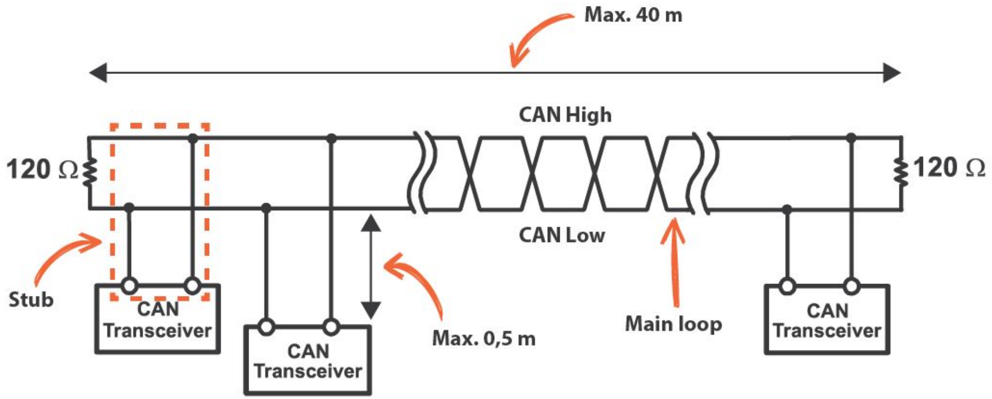
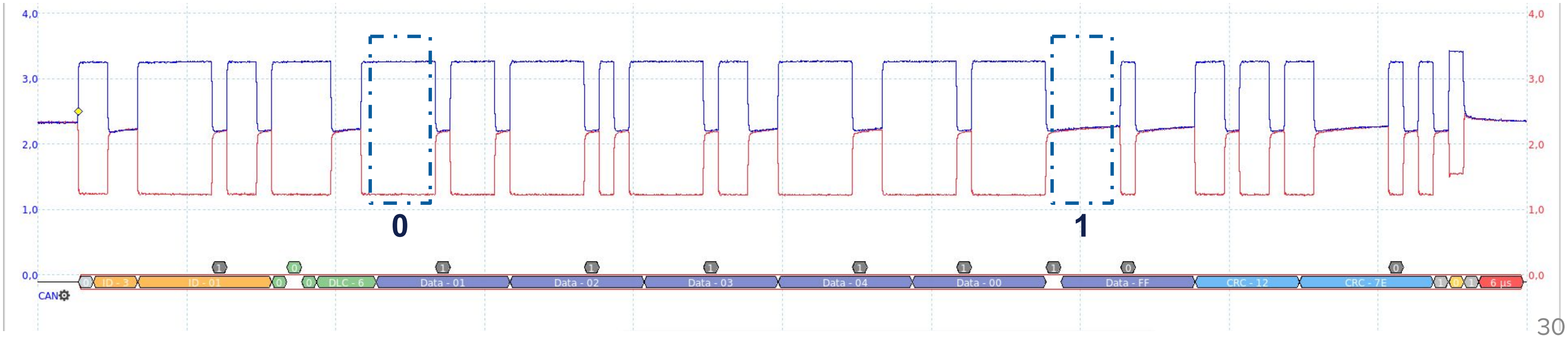


Illustration: [link]

CAN bus signaling

- **Differential signaling**: the **voltage difference** on each wire defines the signal sent (**fault resistant**)
- **Dominant state (0):** CAN High is at ~ 3.5V, CAN Low is at ~ 1.5V
- **Recessive state (1):** CAN High drops to ~ 2.5V, CAN Low level increases to ~ 2.5V



Type of CAN frames

- **Data frame:** sent by a transmitter node to all other nodes
- **Error frame**: sent by any node detecting an error
- **Remote frame**: sent by a node to request the transmission of a data frame with the same identifier
- **Overload frame**: flow control, injects an extra delay after a data or remote frame

Anatomy of a data frame

- **Arbitration ID:** from 0x000 to 0x7FF (11 bits) in standard mode, up to 0x1FFFFFFF (29 bits) in extended mode
- **RTR**: defines if it's a data frame or remote frame
- **IDE**: defines the arbitration ID mode (standard/extended)
- **r0**: recessive (1) for CAN-FD frames
- **Data:** 8 bytes, up to 64 bytes in CAN-FD
- **CRC/ACK**: used for error detection
- **Bit stuffing**: if 5 same bits are consecutive, an opposite bit is added to the frame

CAN bus signalling

- **CAN**: 10, 20, 62.5, 125, 250, 500, 800 and 1 000 Kb/s

  - Most commonly used speed is **500 Kb/s**
  - Non-critical buses use lower speed

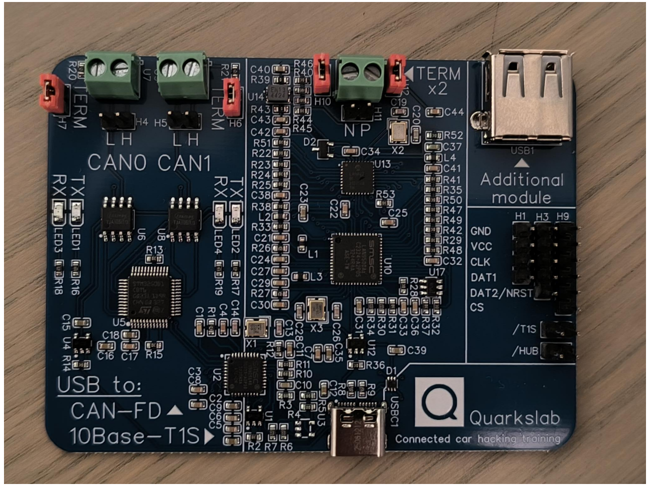
- **CAN-FD:** up to 8 Mb/s

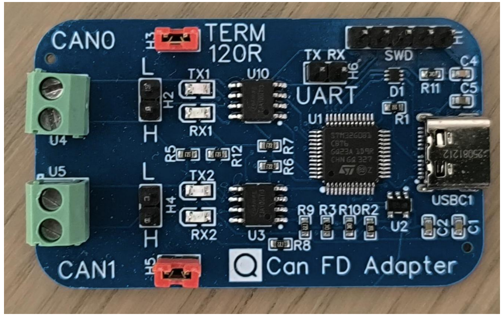
  Arbitration phase limited to 1Mb/s to be **backward** compatible with classic CAN

- **CAN-XL:** up to 20 Mb/s

Provided CAN adapter

- Based on an **STM32G0 chip** + 2x **TJA1051** transceiver
- Supports **CAN-FD**
- Appears as a native CAN interface, supporting **CANSocket**
- Built-in selectable **120 ohms terminations**
- Uses candleLight firmware
- You'll get the smaller CAN adapter to practice on real ECUs after the training





Setting up the CAN adapter

- Find out the interface name

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
[...]
3: vcan0: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/can
4: vcan1: <NOARP,UP,LOWER_UP> mtu 72 qdisc noqueue state UNKNOWN group
default qlen 1000
 5: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
```

Setting up the CAN adapter

- Using ip command, set the device at the desired speed (here 500Kbps)

```
$ sudo ip link set can0 up type can bitrate 500000 dbitrate 500000 fd on
# To support classic CAN (no-fd)
$ sudo ip link set can0 up type can bitrate 500000
```

- If you need to change the speed of the interface, you'll need to bring it down first

```
$ sudo ip link set can0 down
```

- You can also rename the interface, to have the same label than the CAN bus you're working on

```
$ sudo ip link set can0 name CAN-HS
```

Setting up the CAN adapter

- You may also need to set a bigger buffer when sending a large amount of data

```
$ sudo ip link set can0 txqueuelen 1000
```

- If everything is correctly set, you must get the following output

```
$ sudo ip a | grep can
3: can0: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UP group default qlen 1000
    link/can
```

Reading a CAN bus: candump

- The can-utils library has many tools to work with CAN bus
- **Candump** display every message going through the bus

```
$ candump -x -e -a can0

can0  TX - -  2e0  [3]  02 06 64                    '..d'
can0  TX - -  130  [3]  02 06 64                    '..d'
can0  RX - -  121  [2]  01 46                       '.F'
can0  RX - -  124  [2]  01 46                       '.F'
can0  TX - -  128  [8]  07 33 68 65 65 6C 70 6F     '.3heelpo'
```

Options :
-x: display RX/TX
-e: display error frames
-a : ASCII output

Message data

Message length

Arbitration ID

Reading a CAN bus

- Filters can be applied to only display specific arbitration ID or a range of ID using masks
- For each filter, add "," followed by the desired arbitration ID then ":" and the desired mask
  A binary comparison is made with the mask, only ID matching mask "1" bit are displayed

```
$ candump -x -e -a can0,123:7FF,030:7F0

can0  TX - -  123   [3]  02 06 64                    '..d'
can0  RX - -  031   [5]  00 01 02 A6 64              '....d'
can0  RX - -  036   [2]  01 46                       '.F'
can0  RX - -  123   [2]  01 46                       '.F'
can0  TX - -  03F   [8]  07 33 68 65 65 6C 70 6F     '.3heelpo'
```

Write on a CAN bus: cansend

- **Cansend** is the most basic tool to send data over the CAN bus

```
$ cansend can0 321#c0ff33
```

- **Mandatory arguments :**
  can0: the can bus interface
  321#c0ff33: **arbitration ID** in hex followed by a hashtag and **1 to 8 bytes** of data

- An **empty frame** can also be sent

```
$ cansend can0 321#
```

- To send a **CAN-FD frame**, use 2 “**#**” followed by a flag bit (0 by default), then the message

```
$ cansend vcan0 321##0c0ff33
```

Lab 1 - setting and interacting with a CAN Bus

Goals

- On your CAN-FD adapter board, connect **CAN0** and **CAN1**
- Using “**ip**” command, configure and activate the two CAN bus with a bitrate of **500 000 kbps** with **FD active**
- Complete challenges **Ignition - canutils**

Scapy: CAN

- **Scapy** is a packet manipulation program written in **Python**
- Very useful to **capture**, **craft** packets on different kind of networks
- Can load or save **pcap** to interact with **Wireshark**
- More info on https://scapy.net/
- Has multiple **automotive** libraries 😎
  Complete documentation is available here:
  https://scapy.readthedocs.io/en/latest/layers/automotive.html

Scapy: CAN

- To be able to use **Scapy** with CAN packets, it is mandatory to load at least the layer CAN

```
$ scapy
>>> load_layer("can")
```

- From a Python script, you can import **Scapy** using

```
from scapy.all import *
```

- **"Contrib"** are additional modules that extend the capability of **Scapy**. Multiple contribs are available for the automotive, like cansocket which allows to communicate with socketcan sockets, like the one of our CAN adapter

```
$ scapy
>>> load_layer("can")
>>> load_contrib("cansocket")
```

Scapy: CAN - writing on a CAN bus

- Try the following command to send a message on the bus

```
$ scapy
>>> load_layer("can")
>>> load_contrib("cansocket")
>>> s = CANSocket(channel="can0") # add fd=True for CAN-FD support
>>> s.send(CAN(identifier=0x123, data=b'\x01\x02\x03'))
```

- We load the **can layer** and the **cansocket contrib**, which are mandatory
- We create a “socket” on our CAN interface
- Using the **CAN method**, we create a CAN packet and send it through our socket
- The option **flags=’extended’** could be added to our packet to have an **extended ID**

Scapy: CAN - reading on a CAN bus

- **Scapy** has three methods, **recv**, **sr** and **sr1** which means **Receive** and **Send and Receive**. **sr** and **sr1** first send a packet, then capture the result(s)
- However, with all the traffic on the CAN bus, those methods are useless
- The **sniff** method fills our needs, try the following commands in your **Scapy** terminal

```
>>> pkts = s.sniff(count=5)
<Sniffed: TCP:0 UDP:0 ICMP:0 Other:5>
>>> for pkt in pkts:
. . .:        pkt.show()
```

Scapy: CAN - using options and callback with sniff

- **Option** count sets the maximum number of CAN frame to capture
- The **timeout** option (floating number) sets the duration, in seconds, before the function ends
- Using **prn option** sets a callback to a method or a lambda on the captured frame
- Try the following command:

```
>>> s.sniff(timeout=10.0, count=50, prn=lambda x: x.show())
```

Scapy: CAN - filtering the socket

- As **Scapy** standard filters are based on **Berkeley Packet Filter (BPF)**, they do not work with the **CAN** layer
- However, the **cansocket** contrib handles filter like **candump** (identifier + bit mask)
- Filters have to be set during the **socket initialization**
- Let’s update our socket

```
>>> s.close()
>>> s = CANSocket(channel="can0", can_filters=[{"can_id":0x123, "can_mask":0x7FF}])
>>> s.sniff(timeout=10.0, count=50, prn=lambda x: x.show())
```

Scapy: CAN - loading/saving captures

- **Scapy** supports both **Wireshark** and **candump** logs
- Using **rdpcap** or **wrpcap**, it is possible to read/write a pcap file

```
>>> pkts = s.sniff(count=50)
>>> wrpcap("./test.pcap", pkts)
>>> pcap = rdpcap("./test.pcap")
>>> pkts
>>> pcap
```

- Candump logs can only be read using **rdcandump** method

```
>>> pkts = rdcandump("path_to_your_candump.log")
>>> pkts
<candump.log: TCP:0 UDP:0 ICMP:0 Other:52571>
```

Lab 2 - using Scapy with a CAN Bus

Goals

- can-utils is limited to process complex frames, perform computation on CAN messages or work with diagnostic protocols
- Various Python modules support CAN messages, **Scapy** is the one we daily use, as it implements higher-level protocols, like **UDS**
- Complete challenges **CAN 101 - Scapy**

ISO-TP & UDS

Quarkslab

ISO-TP Transport Protocol

- **ISO-TP** protocol allows sending data over the **8 bytes** limit of the standard CAN Bus
- It can carry up to **4095 bytes** of payload
- **ISO-TP** segments messages into **multiple frames**
- The **high nibble** of the first byte of every frame defines its type
- **4 values** are possible:
    - 0: **Single Frame**
    - 1: **First Frame**
    - 2: **Consecutive Frame**
    - 3: **Flow Control Frame**

ISO-TP - Single Frame

- To send up to **7 bytes** using **ISO-TP** protocol, we will use a **Single Frame**
- The **low nibble** of the **first byte** define the **length** of the data transmitted
- Example:

```
$ candump -a can0,7e0:7FF
can0  7e0   [8]  02 10 01 00 00 00 00 00              '........'
```

- Padding: optional, depends on the upper protocol (0x00 or 0xAA)
- Data (2 bytes): 0x1001
- Data length: 2 bytes (0x_2)
- Frame type: Single Frame (0x0_)

- Standard **CAN** frame looks like **ISO-TP Single Frame**, you can differentiate them if padding is used

ISO-TP - Multiple frames: First Frame

- **ISO-TP** can send up to **4 095 bytes.** If a message has more than 7 bytes, the **high nibble** of the **first frame** will be **0x1**, which means "**First Frame**"

- The **low nibble** of the first byte and the **second byte** are the **length** of the transmitted message, from **0x000 to 0xFFF**

- The **replying ECU** will wait for a **Flow Control Frame** to send the rest of the message

```
$ candump -a can0,7e0:700
can0  7e0   [8]  02 09 02 00 00 00 00 00        '........'
can0  7e8   [8]  10 14 49 02 41 42 43 44        '..1.abcd'
```

Data (OBD-II positive response + first 4 bytes)

Data length: 20 bytes (0x014)

Frame type: First Frame (0x1_)

ISO-TP - Multiple frames: Flow Control Frame

- To get the **remaining frames** of the message, the querying device has to send a **Flow Control Frame** after receiving the **First Frame**
- The **second byte** tells the ECU how many frames will be sent without waiting for a new **Flow Control Frame**. Set it to **0x00** for cancelling further control
- The **third byte** set the delay in milliseconds between two **Consecutive Frames**

```
$ candump -a can0,7e0:700
can0  7e0   [8]  02 09 02 00 00 00 00 00         '........'
can0  7e8   [8]  10 14 49 02 41 42 43 44         '..1.abcd'
can0  7e0   [8]  30 00 0A 00 00 00 00 00         '........'
```

Interval between two Consecutive Frame (10ms)

Flow control: no further control (0x00)

Frame type: Flow Control Frame (0x3_) with Clear to Send status (0x_0)

ISO-TP - Multiple frames: Consecutive Frames

- Once the **Flow Control Frame** is received, the ECU will send the rest of the message using **Consecutive Frames**

- The **low nibble** of the first byte will increment and roll from 0x1 to 0xF for each frame of the message

```
$ candump -a can0,7e0:700
can0  7e0   [8]  02 09 02 00 00 00 00 00        '........'
can0  7e8   [8]  10 14 49 02 41 42 43 44        '..1.abcd'
can0  7e0   [8]  30 00 0A 00 00 00 00 00        '........'
can0  7e8   [8]  21 45 46 47 48 48 50 51        '.efghijk'
can0  7e8   [8]  22 52 53 54 55 56 57 58        '.lmnopqr'
```

Data

Frame type: Consecutive Frame (0x2_)

Lab 3 - ISOTP

Goals

- Complete challenges **CAN 101 - ISOTP**

UDS - Universal Diagnostic Services

- **UDS** is a mandatory protocol for diagnosis, tuning and update operations on ECUs
- It uses **Service** and **Sub-Function**
- Queries are made by the **Tester** (client) to a **Server** (ECU)
- Each **Server** has its own **Request arbitration ID** and **Reply arbitration ID**
- **Reply arbitration ID** = **Request arbitration ID + 0x08** (normally…)
- For each query, the **Server** replies with a **positive response** (Service code + 0x40) or **negative response** (0x7F)
- Usual **arbitration ID** range is **0x700** to **0x7FF & 0x18DA0000-0x18DAFFFF, 0x7DF** being reserved as a broadcast request

UDS - Universal Diagnostic Services

Some useful services

- **0x10:** Diagnostic Session Control
- **0x11**: ECU Reset
- **0x27**: Security Access
- **0x29**: Authentication
- **0x3E**: Tester Present
- **0x22**: Read Data By Identifier
- **0x23**: Read Memory By Address
- **0x2E**: Write Data By Identifier
- **0x2F**: Input/Output Control by Identifier
- **0x3D**: Write Memory By Address
- **0x31**: Routine Control
- **0x34**: Request Download
- **0x35**: Request Upload

Negative Response Code (NRC)

- **0x10**: General Reject
- **0x11**: Service Not Supported
- **0x12**: Sub-function Not Supported
- **0x13**: Incorrect Message Length or Invalid Format
- **0x22**: Conditions Not Correct
- **0x24**: Request Sequence Error
- **0x31**: Request Out Of Range
- **0x33**: Security Access Denied
- **0x35**: Invalid Key
- **0x36**: Exceeded Number of Attempts
- **0x7E**: Sub-Function not Supported in Active Session
- **0x7F**: Service Not Supported in Active Session

And much more: https://en.wikipedia.org/wiki/Unified_Diagnostic_Services
https://automotive.softing.com/fileadmin/sof-files/pdf/de/ae/poster/UDS_Faltposter_softing2016.pdf

UDS on CAN

- **UDS** request/response are send using the **ISOTP** protocol
- The first byte of the payload is the **Service**
- Other bytes depend on the requested **Service**
- Most of the UDS implementation requires **padding**

```
$ candump  can0,7e0:7FF
can0  7e0   [8]  02 10 01 AA AA AA AA AA
# Diagnostic session control with SubFunction 01 (defaultSession)
can0  7e0   [8]  03 21 F1 90 AA AA AA AA
# Read data by identifier: DID 0xF190
```

Scapy: UDS

- Multiple **automotive contribs** exist in Scapy, one of them handle the **UDS protocol**

```
>>> load_contrib("isotp") # Loading ISOTP contrib is required to create ISOTP sockets
>>> load_contrib("automotive.uds")
```

- You can craft a **UDS message** calling the related **UDS Service constructor**
 Reminder: if you're running Scapy from the terminal, the **autocompletion** using "tab" works

```
>>> UDS_                    # Press tab to see all the supported services
>>> ls(UDS_DSC)             # ls command lists all the arguments
diagnosticSessionType: ByteEnimField                    = ('0')
>>> session = UDS_DSC(diagnosticSessionType = 2)
>>> isotpsocket.send(UDS()/session)
>>> # An UDS Service has to be pack into an UDS frame to be sent: UDS/UDS_xx()
```

Scapy: UDS - some supported services

0x10: Diagnostic Session Control	**UDS_DSC**
0x11: ECU Reset	**UDS_ER**
0x27: Security Access	**UDS_SA**
0x3E: Tester Present	**UDS_TP**
0x22: Read Data By Identifier	**UDS_RDBI**
0x23: Read Memory By Address	**UDS_RMBA**
0x2E: Write Data By Identifier	**UDS_WBDI**
0x2F: Input/Output Control by Identifier	**UDS_IOCBI**
0x3D: Write Memory By Address	**UDS_WMBA**
0x31: Routine Control	**UDS_RC**
0x34: Request Download	**UDS_RD**
0x35: Request Upload	**UDS_RU**

Scapy: UDS - automatic NRC description

- When creating the ISO-TP socket with Scapy, adding **basecls=UDS** option give a full support of the **UDS protocol**, even the **NRC** automatic translation

```
>>> isotpsocket = ISOTPSocket("can0", tx_id=0x7e0, rx_id=0x7e8, padding= True,
basecls=UDS)
```

- Now try an **unsupported request** on the **ECU**

```
>>> isotpsocket.sr1(UDS()/UDS_SA(securityAccessType = 0xFF), timeout=1.0)
Begin emission:
Finished sending 1 packets.
Received 1 packets, got 1 answers, remaining 0 packets
<UDS  service=NegativeResponse |<UDS_NR  requestServiceId=SecurityAccess
negativeResponseCode=subFunctionNotSupported>
```

Scapy: UDS

- Using the **UDS_NR** as a constant, you can check if the captured packet is an error, without looking at the packet data

```
>>> pkt = isotpsocket.sr1(UDS/UDS_SA(securityAccessType = 0xFF), timeout=1.0)
Begin emission:
Finished sending 1 packets.
Received 1 packets, got 1 answers, remaining 0 packets
>>> pkt == UDS_NR # UDS_NR in pkt also works
True
>>> pkt.show()
```

Lab 4 - Scapy and UDS

Goals

- This training does not aimed to make you UDS experts, but we will use few basic Services in the various
- Complete challenges **CAN 101 - UDS**

Automotive Ethernet

Automotive Ethernet: need for speed

- As car are becoming more and more **complex** (assisted/autonomous driving), there is a growing need for:
  - **Low-latency**
  - **Robust links** over simple wires
  - **Flexible technologies** that cover multiple use-cases

Illustration: [link]

Automotive Ethernet: two standards

- First implementation: **100/1000Base-T1**
- Defined by **IEEE 802.3bw 2015**
- "Classical" **point-to-point** network

- Evolution: **10Base-T1S**
- Defined by **IEEE 802.3cg 2020**
- **Multidrop** network

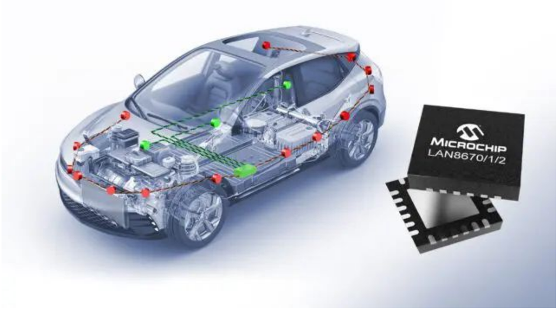
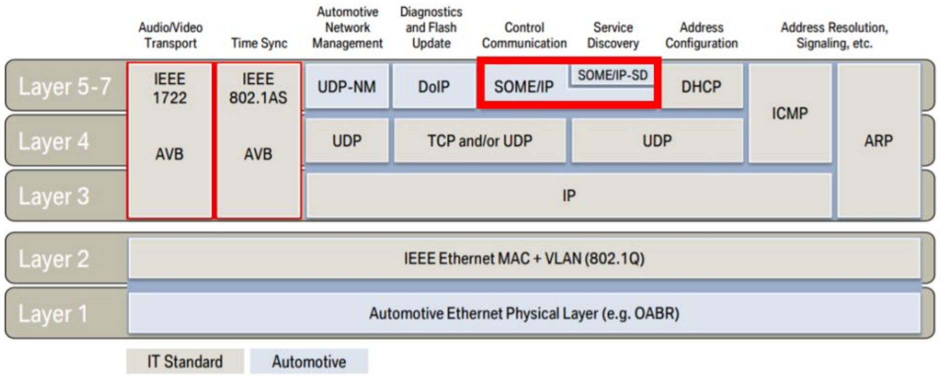


Illustration: [link]

Difference between Ethernet & 100/1000Base-T1

- Only the **physical** layer differs:
  - Uses single differential **unshielded copper twisted pair**
  - Uses **PAM-3** signalling
  - Maximum length is **15 m**
  - **Connectors** are not defined (no RJ45 !)
  - A node is set as **Master**, the other as **Slave**, to handle **echo cancellation**



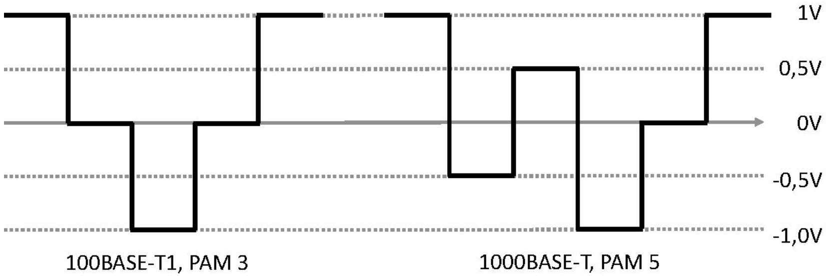


Illustration: [link] & [link]

Network topology

- **ECUs** are linked **port to port** or through **switches**
- An **ECU** can be a **switch** (gateway)
- Several **VLANs** are used for **security** or to define different levels of **quality of services**

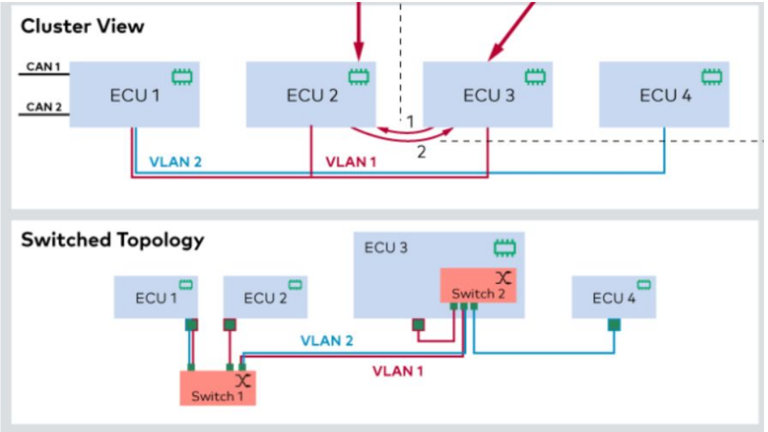


Illustration: [link]

10Base-T1S Automotive Ethernet

- Allows **2 to 8 nodes** to communicate over a **single twisted pair,** up to 25m
- Aims to replace classical automotive networks, like CAN, having an all-Ethernet network

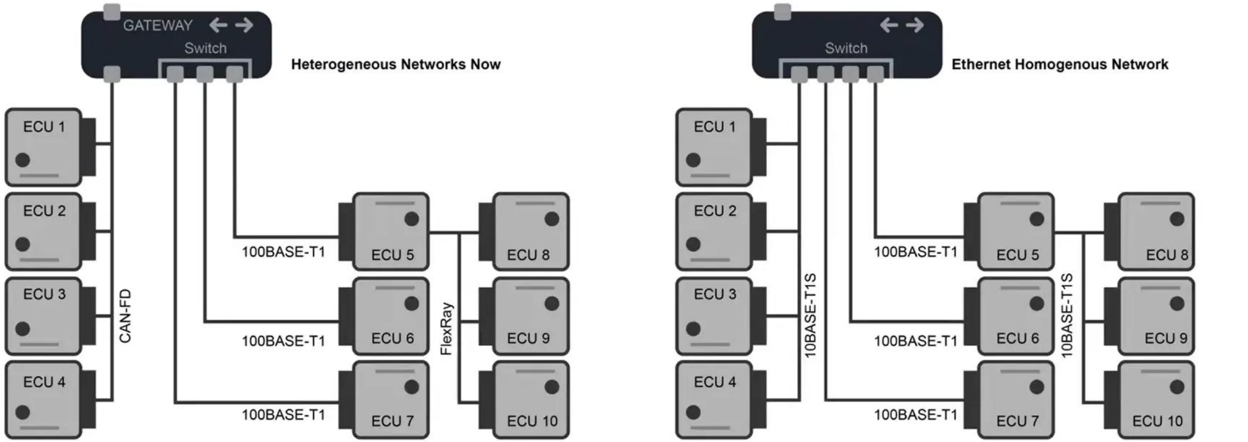
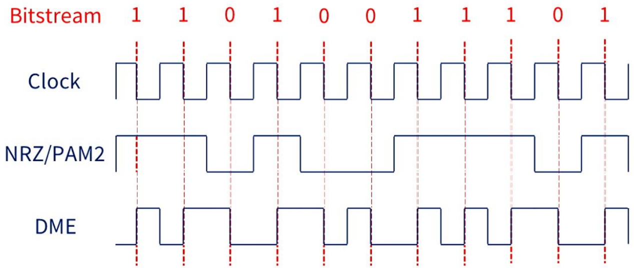


Illustration: [link]

10Base-T1S Automotive Ethernet

- Also uses single differential **unshielded copper twisted pair**
- But relies on **Differential Manchester Encoding (DEM)** signalling
- Bus is terminated by **100 ohms resistors**
- Each **node** has an **ID**, 0 being for the **Master**, for the **Physical Layer Collision Avoidance** (PLCA)
- The **Master** send periodic **beacon** . **Slave** nodes are given a **transmit opportunity** in order of their ID
- A **silence** (~20 bits) is when a **node** has **no data to transmit**. It could also send a **commit** to buy additional time to transmit data



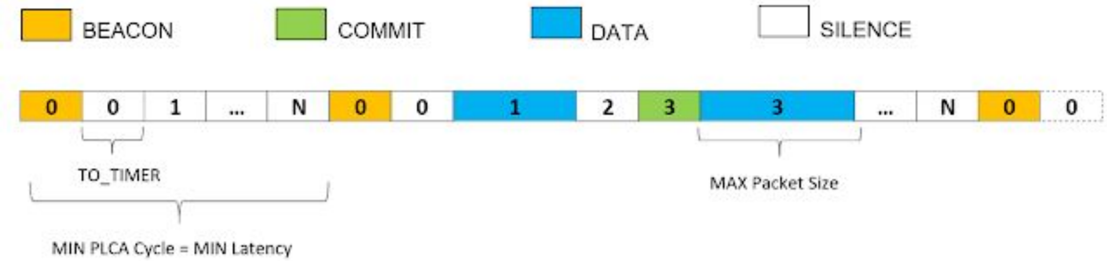


Illustration: [link] & [link]

Connecting to an Automotive Ethernet network

- To connect to an automotive Ethernet network, a **Media Independent Interface (MII)** is required
- It **bridges** classical and automotive ethernet **physical layers** so you can plug an RJ-45

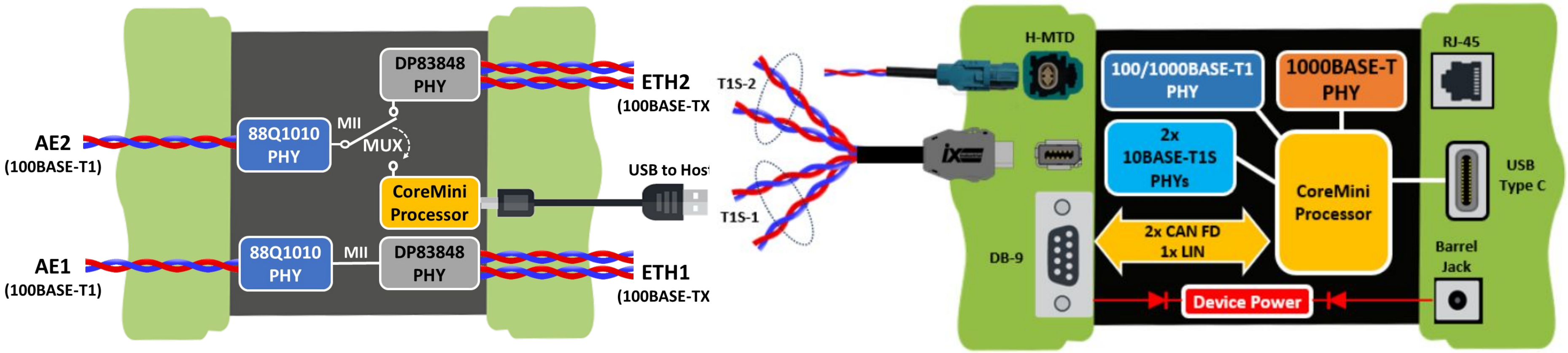


Illustration: [link] & [link]

Automotive Ethernet
DoIP

Quarkslab

Diagnostics Over IP

- **DoIP** (Diagnostics Over IP) allows **remote** and **quicker** diagnostic of a car **(ISO 13400)**
- It's a **transport protocol** for diagnostic services like **UDS** over IP
- It also **manages** specific **services** like:
  - Vehicle Identification
  - Routing Activation
  - Node information
  - Aliveness Mechanism
- It uses both **TCP** and **UDP**
- Must use port **13400**

Diagnostics Over IP

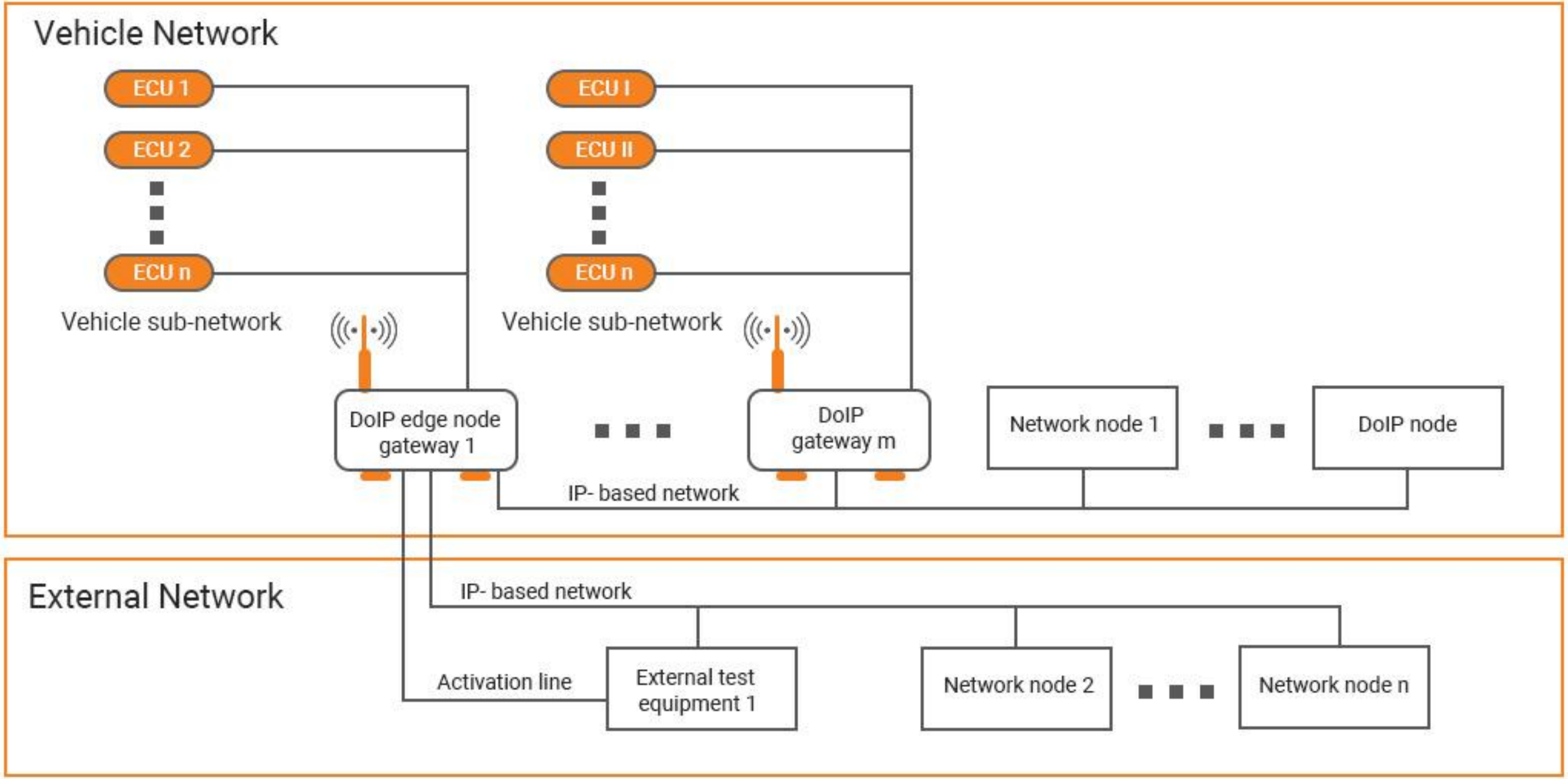
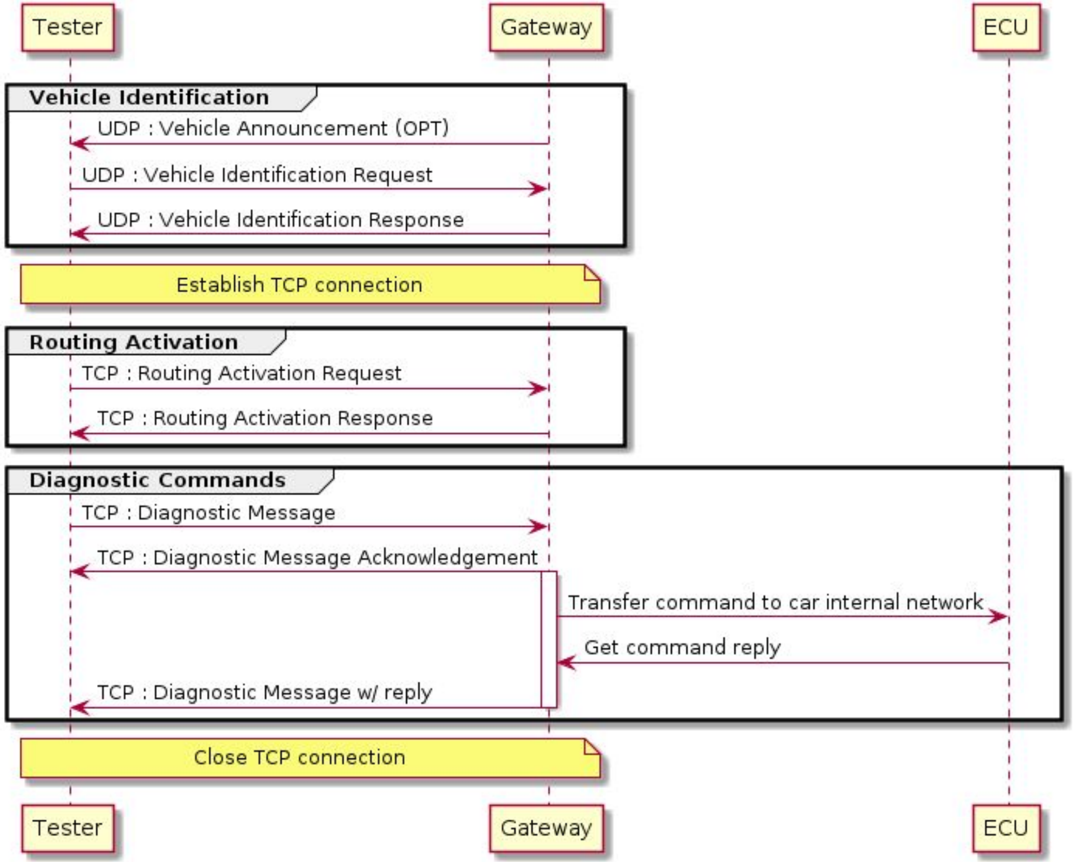


Illustration: [link]

Diagnostics Over IP - Flowchart



Diagnostics Over IP - Message

8 bits	8 bits	8 bits	8 bits
Protocol Version *(0x02)*	Inverse Protocol Version *(0xFD)*	Payload Type *(0x8001)*	
Payload Length			
Payload			

Diagnostics Over IP: payload structure

- An **ECU** is identified by its 2 bytes **Logical Address**
- **GW/Node Logical Addresses** could be obtained using **Vehicle Identification Requests**
- **Manufacturer Specific Addresses** are in range **0x0001 - 0x0DFF & 0x1000 - 0x7FFF**

Item	Position (Byte)	Length (Byte)
Source Address	0	2
Target Address	2	2
Data	4	...

Diagnostics Over IP: payload types

- **0x0001** : Vehicle Identification Request Message
- **0x0002** : Vehicle Identification Request Message with EID [1]
- **0x0003** : Vehicle Identification Request Message with VIN
- **0x0004** : Vehicle Announcement Message/Vehicle Identification Response
- **0x0005** : Routing Activation Request
- **0x0006** : Routing Activation Response
- **0x0007** : Alive Check Request
- **0x0008** : Alive Check Response
- **0x4001** : Diagnostic Entity Status Request
- **0x4002** : Diagnostic Entity Status Response
- **0x8001** : Diagnostic Message
- **0x8002 :** Diagnostic Message Positive Acknowledgement
- **0x8003 :** Diagnostic Message Negative Acknowledgement

1. Entity Identifier, most of the time the MAC address

Diagnostics Over IP: sending a DoIP request w/ Scapy

- Using **“automotive.doip”** contrib we can craft/decode packets
- Reminder: to use raw **network interfaces**, scapy has to be run as “root”

```
>>> load_contrib("automotive.doip")
>>> s = L3RawSocket(iface="enp0s3")
>>> doip = DoIP(payload_type=0x0003, vin=b'VIN1234567890ABCD')
>>> resp  = s.sr1(IP(dst="192.168.11.123")/UDP(dport=13400)/doip, timeout=2)
```

1. https://github.com/secdev/scapy/blob/master/scapy/contrib/automotive/doip.py

Diagnostics Over IP: routing activation

- Before sending **Diagnostic Message**, a route must be set over **TCP**
- Using payload type **0x4001**, the tester must send a valid **Logical Address** and an **Activation Type**
- If the route is correctly set, the **DoIP** gateway/node will return its **Logical Address**
- When creating a **DoIP** TCP socket using `` `DoIPSocket` ``, Scapy will by default set a **Source Address 0xE80** and an **Activation Type 0x00**

```
>>> load_contrib("automotive.doip")
>>> socket = DoIPSocket("192.168.11.123", source_address=0xE80,
activation_type=0x00)
>>> socket = DoIPSocket("192.168.11.123") # Does the same
```

Diagnostics Over IP: sending a DoIP message

```
>>> load_contrib("automotive.uds")
>>> load_contrib("automotive.doip")
>>> uds = UDS()/UDS_DSC(diagnosticSessionType= 0x01)
>>> doip = DoIP(payload_type=0x8001, source_address=0xe80, target_address=0x17ea)
>>> socket = DoIPSocket("192.168.11.123")
>>> resp  = socket.sr1(doip/uds, timeout=2)
```

1. https://github.com/secdev/scapy/blob/master/scapy/contrib/automotive/doip.py

Lab 5 - DoIP

Goals

- Complete challenges **Ignition - Automotive Ethernet**

Automotive security: good practices

Quarkslab

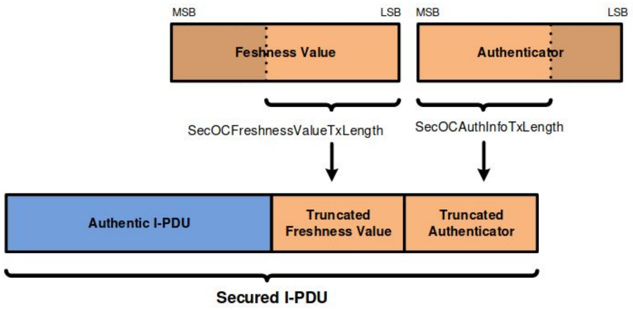
Automotive network security: good practices

- CAN networks have known vulnerabilities, including:
  - Non encrypted data and non authenticated sender
  - Replayable messages

- **AutoSAR** implements **SecOC** to authenticate CAN messages

- Using **TLS encryption** is also recommended in Automotive Ethernet networks to prevent **man-in-the-middle** attacks

- High-end designs already use **MACSEC**



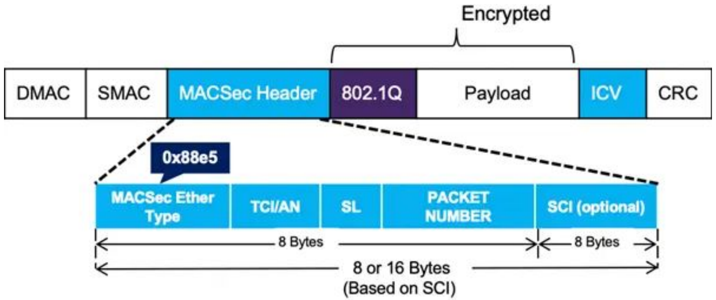


Illustration: [link] & [link]

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: www.quarkslab.com

@quarkslab

Quarkslab