Introduction to Car Hacking

Barbhack 2024

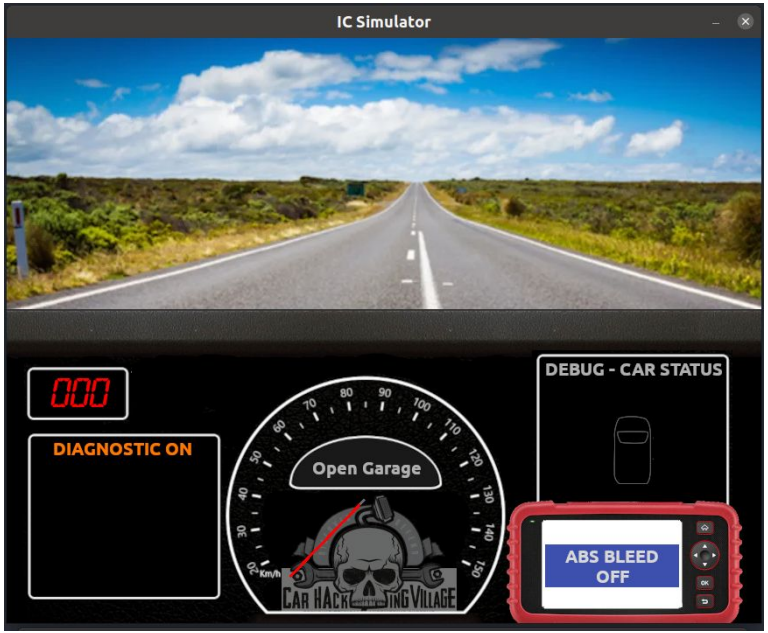
Philippe AZALBERT - @Phil_BARR3TT

Quarkslab

Training materials

Materials for this training are available at:

https://github.com/phil-eqtech/CH-Workshop



Workshop Agenda

- Automotive Security overview
- CAN bus hands-on
- ECU HW/SW reverse engineering
- Bonus track: fun w/ RF and TPMS

Quarkslab

Overview of Automotive Security

Quarkslab

ECUs, the cores of a vehicle

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**.

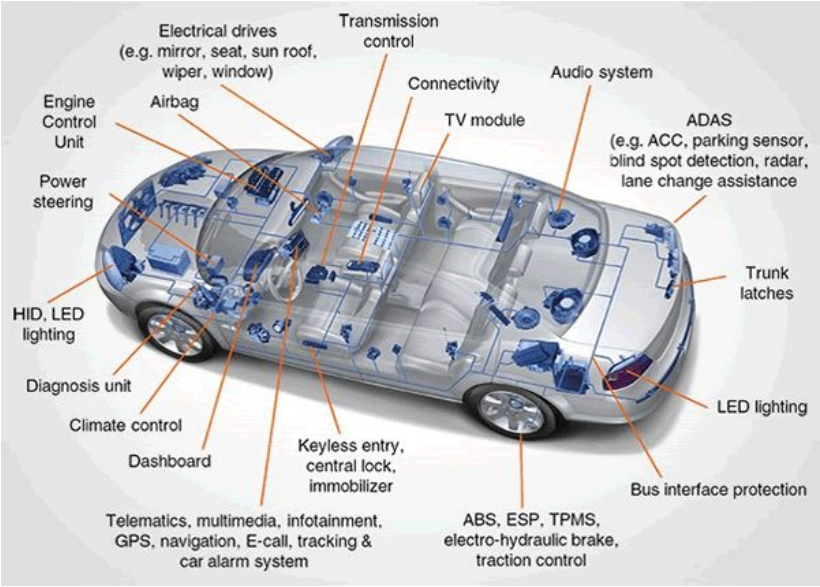


Illustration: [link] [link]

ECUs communications: multiple internal networks...

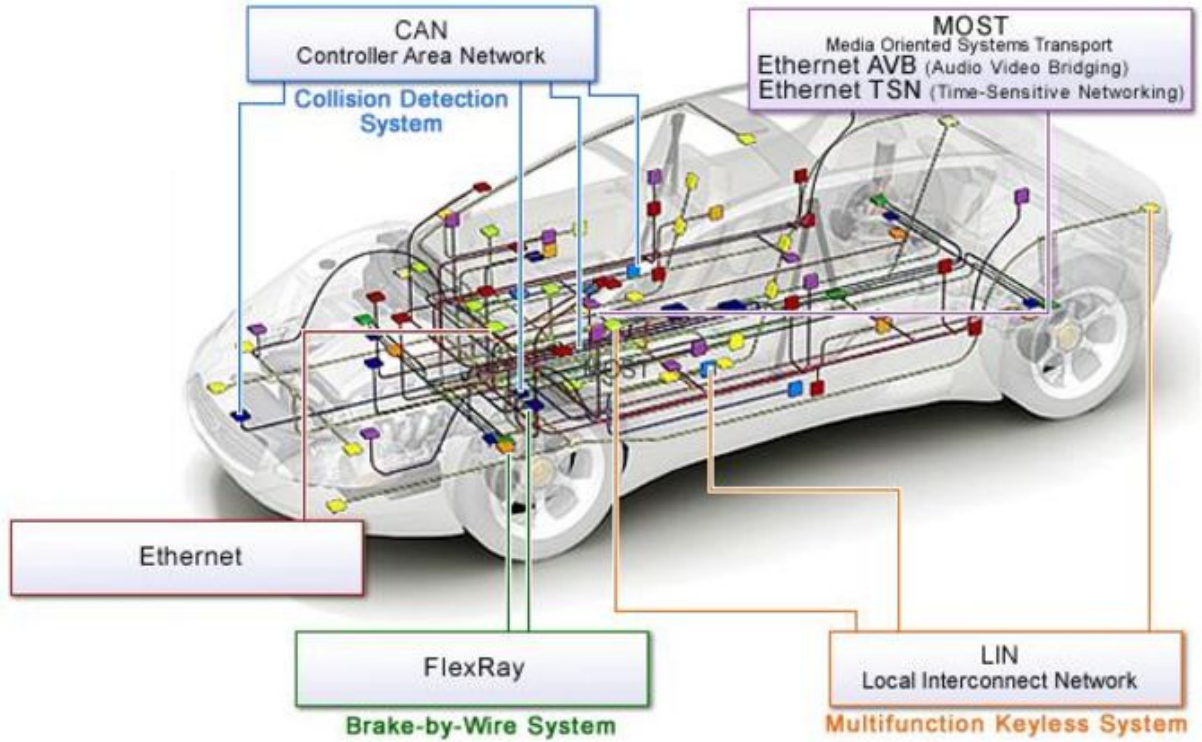


Illustration: [link]

.. and several attack surfaces

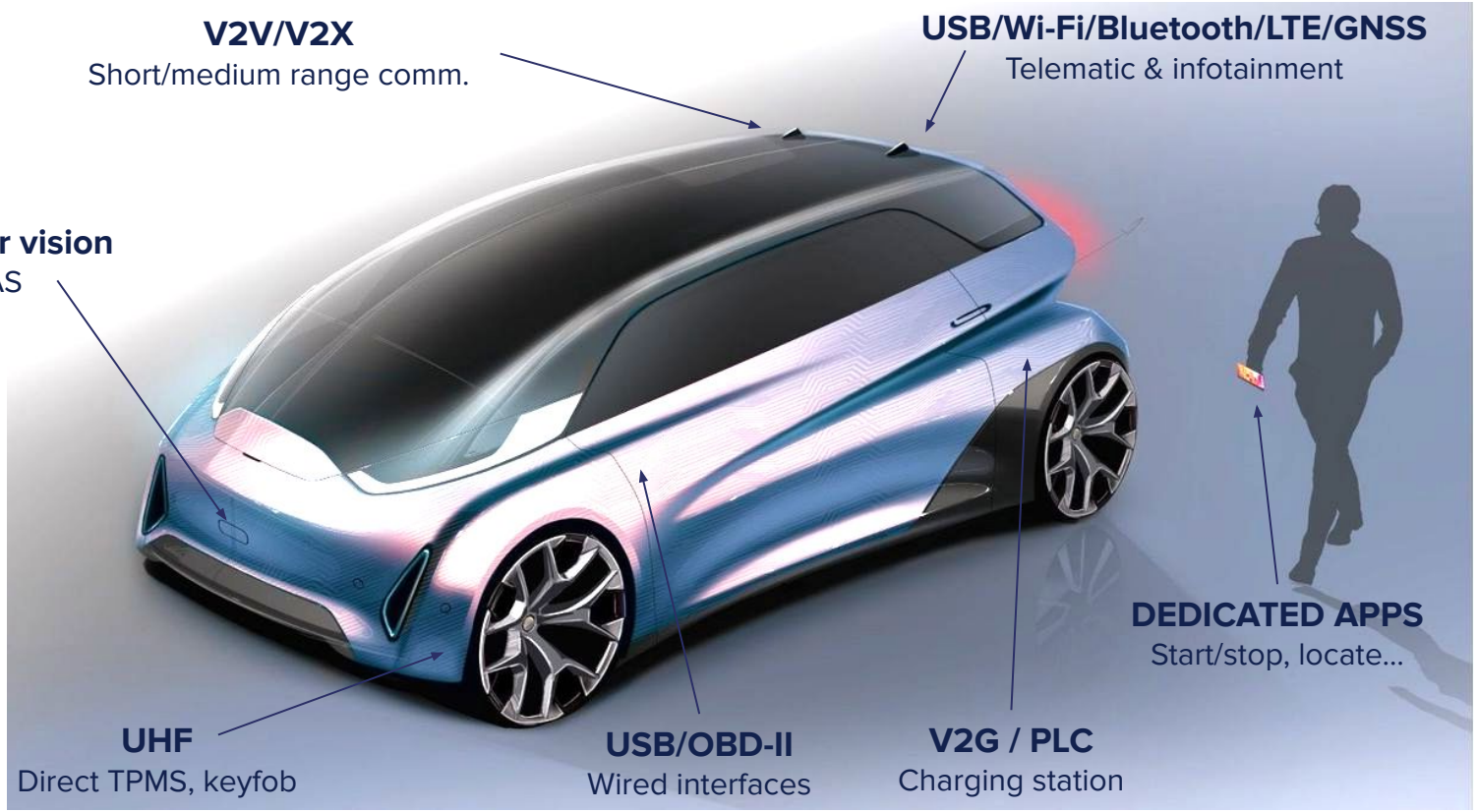


Illustration: [link]

Car network overview

Controller Area Network (ISO 11898)

- **Two wires** half-duplex network protocol with **8 bytes** of payload and speed up to **1 Mbit/s**
- **Fault resistant** protocol
- **CAN-FD** (Flexible Data Rate) allows speed up to **8 Mbit/s** and **64 bytes** of payload
- **CAN-XL**: speed of **20 Mbit/s** and **2048 bytes** of payload, able to encapsulate Ethernet traffic
- **Most commonly** used in-vehicle network to connect **ECU**s

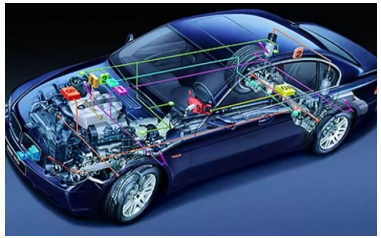


Illustration: [link]

Car network overview

CAN - a reliable but not secure network

- **CAN** has no native protection against **replay attacks** or **forged packets**
- No native **encryption**
- However, the **Secure On-board Communication** (SECOC) tends to mitigate those knows weaknesses

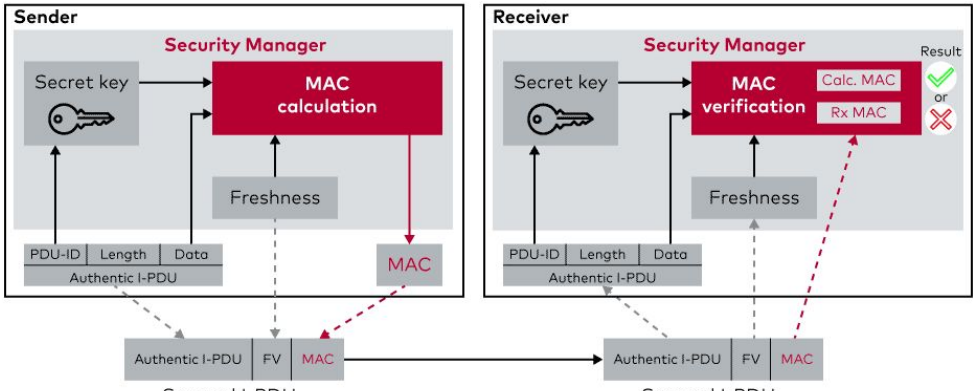


Illustration: [link]

Car network overview

CAN - network segregation

- **CAN** networks are segregated, using a **gateway** to secure in-vehicle networks from unsecure connectivity (LTE, OBD-II), which was not often before the 2015 “Jeep Hack”.

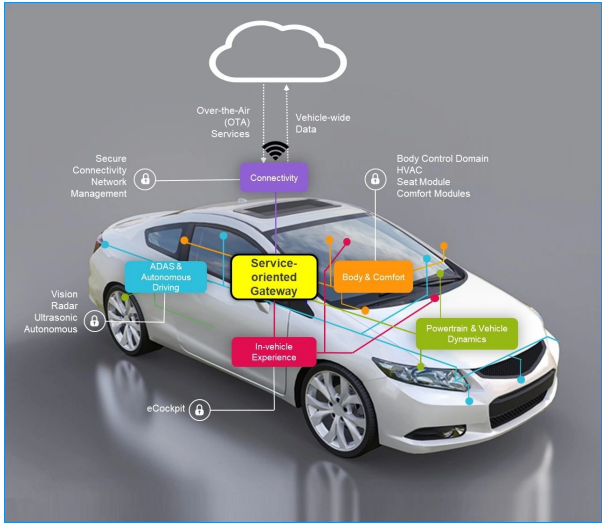


Illustration: [link]

Car network overview

Automotive Ethernet

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

Illustration: [link]

Car network overview: Automotive Ethernet

Automotive Ethernet - multiple point-to-point networks

- **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**

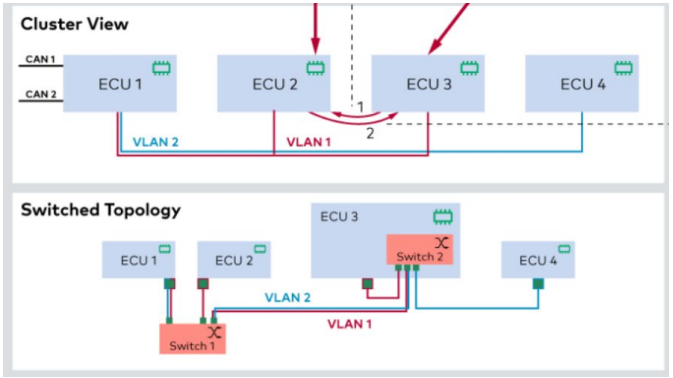


Illustration: [link]

Car network overview: electrical vehicles connectivity

Advanced charging capabilities

- **Electric vehicles** must communicate with the charging station
- **Basic communication** is done using **PWM** signaling on the **Control Pilot**
- **Chademo** plug integrates a **CAN** connectivity
- **CCS** plug uses **PLC** protocol (GreenPhy Home plug), allowing a full ethernet connectivity

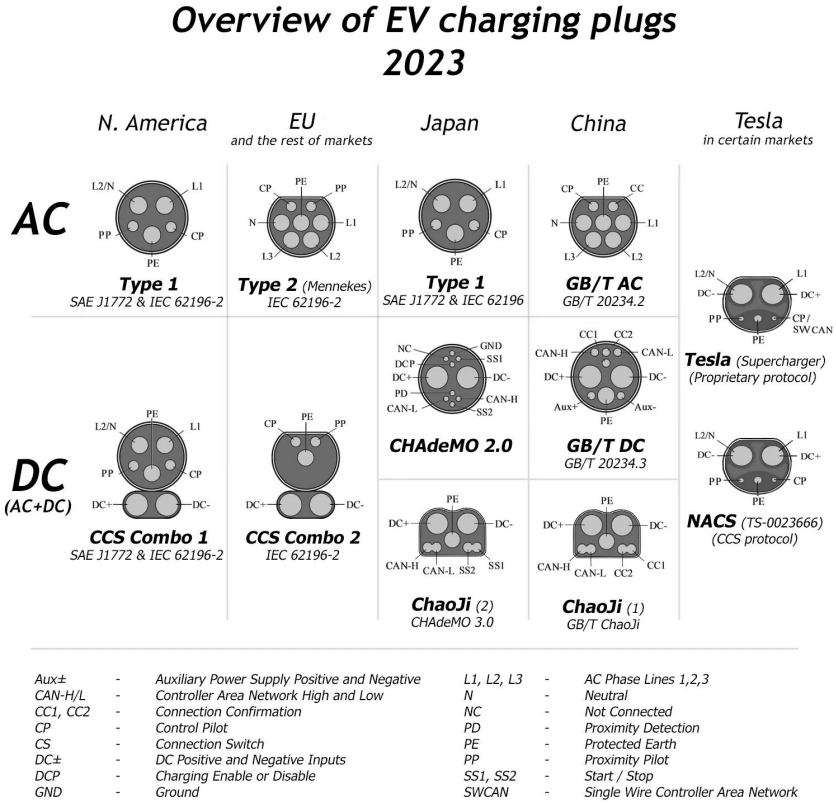


Illustration: [link]

Car network overview: electrical vehicles connectivity

Plug n' Charge: allowing secured and automated charging (ISO 15118)

- Designed to allows **confidentiality, data integrity and authenticity**
- Relies on several PKIs: **Charge Point Operator** (CPO), **Certificate Provisionning Service** (CPS), **Mobility Operator** (MO), **Car Manufacturer** (OEM)
- **ISO 15118** requires a mandatory **TLS communication**

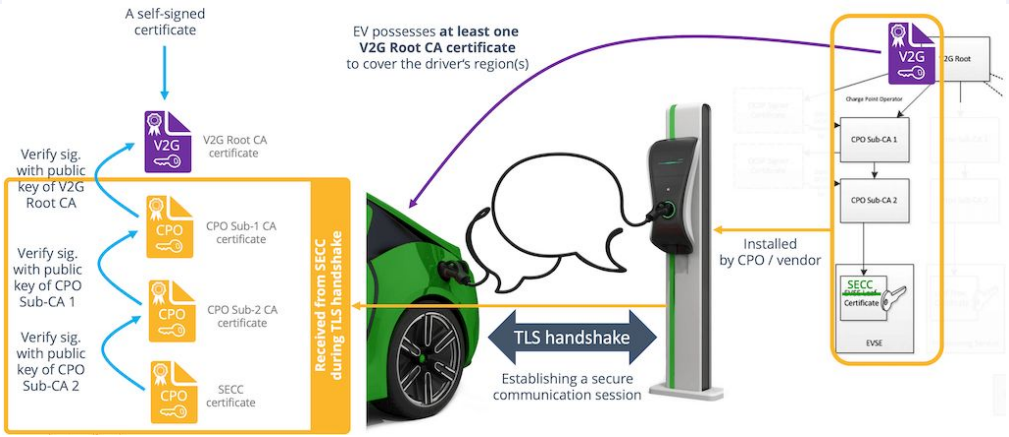


Illustration: [link]

Targets of interest

Telematic ECU - TCU

- Handles **backend** connectivity and **eCall** services (mandatory in Europe)
- Connects to **LTE networks**, sometimes exposes a **Wi-Fi hotspot**
- Offers **WAN** connectivity to others ECUs through **Automotive Ethernet** or **USB-ETH** networks

Illustration: [link]

Targets of interest

Gateway ECU

- Acts as a **firewall** and **IDS** to protect other ECUs and detect invalid traffic on the CAN bus
- **Backbone** of the **in-vehicle security**, compromising it allows to **pivot** from one CAN bus to another
- They are connected to the **OBD-II** port and (almost) every CAN bus

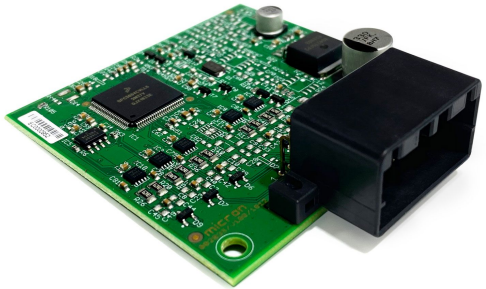


Illustration: [link]

Targets of interest

Infotainment unit (IVI)

- A complex ECU running a “classical” **OS** (Automotive Grade Linux, QNX, Android…)
- Present a large amount of **attack surfaces**, via the different connectivity: Bluetooth, Wi-Fi, USB…
- There are often a secret “Engineer Mode”, allowing access to specific features or logs

Illustration: [link] [link]

Targets of interest

UDS - Diagnostic protocol

- **UDS** (Unified Diagnostic Services) protocol is **mandatory** in every road vehicles
- It allows **configuration**, **diagnostic** of each ECU but also **firmware update** and much more…
- Finding **development** Services can give **advanced access** to an ECU (TLS downgrade, security bypass…)
- It’s a verbose protocol, the **Negative Response Code** helps understanding what’s supported

Illustration: [link]

Target of interest

UDS - useful services

- **0x10:** Diagnostic Session Control
- **0x11**: ECU Reset
- **0x27**: Security Access
- **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

UDS - Negative Response Code (NRC)

- **0x10**: General Reject
- **0x11**: Service Not Supported
- **0x12**: Sub-function Not Supported
- **0x13**: Invalid Message Length or Invalid Format
- **0x22**: Conditions Not Correct
- **0x10**: 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

Targets of interest

Mobile applications / API accesses

- Most of recent Car Hacking vulnerabilities are related to unsecure API applications:

- In 2016, a research showed an unsecure **API** used to control remotely a car from a mobile application (**Source**: https://www.troyhunt.com/controlling-vehicle-features-of-nissan/), the **API** was not encrypted and the **VIN** was used as a token to control a car
- More recent example with unsecure **JWT** usage: DEFCON 29 - Car Hacking Village Keynote
- Even more recent example about Honda: DEFCON 30 – Car Hacking Village
- Could it be a never ending story: 18 manufacturers affected by API hacking fall 2022

- **Effects**: an attacker could open, start or stop a car remotely

Illustration: [link]

Evolutions - ISO 21434

UN Regulations R155 & R156 refer to ISO 21434

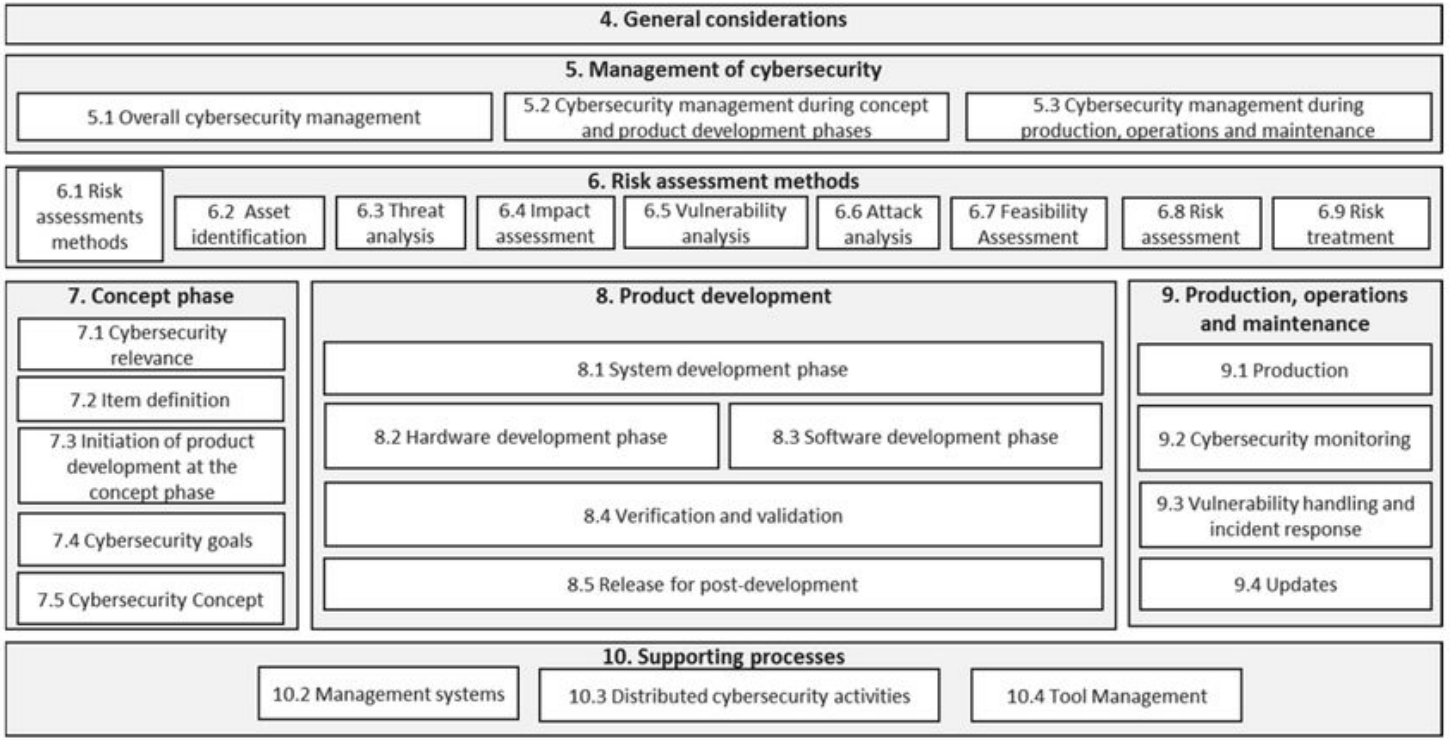


Illustration: [link]

Evolutions - 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**

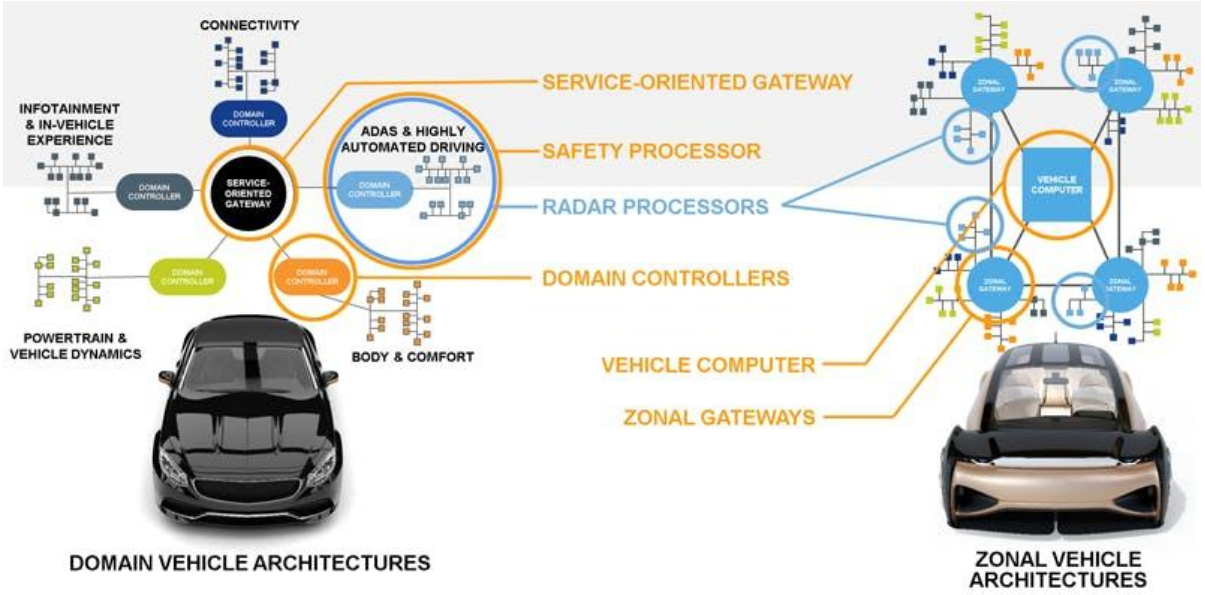


Illustration: [link]

AutoSAR

AUTomotive Open System Architecture

- Development partnership of **automotive parties** (manufacturer, Tier 1 suppliers) founded in 2003
- Defines a **standardized software architecture** for ECUs, **methodology** and **procedures**

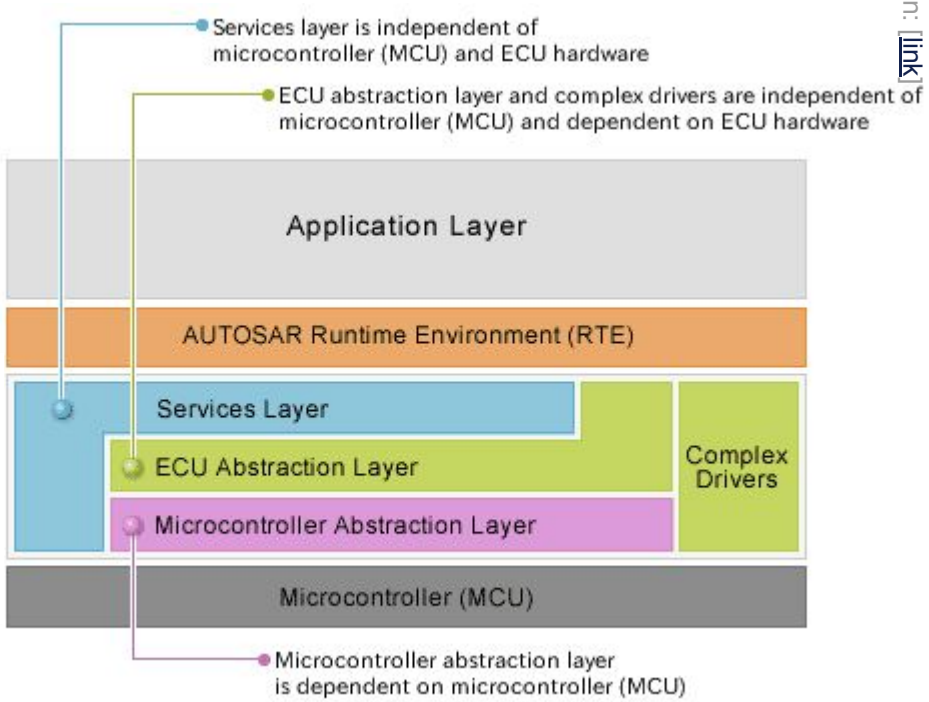


Illustration: [link]

CAN bus hands-on

Quarkslab

CAN 101

Main concepts

- A **CAN** message is composed of :
  - An **Arbitration ID** (0x000-0x7FF, up to 0x1FFFFFF for extended **Arbitration ID**)
  - A payload, from 0 to 8 bytes on a classic CAN bus (64 on CAN-FD)

- An **ECU** uses several **Arbitration ID**, the lower the ID the higher the priority of the message every message going through the bus

- CAN messages are usually send on a **periodic interval**, some messages can be send on event

CAN 101

Main concepts

- A **CAN** message is composed of :
  - An **Arbitration ID** (0x000-0x7FF, up to 0x1FFFFFF for extended **Arbitration ID**)
  - A payload, from 0 to 8 bytes on a classic CAN bus (64 on CAN-FD)

- An **ECU** uses several **Arbitration ID**, the lower the ID the higher the priority of the message every message going through the bus

- CAN messages are usually send on a **periodic interval**, some messages can be send on event

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 vcan0

vcan0  TX - -  2e0   [3]  02 06 64                  '..d'
vcan0  TX - -  130   [3]  02 06 64                  '..d'
vcan0  RX - -  121   [2]  01 46                     '.F'
vcan0  RX - -  124   [2]  01 46                     '.F'
vcan0  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

Finding specific CAN message

Cansniffer

- As it is difficult to observe changes or follow multiple messages with candump, an other tool exists, **cansniffer**
- Using the option **“-c”**, it colors any changing byte in a message
- Recurrent **unmodified** message are **hidden** after a few seconds

```
$ cansniffer -c vcan0

32|ms        | ID    | data …        < can0 # l=20 h=100 t=500 slots=2 >
01002        | 123 | EF 00 AD 12              ....
01101        | 3AA | 00 01 02 03              ....
01102        | 620 | 00 11 22 33 44 55 A3 …!,7.
```

Write on a CAN bus

Send simple message

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

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

- **Mandatory arguments :**
  vcan0: 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 vcan0 321#
```

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

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

Cangen

Poor man fuzzer

- Can-utils include **cangen** to generate message with a specific interval
- **Arbitration ID**, **length** and **data** can be replaced by **i** or **r** to get **incremental** or **random** values
- Example: to fuzz every possible message for a specific arbitration ID

```
$ cangen -I 123 -L 4 -D i -g 100 vcan0
```

Warning !

- Fuzzing a **real car** can “brick” it, due to the **crash frame**
- Avoid fuzzing arbitration ID below **0x400**

Recording/replaying a bus

Candump - recording

- **Candump** has an option “-l” (minus L) to record all the frames
- Filtering rules apply when recording

```
$ candump -l vcan0,123:7FF
Disabled standard output while logging.
Enabling Logfile 'candump-2022-04-12_154654.log'
```

- Log files are simple text files containing the following data

```
$ cat candump-2022-04-12_154654.log
(1605875935.822384) vcan0 123#8C0001FFFFFFFFFF
(1605875935.830851) vcan0 123#000000000000
(1605875935.831253) vcan0 123#CE0CE000
```

Recording/replaying a bus

Candump - replaying

- Log files are useful for **offline** analysis, but also can be **replayed**
- **Canplayer** allows such functionality

```
$ canplayer -I candump-2022-04-12_154654.log -l i vcan0=vcanX
```

- **Options:**
  -I (upper i): mandatory, defines the can log to be loaded
  -l [num] (lower l): processes the log num times. Use '-l i' for infinite loop
  vcan0=vcanX: optionnal. Interface assignments like <write-if>=<log-if>, could be multiple
- You can also use **cansniffer** and **candump** with **canplayer**.

ISO-TP

Extend payload size

- The **ISO-TP** protocol allows to send payloads up to **4096** bytes
- It **fragments** the message, using **Single Frame** (0x), **First Frame** (0x1x), **Flow Control** (0x3x) and **Consecutive Frames** (0x2x)

```
$ candump -a vcan0,7e0:700
vcan0  7e0   [8]  02 09 02 00 00 00 00 00        '........'
vcan0  7e8   [8]  10 14 49 02 41 42 43 44        '..1.abcd'
vcan0  7e0   [8]  30 00 0A 00 00 00 00 00        '........'
vcan0  7e8   [8]  21 45 46 47 48 48 50 51        '.efghijk'
vcan0  7e8   [8]  22 52 53 54 55 56 57 58        '.lmnopqr'
```

Data

Frame type

UDS protocol

Diagnostic over CAN

- **UDS** requests are sent using **ISOTP**
- The **second byte** defines the response level
  - **Requested service + 0x40** = **positive** response. The 3rd byte will be the requested Subfunction
  - Value **0x7F** = **negative** response. The 3rd byte is the requested service and the 4th byte describe the error using a **Negative Response Code (NRC)**
- Example: **positive response**

```
vcan0  7e0   [8]  02 01 2F 00 00 00 00 00
vcan0  7e8   [8]  03 41 2F 59 00 00 00 00
```

- Example: **negative response** with **NRC** 0x12 (PID not supported)

```
vcan0  7e0   [8]  02 01 2F 00 00 00 00 00
vcan0  7e8   [8]  03 7F 01 12 00 00 00 00
```

Playing with our custom IC Sim

Goals

- Using our custom **ICSIM**, can you:
  - Find the **Arbitration ID** for the speed
  - Unlock the **doors**
  - Switch the **beams** off during the night
  - Craft your first UDS message to request the **RoutineControl** Service (0x31), changing the **DataIdentifier** to 0x4122

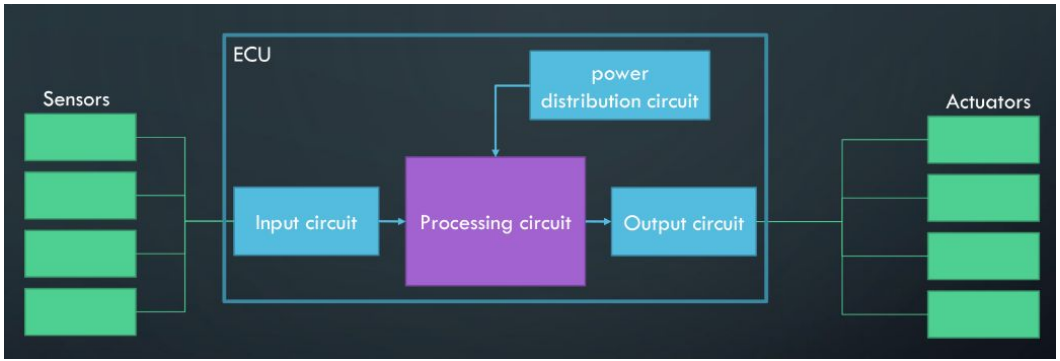
ECU HW/SW Reverse Engineering

Quarkslab

ECU reverse engineering

ECU inner workings

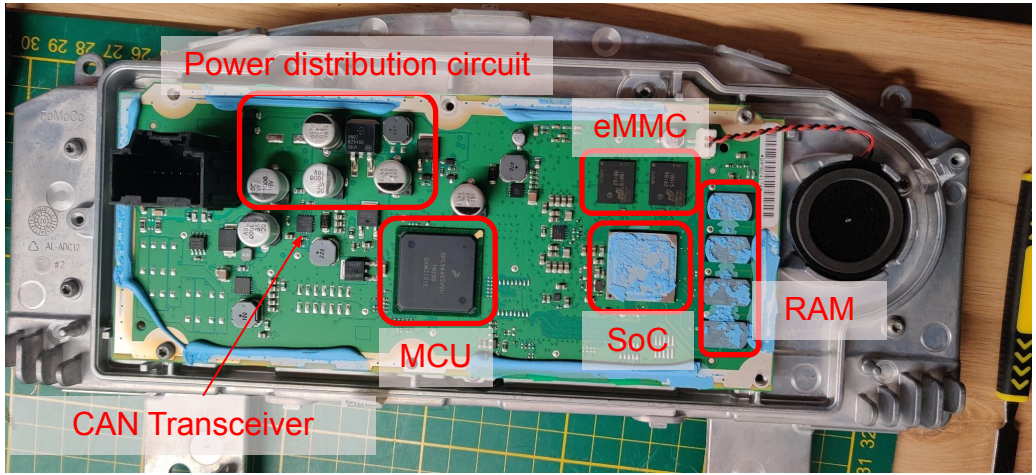
- An **ECU** will have one or multiple **microcontrollers (µC)** or **system-on-chip** (**SoC**), each running its own **firmware**/**OS**
- **µC/SoC** are powered via a **Power Distribution Circuit** and handle multiples **sensors** and **actuators** through input/output chips, like a CAN transceiver for example



ECU reverse engineering

ECU internal example #1 – Modern Instrument Cluster

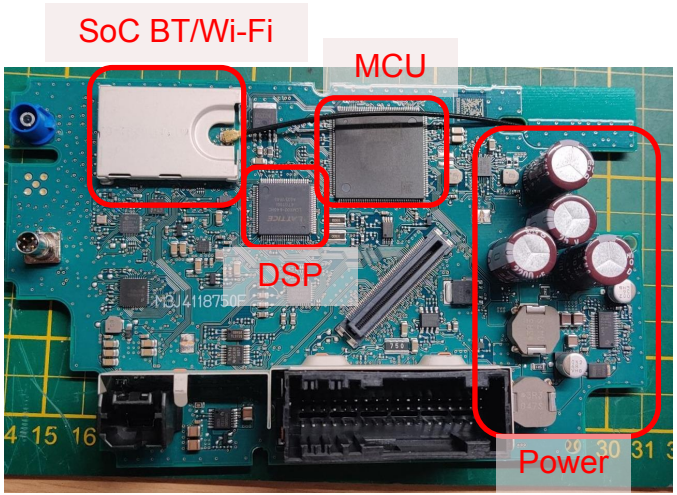
- **MCU:** PowerPC SPC58 handles the CAN communication and most of the GPIO
- **uC/SoC**: IMX.6 SoC running QNX manage the display. The OS is stored on one of the two eMMCs

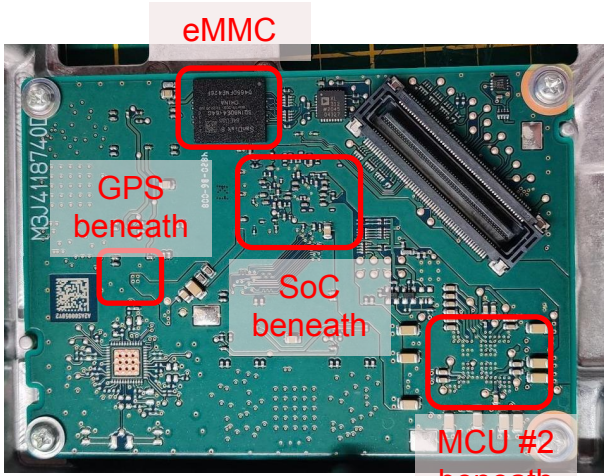


ECU reverse engineering

ECU internal example #2 – IVI

- Infotainement unit are one of the most complex **ECUs**, having several **SoCs** for the OS and the various radio protocol (Bluetooth, Wi-Fi,...)





ECU reverse engineering

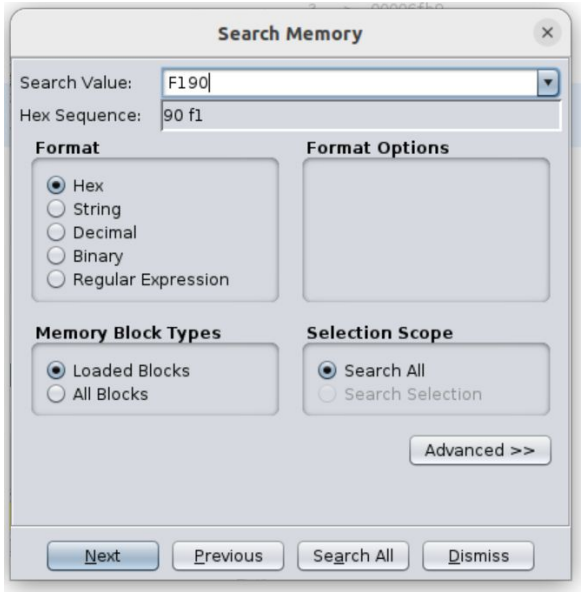
1st step: getting the firmware

- An **ECU** will have one or multiple **firmware(s)**, depending on how many **µC/SoC** are inside
- For **SoC**, the **firmware/OS image** could be stored inside an **external Flash memory**
- Each **µC/SoC** will have a **debug port** on which you may read its firmware, however such ports are often **disabled/secured**
- You could find firmware images online on **chip-tuning/reprogramming forums**. However, dumps are sometime **incomplete as** only a part of the memory was read
- Having access to a **diagnostic tool** or looking at the **manufacturer website** could provide firmware through **updates**, but they could be **encrypted**

ECU reverse engineering

Reverse engineering the firmware - DID database

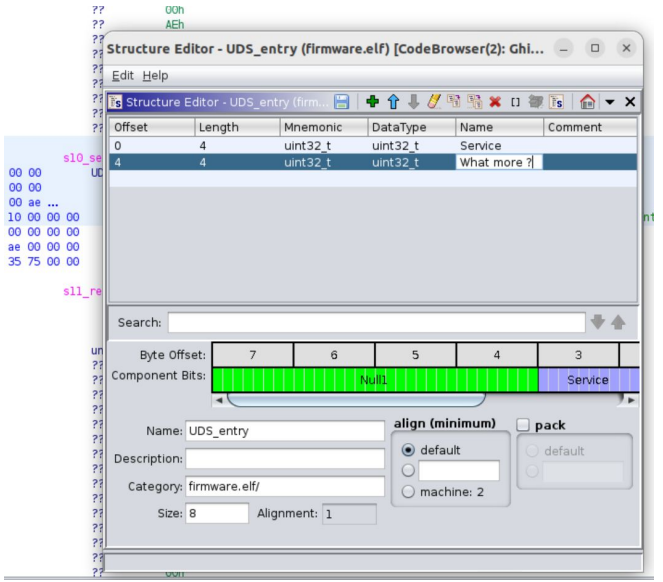
- **DIDs** supported by **ReadDataByIdentifier** Service are a good hint to find references to their functions and associated data
- It is common that **DIDs** are stored as an array of struct, including theirs DID value and a pointer to a function managing it
- Typically, search for value **0xF190 (VIN DID)** and analyze if you see a pattern
- Try to create the correct struct and apply it to each **DID**
- You may have 2 kinds of struct, one for the **RDBI** and one for the **WDBI**



ECU reverse engineering

Reverse engineering the firmware - UDS database

- The same applies for the **UDS Services**: they are commonly available in a DB, which is an array of struct
- Using one off the DID function you found, try to cross-ref to the DID handler and search for the start address in the memory
- **Binbloom** [https://github.com/quarkslab/binbloom] allows the detection of the UDS database in many FW
- Look around to identify the different supported **UDS Services**, try to guess the correct struct and label all the functions



ECU reverse engineering

UDS function identification

- As the **UDS** protocol is verbose, you can also look for **Negative Response Code** to identify or confirm such functions

- Examples:
  0x10: General Reject
  0x11: Service Not Supported
  0x12: Sub-function Not Supported
  0x13: Invalid Message Length or Invalid Format
  0x22: Conditions Not Correct
  0x10: Request Out Of Range
  0x33: Security Access Denied
  0x35: Invalid Key

```
int iVar1;
undefined uVar2;

if ((bRamd000f36d == (bRamd000f36c ^ 0xff)) && (bRamd000f36c == 0x3c)) {
  uVar2 = 0;
  bRamd000f36c = bRamd000f36d;
}
else if (*(char *)(*(int *)(param_1 + 0x18) + 8) == '\0') {
  iVar1 = (*(code *)(*(uint *)(*(int *)(param_1 + 0xc) + 0x30) & 0xfffffffe))(6);
  if (iVar1 == 0) {
    if (**(short **)(*(int *)(param_1 + 0x1c) + 0x20) == 2) {
      if (*(char *)(**(int **)(*(int *)(param_1 + 0x1c) + 0x28) + 1) == '\x01') {
        uVar2 = 0x78;
        *(undefined *)(*(int *)(param_1 + 0x18) + 8) = 1;
        **(undefined **)(*(int *)(param_1 + 0x1c) + 8) = 0x78;
      }
      else {
                    /* NRC: subfunction not supported */
        uVar2 = 0x12;
      }
    }
    else {
                    /* NRC: Incorrect Message Length or Invalid Format */
      uVar2 = 0x13;
    }
  }
  else {
                    /* NRC: Conditions Not Correct */
    uVar2 = 0x22;
  }
}
```

Reversing your first ECU firmware

Goals

- The provided firmware is from a **Tricore TC166** architecture, base address is **0x8000_0000**
- Load it in your favorite RE tool and try to:
  - Locate the **UDS** database (using binbloom is a good thing 😉)
  - Identify **NegativeResponseCode**
  - Look for **DID 0xF187** and identify a DID database structure

Illustration: [link]

Bonus track: fun with RF and TPMS

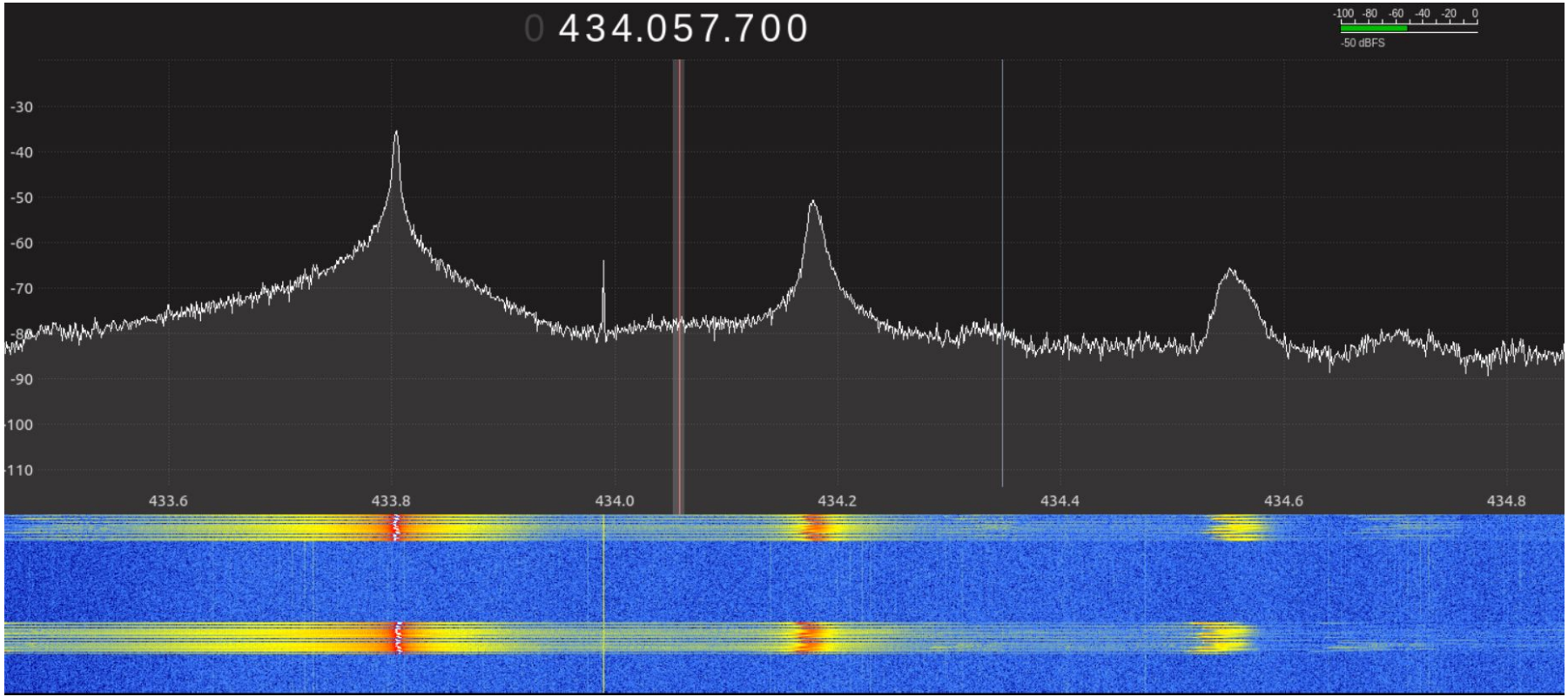
Quarkslab

Tire Pressure Monitoring System

Connected Tires

- Car using direct **TPMS** (Tire Pressure Management System) have a sensor in each tire
- As there is **no encryption**, it is quite simple to transmit fake messages

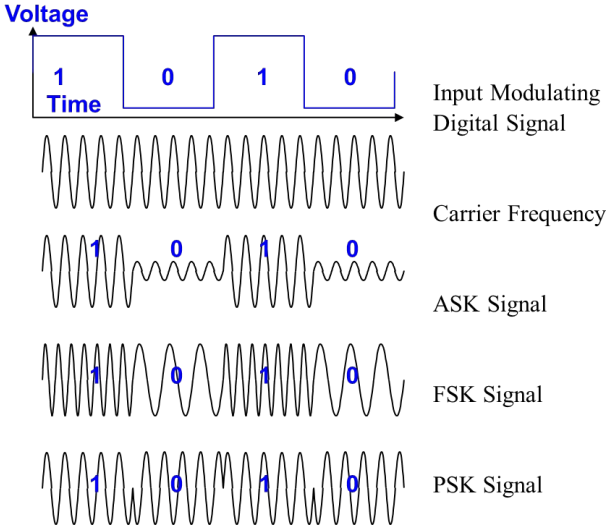
Tire Pressure Monitoring System

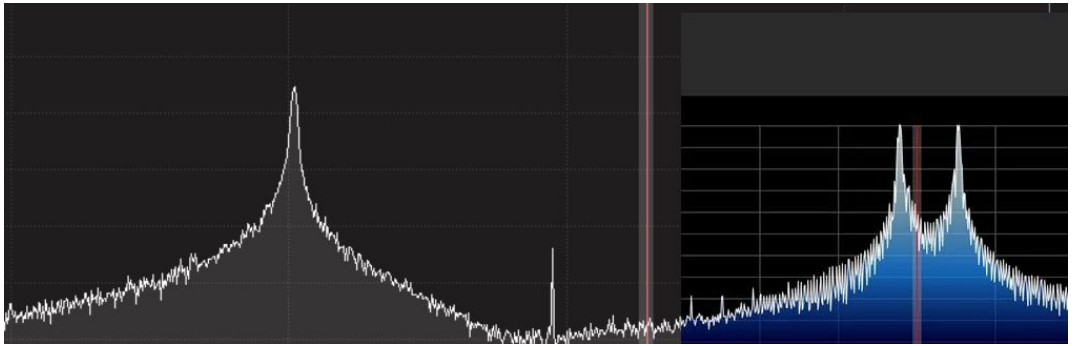


Tire Pressure Monitoring System

Identifying the modulation

- Those sensors use basic modulation, like **ASK**, **FSK**





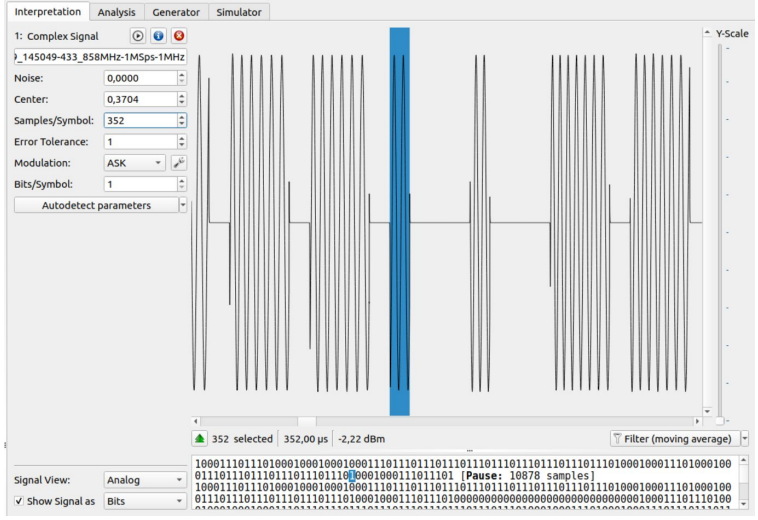
ASK **FSK**

Illustration: [link]

Tire Pressure Monitoring System

Decoding the message

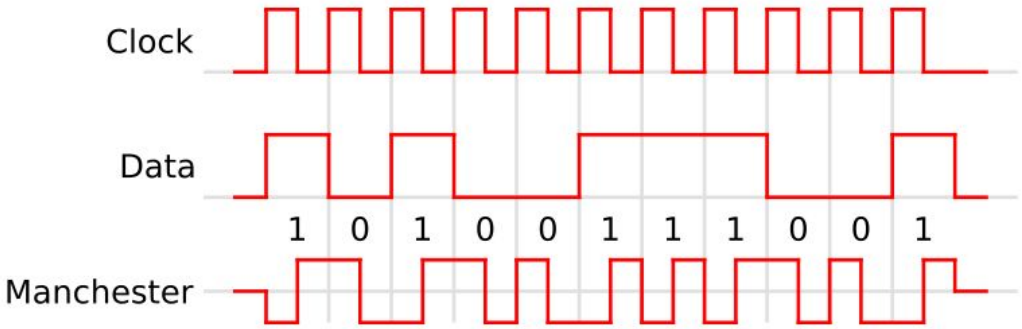
- Tool like **URH** (Universal Radio Hacker) simplifies the decoding work
- You'll need to find the correct **modulation** and **bitrate** of your signal



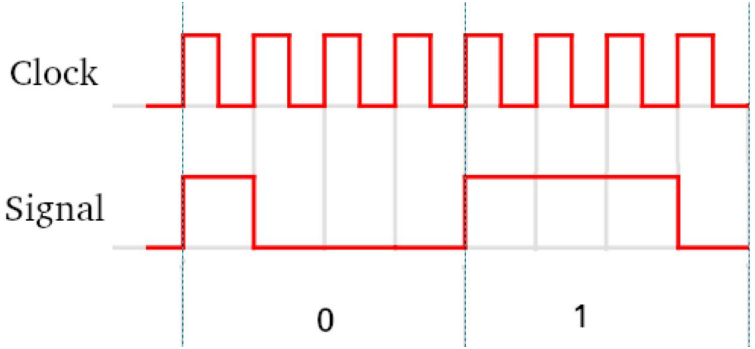
Tire Pressure Monitoring System

Decoding the message

- To ensure **data integrity**, as RF messages are asynchronous, you'll find:
  - A **preamble**, to wake-up the receiver
  - Most of the time an **encoding scheme,** to alternate 0/1 and simplifies resynchronization
  - A **CRC**



Manchester encoding



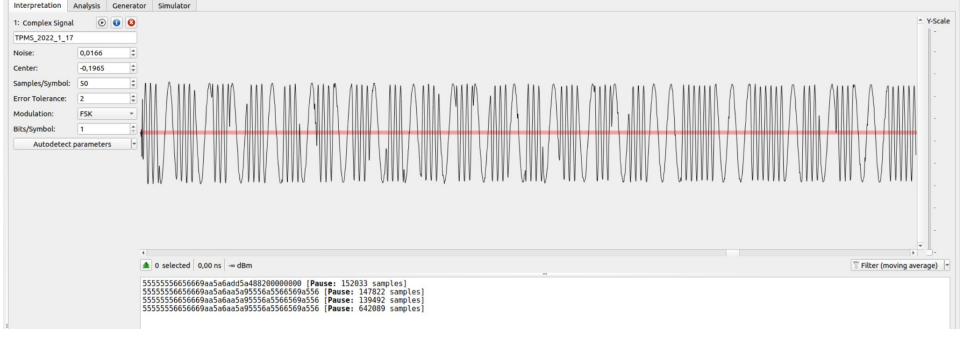
Long/short pulse substitution

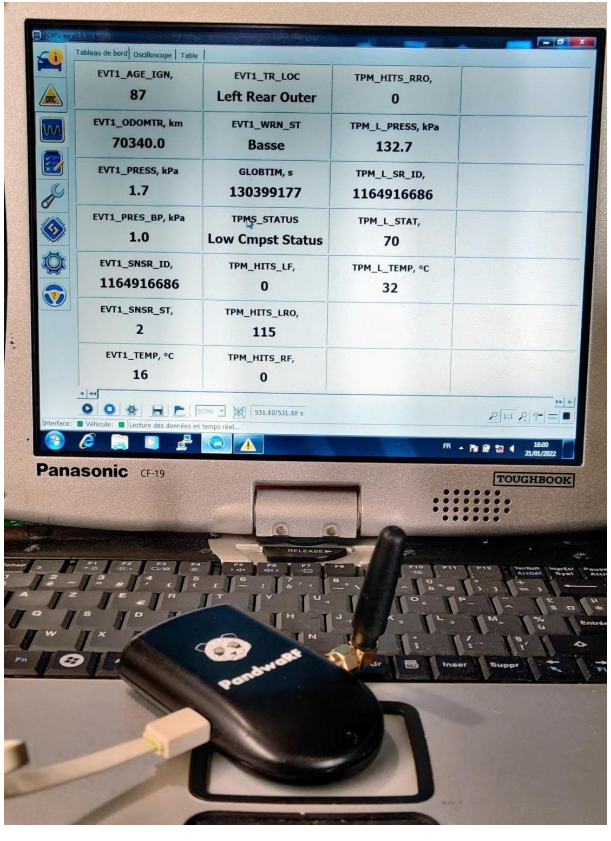
Illustration: [5]

Tire Pressure Monitoring System

Spoofing TPMS message

- Using compatible hardware, it simple to craft fake **TPMS** signals
- Using **URH/GNURadio** and supported SDR transmitter
- **Rfcat** with **CC1101** (yardstickone, Pandwarf) or an EvilCrow RF





Tire Pressure Monitoring System

Goals

- Analyse the provided capture of a **TPMS** signal:
  - Try to find the id of the **TPMS** on the signal (ASIC: 456F37CE)
  - Looking at **rtl_433** repo can help you understanding the various part of the signal:

    https://github.com/merbanan/rtl_433/blob/master/src/devices/tpms_ford.c

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: www.quarkslab.com

@quarkslab

Quarkslab