2020 IEEE Symposium on Security and Privacy

≪ I Know Where You Parked Last Summer ≫
Automated Reverse Engineering and Privacy Analysis of Modern Cars

Daniel Frassinelli*, Sohyeon Park† and Stefan Nürnberger‡
CISPA Helmholtz Center for Information Security,
Saarland Informatics Campus, Saarbrücken, Germany
Email: *daniel.frassinelli@cispa.saarland, †sohyeon.park@cispa.saarland, ‡nuernberger@cispa.saarland

***Abstract*—Nowadays, cars are equipped with hundreds of sensors and dozens of computers that process data. Unfortunately, due to the very secret nature of the automotive industry, there is no official nor objective source of information as to what data exactly their vehicles collect. Anecdotal evidence suggests that OEMs are collecting huge amounts of personal data about their drivers, which they suddenly reveal when requested in court.**

In this paper, we present our tool `AutoCAN` for privacy and security analysis of cars that reveals *what* data cars collect by tapping into in-vehicle networks and extracting time series of data and automatically making sense of them by establishing relationships based on laws of physics. These algorithms work irrespective of make, model or used protocols. Our results show that car makers track the GPS position, the number of occupants, their weight, usage statistics of doors, lights, and AC. We also reveal that OEMs embed functions to remotely disable the car or get an alert when the driver is speeding.

***Index Terms*—Automotive privacy & security, telematic control unit, reverse engineering, CAN.**

I. Introduction

Cars have ceased to be purely mechanical devices since their computerised counterparts are usually cheaper to manufacture and provide more functionalities. Even in the entry-level segment, modern cars feature at least ten different computers, so-called *Electronic Control Units* (ECUs), and over 50 in luxury sedans. Some ECUs are mandated for active passenger safety (e.g., airbag ECU), or are used to enhance comfort and provide infotainment. Most ECUs take actions based on sensors like temperature, light, pressure, or user input such as switches or touch screens. Due to the plethora of sensors, a modern car easily collects thousands of data points a second.

This data can be accessed locally through the standardised On-Board Diagnosis (OBD) port. Also, with the addition of the Telematic Control Unit (TCU), this data is often transmitted to the cloud. Furthermore, OEMs use the car's Internet connectivity for cabin pre-heating, door unlocking and remote start. There are also 3G/4G dongles for OBD available that process the car's data for insurance purposes [1], [2] and remote diagnosis [3].

In the public, there is little awareness about data collection in cars as this collection happens unnoticeably. To the best of our knowledge, no OEM informs the occupants about *which* data is collected, *where* it is transmitted, *who* has access to it, and *what* it is used for. However, there have been cases where OEMs refuted customers' claims based on such collected data. For example, TESLA and BMW provided evidence in court after the defendant's car had already been destroyed in an accident because the car transmitted data about the exact speed and position at the time in question [4], [5]. Over the years, researchers have demonstrated that it is possible to fingerprint a driver based on brake pedal usage [6] and to understand the distraction level based on steering wheel movement [7]. A lot more sensors are present in modern cars that can be used to infer sensitive information such as speeding, driver identity, typical pick-up points, or simply profiling.

However, an analysis of data collection and usage is difficult as the used protocols and encodings are proprietary and obfuscated. While the physical/link layer is based on standards like CAN, the payloads are undocumented, differ even between models of the same brand and are usually considered intellectual property. Reverse engineering them is error-prone and labour-intense as it requires driving, testing, manual inspection and the outcome might only be valid for one specific model. To make matters worse, encodings are often squeezed in the least amounts of bits possible, differ in endianness and have to be post-processed by factors and offsets to give meaningful physical values. Hence, the possible interpretations of the sniffed bits grow exponentially.

A. Contributions

In this paper, we provide algorithms to automatically make sense of unstructured data that describes a physical process, like driving. To further substantiate our findings, we reverse engineer the software of a modern Telematic Control Unit, which collects, processes and transmits data in a connected car. This reveals *which* data is collected, *how often* and to *whom* it is sent. We show that a CAN log without prior knowledge of make or model suffices to derive personal information or the technical information needed to mount attacks like [6]–[10]. Besides the privacy aspect, our approach can be used as the basis for security research and attacks on cyber-physical systems since it delivers the necessary knowledge to make attacks more platform-independent. In short,

- we develop an open-source[1] tool which is able to automatically extrapolate structured interpretations of CAN data,
- we demonstrate that a passive, non-knowledgeable attacker can easily infer private data from CAN logs *without* requiring physical access to the car,

[1] We will publish the code after submission.

DOI 10.1109/SP40000.2020.00081

- we evaluate our approach by comparing `AutoCAN` findings with manually reverse engineered ground-truth on four different vehicles,
- we examine the security of a state-of-the-art telecommunication unit of a modern car, and
- we conduct an evaluation of which remote capabilities a manufacturer has over its vehicles.

II. Background

A modern car can be seen as a distributed system where (1) *sensors* gather data (e.g. accelerometer), (2) one or more ECUs take actions based on this data (e.g. stability control), and finally (3) *actuators* receive and execute commands from the ECUs (e.g. brake front left wheel). As a means to reliably interconnect sensors, controllers, and actuators, several automotive network standards exist. The most widespread technologies are listed in Table I. Our algorithms are independent of the used protocol or network. In this paper, we focus on CAN as it is by far the most widespread network used in cars today.

Table I: Automotive Network Protocols

Technology	**Medium**	**Topology**	**Speed [kbit/s]**
LIN	single wire	bus	19
CAN	unshielded twisted pair	bus	1,000
FlexRay	unshielded twisted pair	star & bus	10,000
MOST	plastic optical fibre	ring	46,000
Automotive Ethernet	unshielded twisted pair	star	100,000

A. CAN Bus

CAN (Controller Area Network) was proposed by BOSCH in 1983. Due to its safety and real-time assurances, it became the de-facto standard used in the automotive industry, with various application-layer protocols built on top [11].

The CAN bus is a synchronous multi-master serial bus where each component is connected to the same transmission medium and broadcasts its messages – or *frames* in CAN jargon. Each frame can contain up to 8 bytes, whose payload is not standardised and is specific to the software running on each ECU. Typically, a CAN frame is periodically transmitted and contains so-called *Signals*: encodings of internal variables such as speed, temperature, or status bits. If the length of all transmitted signals exceeds the maximum 8 bytes, a transport layer protocol such as ISO-TP is used [12]. A core feature provided by the CAN bus is the arbitration of sending order. This is achieved by assigning a priority to each frame, which is used during transmission to prioritise high-priority frames over low-priority ones. This priority is at the same the frame ID of a message. An ECU can use multiple frame IDs and can therefore have multiple priorities. The ID is also used by the receivers to decide whether they are interested in the frame or not. Figure 1 shows a typical CAN network with several messages that are exchanged. In the example, the *Window Control* ECU sends a low-priority control frame (ID = `0xE3`) to the rear window to tell it to open or close. However, if the *Domain Controller* detects a possible collision while sending a high-priority steering command (ID = `0x71`) to the Electronic Power Steering (EPS), this frame would get precedence over the *Window Control* frame since it has a higher priority.

Manufacturers may embed one or more CAN buses inside their vehicles, for reasons like (1) separation of roles for safety reasons, (2) fault-tolerance, and (3) cost reduction by using lower speed for less critical ECUs. These buses are typically connected using a gateway ECU that forwards frames from one bus to another.

B. Car connectivity

Modern cars usually feature a special ECU, the *Telematic Control Unit* (TCU). The TCU is connected to the in-vehicle network (e.g. CAN) and provides 2G, 3G, or 4G connectivity. The TCU allows vendors to provide updates over-the-air, additional safety and commodity features, and remote connections via smartphone apps [13]–[16]. The TCU is also used to collect usage and diagnostic statistics (e.g., battery usage for electric cars).

III. Problem Statement & Assumptions

Enev et al. use machine learning to fingerprint a driver: after just one hour of driving and by using only the brake pedal sensor, they are able to fingerprint the driver with high accuracy [6]. Nishiwaki et al. use live CAN bus data in combination with Hidden Markov Models and Gaussian Mixture Model to understand the level of distraction of the driver, together with its driving patterns and habits [7]. One can imagine how this information could be used in an unintended way; for example by insurance companies [1], [2] or for legal disputes [4], [5].

In all these prior works, researchers had to *manually* decode the payloads of specific frames of the car to extract the data they cared about – in case of Enev et al. the brake pedal. This is a very tedious, error-prone, and difficult to scale approach – mainly because it requires access to the specific vehicle under test. A one-fits-all solution is currently not possible since all in-vehicle networks only standardise the physical layer and link layer. The higher layers, i.e., the payload, differs for each maker – sometimes even differs for each model by the same maker. Manufacturers treat the protocols, i.e. how data is encoded, how often it is expected to be sent, what actions the receiver has to take, all as company secrets. It is simply not possible to get documentation from OEMs – often not even under NDA. This also explains the high prices that vendors of handheld diagnostic devices ask since they have an enormous amount of work to do to reverse-engineer the protocols. As a consequence, automotive security and privacy researchers have to perform a significant amount of reverse engineering before conducting the actual research. This makes it hard for researchers to get into the topic, and unlikely for the automotive industry that others discover security and privacy flaws. One of the most prominent examples of how such "obscurity" has been abused by makers is the Volkswagen emissions scandal [17].

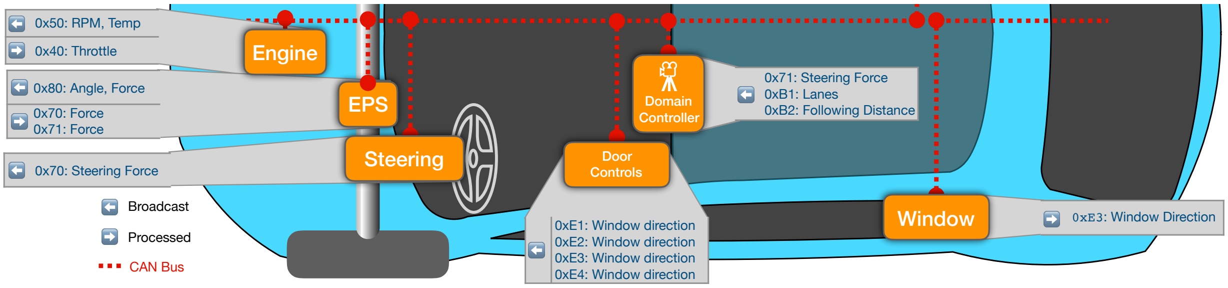


Figure 1: Exemplary CAN network: ECUs are interconnected to exchange data. Frames are broadcasted to everybody else, but only selected frames are processed by the receivers.

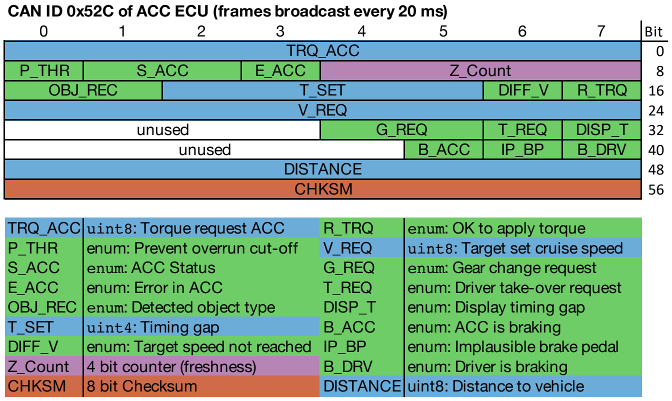
Reverse-engineering in-vehicle networks faces several challenges because:

1) it is almost impossible to understand who's the sender or receiver of a frame by just tapping into the bus,
2) there is no visible information flow: ECUs just constantly broadcast their own state and react on somebody else's state,
3) each frame typically transmits several different signals, laid out in custom binary encoding that is not documented and is different for each ECU, often even different for each frame ID belonging to the same ECU. Figure 2 depicts a CAN frame with related signals. Any receiving ECU must know the exact payload layout in order to make sense of the frame,
4) the software running on ECUs and processing such frames is proprietary and the program memory typically protected against read-out,
5) even standardised access such as *On-Board Diagnostics* (OBD) is typically implemented in addition to proprietary protocols and is only concerned with emissions-related ephemeral sensor readings. It does neither include stored information in ECUs nor privacy-related sensor values. It is not required for electric cars, hence most electric cars do not even support OBD.

A. Example of in-vehicle information flow

When the driver tells the cruise control to set a target speed, a somewhat complex process takes place in the background: Figure 3 shows an example of how a Volkswagen Passat transmits and processes the vehicle speed: The ABS-ESP[2] ECU uses rotational sensors to measure each wheel's RPM. This information is used by the same ABS-ESP ECU to calculate the average RPM of all wheels to get a value proportional to the vehicle's speed. However, it does not know the wheel's circumference, which is stored in the instrument cluster. Hence, the instrument cluster multiplies the received RPM value by the circumference of the wheels in order to get the speed in *kilometres* or *miles* per hour. However, in most countries in the world, the UNECE[3] requires the speedometer to never show less than the actual driving speed [18]. To comply, practically all manufacturers add some 10% to the displayed speed – even if it is digital. This poses a challenge for functions like cruise control that should seem consistent to the driver and have to work with the physically *'wrong'* speed that is shown to the driver. As a result, the engine control module, in our example, uses the *'wrong'* speed broadcasted by the instrument cluster to derive the set-point of the cruise control [19].

This results in complex dependency chains in which similar values are continuously and asynchronously transmitted by different ECUs, often with diverse scale and encoding.



CAN ID 0x52C of ACC ECU (frames broadcast every 20 ms)

0	1	2	3	4	5	6	7	Bit
TRQ_ACC								0
P_THR	S_ACC		E_ACC	Z_Count				8
OBJ_REC		T_SET				DIFF_V	R_TRQ	16
V_REQ								24
unused				G_REQ		T_REQ	DISP_T	32
unused					B_ACC	IP_BP	B_DRV	40
DISTANCE								48
CHKSM								56

TRQ_ACC	uint8: Torque request ACC	R_TRQ	enum: OK to apply torque
P_THR	enum: Prevent overrun cut-off	V_REQ	uint8: Target set cruise speed
S_ACC	enum: ACC Status	G_REQ	enum: Gear change request
E_ACC	enum: Error in ACC	T_REQ	enum: Driver take-over request
OBJ_REC	enum: Detected object type	DISP_T	enum: Display timing gap
T_SET	uint4: Timing gap	B_ACC	enum: ACC is braking
DIFF_V	enum: Target speed not reached	IP_BP	enum: Implausible brake pedal
Z_Count	4 bit counter (freshness)	B_DRV	enum: Driver is braking
CHKSM	8 bit Checksum	DISTANCE	uint8: Distance to vehicle

Figure 2: An example CAN ID and its encoding. Taken from the documentation of a BOSCH Engine ECU that broadcasts 11 different CAN IDs and receives 16 CAN IDs [20].

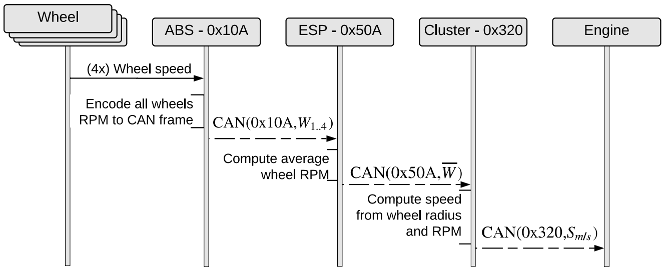

Wheel
ABS - 0x10A
ESP - 0x50A
Cluster - 0x320
Engine
(4x) Wheel speed
Encode all wheels RPM to CAN frame
CAN(0x10A,$W_{1..4}$)
Compute average wheel RPM
CAN(0x50A,$\overline{W}$)
Compute speed from wheel radius and RPM
CAN(0x320,$S_{m/s}$)

Figure 3: ECUs dependencies in a Volkswagen Passat.

[2] Anti Lock Brakes / Electronic Stability Program

[3] United Nations Economic Commission for Europe.

B. Assumptions

We assume to have read-only access to in-vehicle network traffic. We do not need physical access to the car. Hence, we only need a log file of network packets. In case the network access is possible through a vulnerability in an ECU, we assume the attacker cannot or does not inject packets and can only learn from passively sniffing the traffic. The attacker does not know the car model, condition, and driving situation. It is also assumed that she does not have prior knowledge of the layout of the exchanged data. The latter is the worst case if no documentation is available. If an attacker can still identify specific fields (e.g., the brake pedal or steering wheel angle), she can then apply the aforementioned techniques to identify the driver or its geographical position by correlating the steering wheel profile to a map.

Even though our methods work independent of the used protocol and physical layer, in the remainder of this work we focus on CAN as it is the most widespread in-vehicle network technology. We also focused on electric cars as they do not feature OBD as additional help and thereby constitute a worst-case reversing effort.

IV. Underlying Concept

Even though we start from an unknown structure in sniffed CAN traffic, we can make the following observations:

1) One ECU might have different functionalities, but CAN frames with the same sender ID will employ the same logical format. This means that, while the content in terms of bits changes over time, the semantics will not change as they are hard-coded in the software of the ECU that processes those frames.
2) Many sensor values measure the same physical unit (e.g. temperature), just at different places in the car.
3) CAN signals, i.e. interpretations of the unstructured bits, are related to each other by physical laws. Hence, their relations can be described by mathematical formulas.

We leverage these observations to automatically derive signals *without having access* to the car. As a proof of concept, we developed a set of Python-based tools which facilitate the decoding, analysis, and application of learning algorithms to CAN bus traffic. Although in this paper we concentrate on the CAN bus, we expect our methodology to generalise to other physical systems (e.g., industrial control system). The high-level approach works as follows:

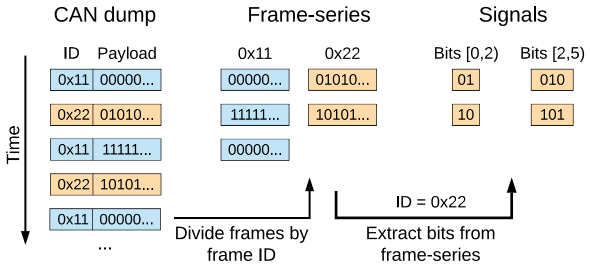


Figure 4: CAN dump to frame-series to signal logic.

1. Frame-series extraction. The first step is to subdivide the frames by origin. This is done by grouping the frames by their sender ID: this allows to generate a frame-series for each sender ID (see Figure 4).

2. Signals segmentation. The second step is applied to each frame-series independently. The purpose is to segment each frame into the various signals it transports. This means we group consecutive bits which likely belong to the same signal together, and create a different signal whenever consecutive bits show very different statistical behaviours.

3. Plausibility analysis. To infer whether a signal contains a plausible value or not, we defined a set of metrics and rules to be able to distinguish, identify, and categorise signals. This is feasible as different signals show clearly distinct behaviour. In Figure 5, we show the four behaviours we identified and wish to distinguish.

A **Continuous** (`uint`) signals like speed, temperature, etc. These values are generally generated by sensors, or are derived from them.
B **Pseudo-random** (`rand`) values, such as checksums, which are generally contained in frames carrying safety sensitive information (e.g. steering wheel angle).
C **Enumeration** (`enum`) signals like gear, parking-mode, break pressed, door open, etc. These fields are generally used to encode a certain state of the car.
D **Cyclic** (`cyclic`) signals that iterate continuously like clocks or counters; they do not necessarily carry information per se, but can be useful to derive more complex patterns.

4. Signal normalisation. A signal can have an offset and a scale that is tailored to the transmitted value. For example, temperatures often use a scale of 0.5 and an offset of -40, i.e., $Temp[°C] = Signal \cdot 0.5 - 40$, which gives a value range from $-40°C$ to $+87.5°C$ (assuming a 7-bits encoding). To account for that and work on similarly scaled signals, we apply min/max normalisation to each signal.

5. Correlation analysis. Values transmitted by ECUs are often correlated to each other by physical and mathematical properties. For example, the acceleration is proportional to the throttle pedal position and the engine temperature changes accordingly to the engine load. This reasoning holds for both values derived directly from sensors (e.g. speed) and values derived from other physical values (e.g. odometer). Figure 6 shows an example of this correlation: it is easy to see how different values transmitted by different ECU can still be correlated to each other.

V. Automated Reversing Methodology

Here we detail the most relevant steps, which are: *signal segmentation*, *plausibility analysis*, and *correlation analysis*.

A. Signal segmentation

As we have no knowledge about how signals have been encoded, we first have to decide how to group and interpret the bits of each frame. Given the limited amount of payload bytes in a frame, signals are often encoded with the minimal

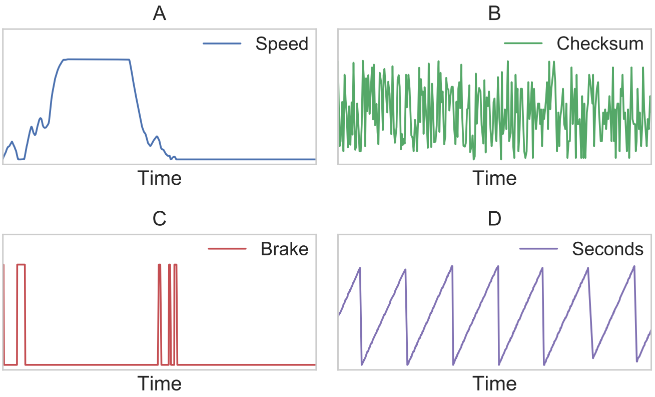


Figure 5: Raw values for the different signal categories identified in the CAN bus.

number of bits they require. To this end, we designed a greedy algorithm which, starting from bit 0, incrementally adds bits to the current guessed signal. The guessed signal is then evaluated via *plausibility analysis* (V-B): based on the result the algorithm decides to (i) keep the guessed signal, add one more bit and iterate, or (ii) commit the guessed signal, update the start bit, and iterate. The algorithm stops when all the bits of the frame have been processed.

The intuition behind the algorithm design is this: First, let us denote a signal with $f(z, i, j)$, where z is the frame ID to which the signal belongs to, i is the signal starting bit, and j is its length in bits. In the following, let us denote with $P(\cdot)$ the *plausibility function* that, given a signal $f(z, i, j)$ in input returns its inferred type $Y \in [\texttt{uint, enum, rand, cyclic}]$ if plausible, or NaN if not. Given these definitions, the algorithm works as follow: Given a signal s of type y,

$$s = f(z, i, j), \qquad y = P(s)$$

and a signal s' of type y' such that,

$$s' = f(z, i, j + 1), \qquad y' = P(s')$$

if $Y == Y'$ then we assume the bits composing s are also part of s'. Hence the algorithm substitute $s = s', j = j + 1$ and iterate with $j = j + 1$. On the opposite, if $Y \neq Y'$ with $Y \neq NaN$, then we assume the signal s is invalid if more bits are added to it, and is therefore complete. Hence the algorithm can split the payload at j, committing the signal $s = f(z, i, j)$ of type y, and start anew with a signal $s' = f(z, i + j, 1)$ with type $y' = NaN$. We found this approach to be efficient in terms of time and memory as its complexity is linear to the number of bits composing the payload. The pseudo-code is described in 1; in the algorithm, *candump* correspond to a complete, unprocessed, CAN dump.

B. Plausibility analysis

Given an arbitrary CAN signal, the purpose of the plausibility analysis is to return its inferred type Y if plausible or NaN if not. This information is used by the segmentation algorithm to decide when to split the signal. To correctly categorise the types identified in Figure 5, which are `uint`, `enum`, `rand`, and `cyclic`, we devised a set of metrics able to capture their different statistical behaviour.

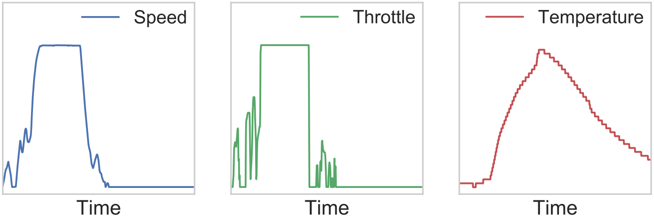


Figure 6: Correlation between CAN signals.

Algorithm 1 `signalsSegmentation`(*candump*)

```
$T \leftarrow$ frameSeriesExtraction($candump$)
for each $can_{ID} \in T$ do
    $F \leftarrow$ getPayloadsBits($T[can_{ID}]$)
    $i,\ j \leftarrow 0,\ 1$
    while True do
        $P_1 \leftarrow$ plausibilityAnalysis($F[i:j]$)
        $P_2 \leftarrow$ plausibilityAnalysis($F[i:j+1]$)
        if valid($P_1$) and ($P_1$ != $P_2$) then
            segmentSignal($i, j, P_1$)
            $i \leftarrow j$
        end if
        if $j+1$ == numBits($F$) then
            if valid($P_1$) then
                segmentSignal($i, j, P_1$)
            end if
            break
        end if
        $j \leftarrow j+1$
    end while
end for
```

Autocorrelation. As a first step, we want to understand how "noisy" a signal is. A noisy signal indicates either a wrong selection of bits or that a truly random value was transmitted. The concept of *autocorrelation* comes in handy and it approximates to $+1$ when elements in a series are correlated (or -1 if they are inversely correlated), and to 0 when they are highly uncorrelated. The autocorrelation function measures the correlation between the i-th value and the $(i-h)$-th value of a time-series, where h is the lag. As can be seen in figure 7, the autocorrelation clearly distinguishes between the different signals.

Hamming distance distribution. We call the second metric we devised *Hamming distance distribution* (`HAMD`). The metric has the role of capturing bit endianness and encoding size. The metric is computed by measuring how individual bits change whenever the signal under test changes. This is computed by keeping a counter for each bit, which is incremented whenever the bit changes. The final distribution is obtained by dividing all counters by the number of times the entire signal changed: this results in a value between 0 and 1, where 1 means that the bit always changes whenever the signal changes, while a

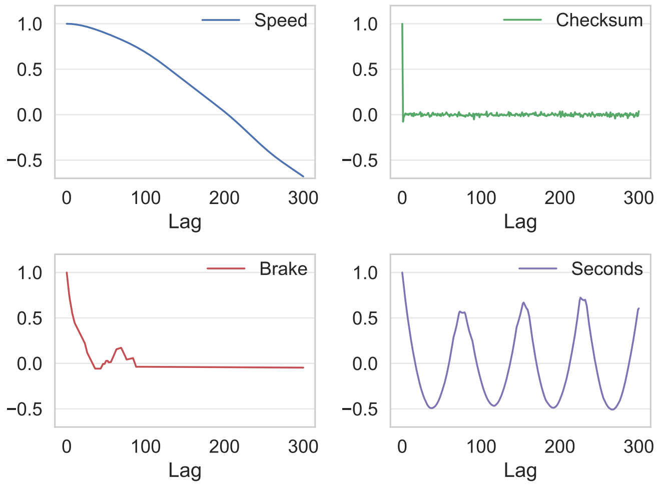


Figure 7: Autocorrelation function.

0 means the bit never changes. Figure 8 shows that little-endian values have an almost logarithmic distribution with greater HAMD to the right. This distribution is a good indication of endianness and integer boundaries. A complete example can be seen in Figure 9, where the HAMD of multiple CAN frames is displayed. In the heatmap, each row corresponds to a single CAN ID and each column to a payload bit. The cell intensity represents the HAMD for the corresponding bit: a value close to 1 implies a very active bit, and a value close to 0 a seldomly changing bit. The heatmap gives a visual intuition of how the signals are encoded, with incremental intensities denoting little-endian encoded values and sharp intensity changes denoting signals boundaries.

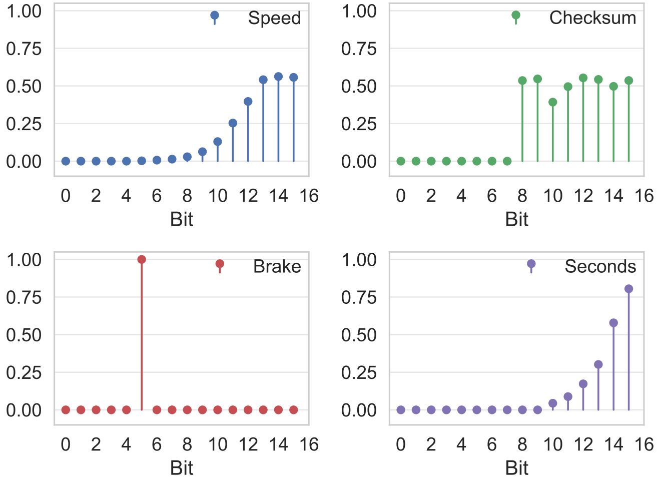


Figure 8: Hamming distance distribution.

Post-hamming distance distribution. To further validate the MSB of uint types, we devised an additional metric which measures the HAMD of the i^{th} bit in respect to the $(i+1)^{th}$. We use this metric as a sanity check: if a bit in position i changes, but the bit in position $i+1$ doesn't, then it is unlikely they belong to the same little-endian encoded integer.

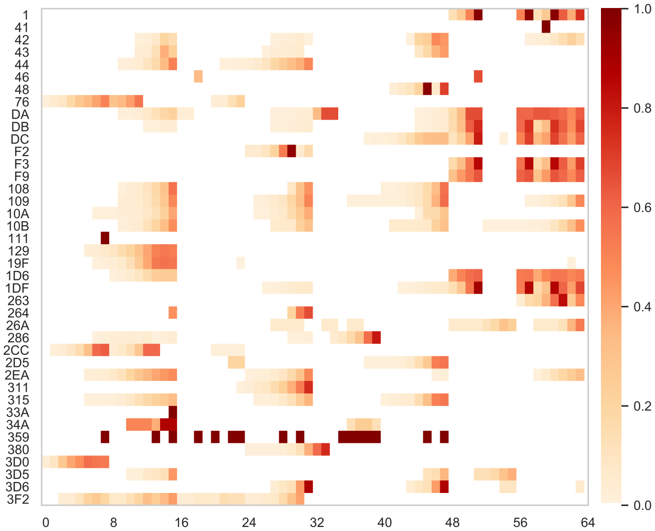


Figure 9: HAMD for some frame IDs of the Smart ED.

Removal of signals error. A common practise in CAN is to encode a signal which contains an unknown, faulty or unavailable measurement with its maximum value (all its bits are 1). This is generally due to booting processes (e.g., the cluster might boot before the ABS – hence it cannot show the speed), and sensor/physical process imprecisions (e.g., voltage fluctuations on ignition). This results in certain signals being composed of only 1s for brief periods of time. For example, is some cars the speed (uint 16) is encoded as 0xFFFF for the first few seconds after ignition, then it moves back to its real value (e.g., 0). To handle such cases, we remove the samples where all bits transition to 1 at the same time.

C. Correlation Analysis

Sensors capture the physical environment through which the car moves. This environment adheres to the laws of physics. This dependency can also be found in the captured sensor data. For example, physics dictate that the first derivative of displacement x w.r.t. time is *velocity*, the second is *acceleration*:

$$\vec{v} = \frac{\partial x}{\partial t}, \quad \vec{a} = \frac{\partial v}{\partial t}$$

A car has all those sensors to measure $x, \vec{v}$ and $\vec{a}$: The displacement, or distance travelled, is the *odometer*, the speed is calculated and displayed in the instrument cluster, and the acceleration is measured for stability control and impact detection. This insight allows us to correlate signals to each other based on their mathematical relation. For example, the distance travelled over time, which can be derived from the speed, must highly correlate to the odometer. Both signals are present in the CAN and, even though the encoding, precision, scale and initial value might be unknown, the normalised time-series would still feature a high correlation.

We use this observation to find relations between discovered signals, which helps labelling new signals if a relative signal is already labeled. We use an exhaustive approach to find relations by brute-forcing mathematical relations until a high correlation is found. To this end, each signal derived from the segmentation is put into a formula and cross-correlated with all other signals. The formulas are created by combining all possible combinations in an abstract syntax tree (AST) whose nodes represent one of following operations: addition ($+$), subtraction ($-$), multiplication ($\times$), division ($\div$), derivative (∂), and integral ($\int$). Each operator node can have one or two operands, namely other signals. All combinations of signals are brute-forced with all combinations of operations up to a depth of two in the AST. Each AST is applied to an entire signal and results in a new, derived signal. This newly derived signal is cross-correlated against existing signals to discover a physical relation. The most correlated signals, together with their mathematical relationship, are presented to the user which can verify the relationship and label the signals.

We extend the approach by finding and visualising all signals in a graph: The rationale behind this graph is to cluster highly related signals together (e.g., battery voltages), to split unrelated signals into different clusters (e.g. counters, checksums) and connect the various clusters via weighted edges to represent their relations. Such graph dramatically simplifies the process of labelling all signals belonging to the same the cluster and provides an easy way to visualise how the signals relate and depend on each other. To this end, we use the *Pearson correlation coefficient* ϕ computed between each pair of signals. The coefficient ϕ returns a value between $[-1, +1]$, where $0.5 \leq |\phi|$ implies strong direct (or inverse) correlation, $0.3 \leq |\phi| \leq 0.5$ implies medium correlation, and $|\phi| \leq 0.3$ implies little correlation. We use the correlation coefficient ϕ, with $|\phi| \geq 0.8$, to derive a graph where the signals are the nodes and the edges are determined by the coefficient ϕ between them.

VI. Evaluation and Findings

We evaluated our algorithms on four different cars:

- 2014 Smart Electric Drive (MY 2014, electric)
- 2013 Ford Fiesta (MY 2013, petrol)
- 2012 Renault Twizy (MY 2012, electric)
- 2018 Renault Zoe (MY 2017, electric)

We designed electronics for a logging box that connects to the standardised SAE J1962 connector present in almost every car. The box logs CAN traffic on the most common pins in the hope that they are connected to at least one power train CAN. It also regularly logs the GPS position every 5s. We conducted the experiments with volunteers who gave their consent that we log all CAN traffic, including their GPS position. The subjects were told to use the car normally for their day-to-day business without the need for a specific pattern. The concept of this study was approved by the Ethical Review Board (ERB) of our institute.

Passively obtained raw data: For each car, we logged the raw CAN traffic and GPS position for several days. We only use the GPS position to verify our results. The GPS position and derived data (like speed) were *not used* to train or tailor any of our algorithms. We subdivided all logs into "trips" by splitting them whenever no CAN activity was detected for more than 60 seconds, whereby we assume the car is switched off. This results in multiple trips per car, with some of them being few minutes and few thousands of frames long, to some of them being few hours and millions of frames long.

Ground Truth for verification: To get a ground truth of CAN encodings for each vehicle, we combined leaked documents, OBD analysis, fuzzing, online resources, and manual reversing with visual feedback from the car to obtain as much information as possible. Luckily, the different sources of data did not contradict each other, i.e., the leaked documents and forum snippets concur with our manual reverse engineering. In Table II, we summarise the dataset used for the evaluation, which is composed of the ten longest trips for each vehicle.

A. Segmentation

The purpose of the segmentation is to decode a payload by cutting it into signals. Each signal is then assigned one of the four derived types. We presented our approach in V-A and V-B: the algorithm obtains good results, and is able to correctly separate `uint`, `enum`, `rand`, and `cyclic`. Figure 10 depicts the segmentation for frame `0xDC` of the Smart ED. The Figure shows the signals, and how the algorithm correctly detects the checksum encoded in bits $[56 : 64)$, the counter used for freshness at bits $[48 : 52)$, and the various enums/uint composing the frame. The algorithm also detects peculiar encodings: the frame encodes the engine RPM as unsigned short in the bits $[34 : 48)$ and uses the flag in position $[33]$ to denote the sign. So unlike a *two's complement* representation, the value 0 can have positive *and* negative sign.

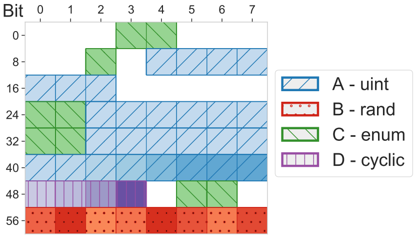


Figure 10: Segmentation of frame `0xDC` for Smart ED.

Figure 11 illustrates the segmentation for the whole powertrain CAN of the Smart ED compared to its ground truth. The heatmap demonstrates that the segmentation reflects the ground truth with good precision, and that (a) `enum` variables are often distinguishable by their clear 1 single-bit `HAMD` value (e.g. the brake pedal for frame 0x111), (b) `uints` have an incremental `HAMD`, (c) `rand` are composed by a sequence of bits with `HAMD` values close to 0.5, (d) `cyclic` signals

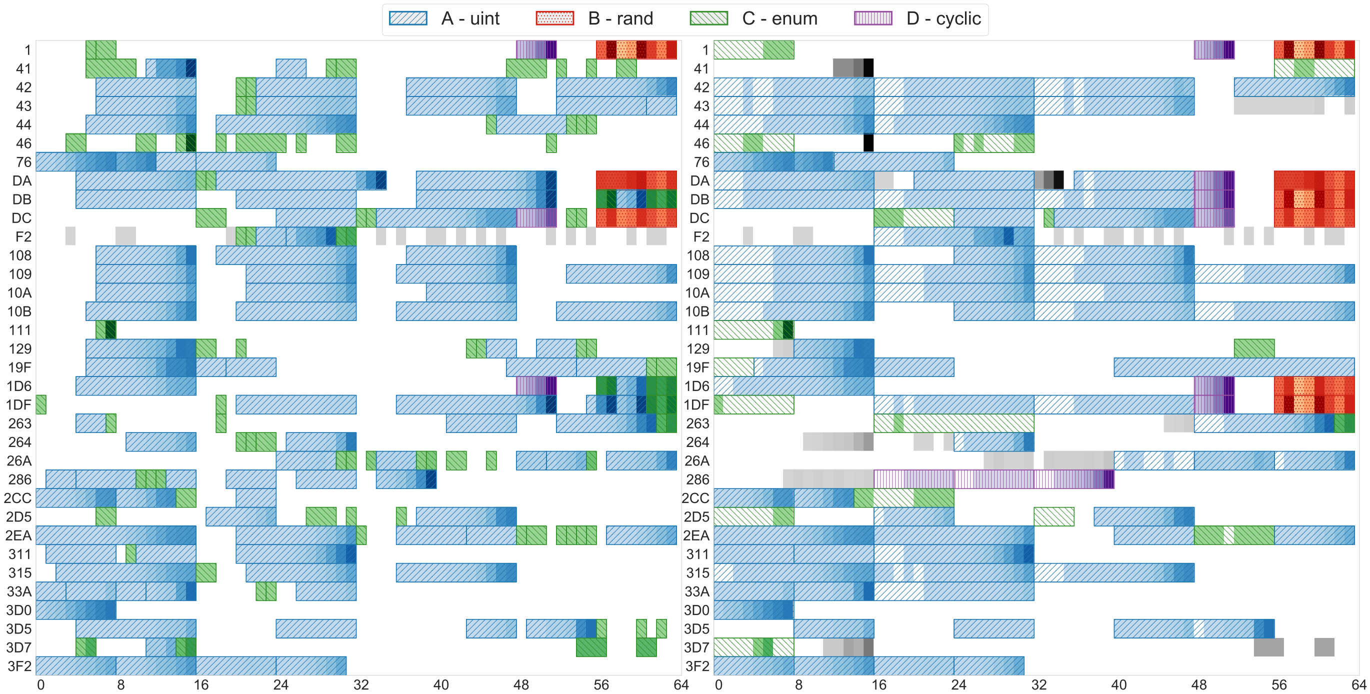


Figure 11: Segmentation of the power-train CAN of the Smart ED (on the left) compared to the ground truth (on the right).

have an almost perfect geometric HAMD.[4] However, a few issues arise as a consequence of some of our segmentation decisions. First, we decided to label a signal as uint only if its post-HAMD is higher than 0 for all its bits. We do this to avoid mistaking a enum as uint (e.g., sign-bit in frame 0xDC). This sometimes leads to a situation where the most significant bit of a uint is misclassified as enum due to the fact it changes only once or twice throughout the trip. Similarly, rand checksums are sometimes not detected as such. This is due to some frames being quite regular (i.e., few unique values), which results in a checksum with few unique values. Both issues would be solved with time i.e., by simply waiting until enough CAN frames are present in the dataset. Additionally, the algorithm won't label a cyclic unless the cycle is repeated multiple times. Signals that don't repeat during the duration of a single trip will always be labelled as uint. For example, frame 0x286 encodes the hour: such signal will never be labelled as cyclic unless it doesn't repeat for at least two times (which would require a 48 hours long trip).

Table II summarises the results. In the Table,

- **Total time and frames** represent the cumulative length in hours and in frames of the trips, respectively.
- **Known signals** are those we manually reverse-engineered and for which we know the bit boundaries and type.
- **Segmented signals** represent the signals returned by the segmentation algorithm. Note that only a subset of those signals is depicted in Figure 11.
- **Segmentation time** is the time required by our algorithm to segment all the frames, on a single core Intel Core i7 3.5GHz, 16GB DRAM, 1TB SSD.
- **Matched signals** is the percentage of segmented signals which are *perfectly* (boundaries and type) matched to known signals.
- **Mean bit-error** is the percentage of bits that are misclassified, i.e., assigned to a different signal or type.

Vehicle	Total time	Total frames	Known signals	Segmented signals	Segmentation time(s)	Matched signals	Mean bit-error
Smart ED	1h	6.8M	107	453	20	0.81	0.093
Renault Zoe	2h	4.3M	40	416	6	0.71	0.076
Ford Fiesta	3.5h	7.5M	81	289	21	0.70	0.105
Renault Twizy	17h	10M	46	161	30	0.89	0.057

Table II: Evaluation of the segmentation algorithm.

[4] The real power-train has more IDs (74) than the depicted ones. However, since we had no ground truth for them, we removed them from the plots. Also, many signals the segmentation discover, especially enums, are not present in the ground truth. Similarly, constant bits (e.g., leading zeros in most uints) are not labelled unless they change once.

B. Comparison to READ

We report a comparison with the state-of-the-art CAN reverse engineering tool, which is the READ algorithm proposed by Marchetti and Stabili (further discussed in Section IX) [21]. Before discussing the results, we would like to note that the segmentation performed by READ is very limited compared to ours. For instance, READ only aims at finding `rand` and `counters` signals, where `counter` are only those signals which perfectly increase by one for every frame (i.e., checksums counters). READ doesn't distinguish between `uint`, `cyclic`, and `enum`, and labels all remaining signals as `PHY`. To be comparable, we adapted our algorithm and ground truth to consider the labels `uint`, non-trivial `cyclic` and `enum` as `PHY`. This allows us to evaluate how our algorithm fare in detecting `PHY` boundaries and `counters`/`rand` if compared to READ. Table III summarise the results: the metrics are the same as for Table II. It can be seen how both algorithms perform well in detecting the boundaries, and basic `rand`/`counters`, with our algorithm marginally better than READ in some cases; we assume this is due to the additional sanity checks we perform to remove frames likely to be containing error signals. Benchmark-wise, the two algorithms requires a similar amount of memory and time to segment the same number of frames.

C. Correlation

The purpose of this phase is to automatically find mathematical relations between signals. We found our approach to be efficient in narrowing down the labelling choices for specific physical signals (e.g., speed) to a couple of possibilities. For example, in our experiments, the integral operator $\int$ applied to the speed was always highly correlated with the odometer (Figure 12) – which follows the laws of physics that dictate that distance travelled is the integral of speed. Furthermore, the throttle pedal position highly correlates with the first derivative of the speed – acceleration. By just computing the AST between all signals with the $\int$ and $\frac{d}{dt}$ operators, and by taking the triplets of most correlated signals, one can easily find the speed-throttle-odometer triplet. Other relations showed a high correlation; for example, the boolean state of the brake pedal highly correlates with negative acceleration values.

After the initial labelling, we calculated the graph between all signals. Figure 13 illustrates a simplified graph for the Smart ED: it can be seen how the graph captures relationships between signals. For example, signals carrying measurements (e.g., voltages) regarding the `traction-battery` are clustered together in a fully-connected sub-graph. Signals related to the `12V-battery` are also clustered together in a fully-connected sub-graph, but are also completely decoupled from the `traction-battery`. Furthermore, by considering fully connected sub-graphs as single nodes and examining how they relate, we can infer their content. For example, the *speed* signal (which belongs to the same cluster as the engine RPM, throttle, etc.) is strongly related to two other clusters: the `traction-battery` cluster and the `torque` cluster. From physics, we know this to be true as a change in torque applied to the wheels directly affects positive and negative acceleration. Acceleration also draws current from the traction battery, thereby decreasing its voltages due to internal resistance. Hence, one can understand what signals a cluster contains by examining such relations. Also, by examining the internals of each cluster, one can derive the internal signals. For example, in an electric car, each cell or cell-pack voltage is transmitted in a different signal: each of these voltages should be very similar over time. This process can be repeated, together with the AST brute-forcing, starting from signals that are relatively simple to infer (e.g., speed, odometer, range), to more complex ones (e.g., recuperation, driving efficiency).

Vehicle	Matched signals	READ seg. signals	READ matc. signals
Smart ED	0.91	375	0.88
Renault Zoe	1.0	144	1.0
Ford Fiesta	0.96	172	0.93
Renault Twizy	1.0	123	1.0

Table III: Comparison against READ algorithm.

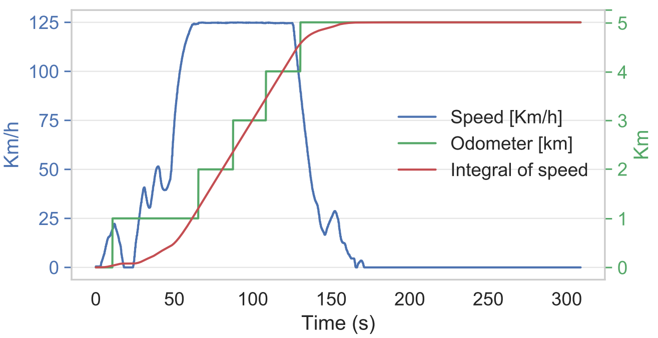


Figure 12: Correlation between odometer and speed.

D. Findings

In Table IV, we report the most privacy relevant signals we extracted. The signals are from the Renault Zoe and are known to correspond to the ground truth.[5] The column *upload* states whether the signal is uploaded to the cloud by default: the details on how and why we obtained such knowledge are explained in the next Section. The complete table with more signals and can be found in the Appendix.

VII. Telematic Control Unit

Makers are very obscure about what the *Telematic Control Unit* is actually doing: there is no documentation about what functionalities are implemented, how many are (or might be) actually used, and when, why, and which data the TCU sends to the cloud. Despite the TCU's obvious purpose of collecting data, little is known about what precisely is collected and

[5] We report the signals for the Renault Zoe instead of the Smart ED as it is the only car for which we reverse-engineered the TCU.

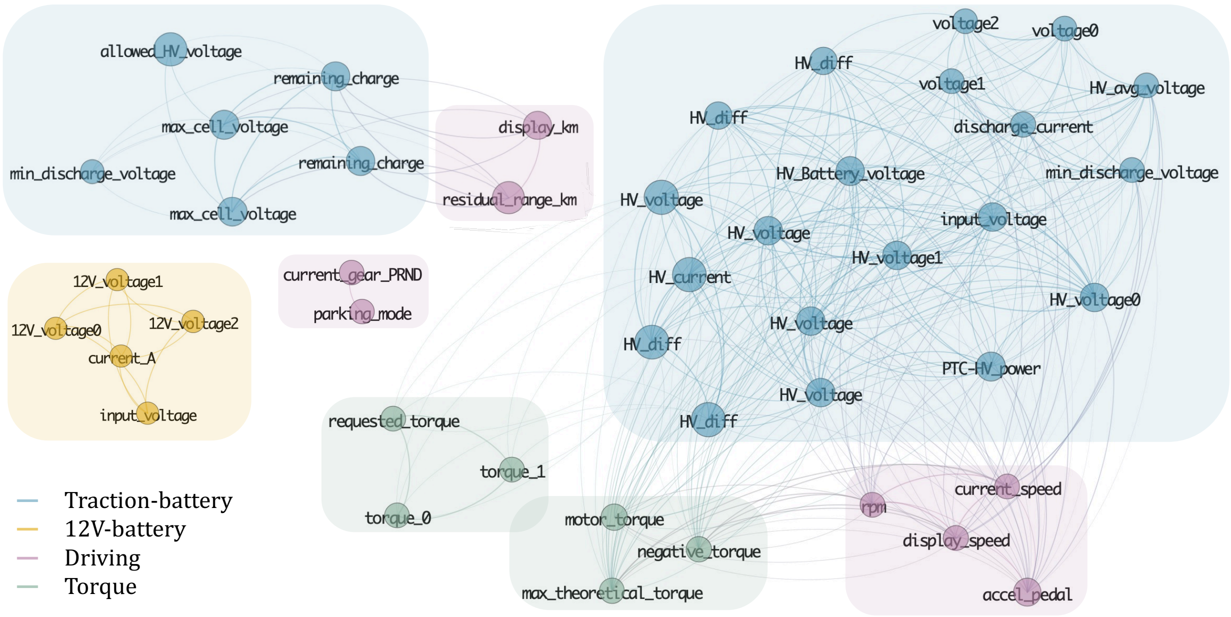


Figure 13: Simplified correlation graph for a subset of signals for the Smart ED.

Signal	Uploaded	Description
Trip distance (km)	Yes	
Trip average consumption	Yes	
Trip average speed	Yes	
Remaining range	Yes	
Odometer	Yes	
Charging cable plugged	Yes	If charging cable is plugged in
Battery Health	Yes	
Battery serial number	Yes	
Ignition on/off	Yes	
Age of vehicle	Yes	Seconds from first start
VIN	Yes	
Brake pedal pressed	No	
Throttle pedal	No	
Speed in cluster	No	Display speed (ca. +5km/h)
Real speed	No	Real speed in km/h
Longitudinal accel.	No	
Transversal accel.	No	
Steering angle	No	

Table IV: Private signals that can be passively derived.

with which granularity. For example, from the two articles by TESLA [4] and BMW [5] it can be seen how both OEMs possess very detailed driving logs. Manufactures also allow users to remotely control basic car functionalities (e.g., HVAC) [13], [14], [16]. However, it is unclear what else can be controlled via the TCU (and how). For example, the most widely sold electric car manufacturer Renault-Nissan states in their battery lease contract that they will *"prevent further recharging of the Battery"* if the user doesn't fulfil the contract clauses [22]. To investigate such claims, we took the most widely sold electric car in Europe, a Renault Zoe from 2015, which shares its remote capabilities with the most widely sold electric car worldwide: the Nissan Leaf.[6] Both models are equipped with a TCU that allows the accompanying smartphone app to preheat the car or gather battery state information remotely. The Renault Zoe features a connected multimedia system. The software, called R-LINK, is embedded into the multimedia touch-panel and provides commodity features such as maps with live traffic (via TomTom), Bluetooth pairing, voice recognition, eco-driving style analysis, Renault app-store access, and remote control of basic car features via the smartphone app [24]. R-LINK connectivity is provided by the TCU, a physically separate component with an embedded SIM card. The TCU can connect to the Internet, send/receive SMSs, and make/receive calls. The TCU is directly connected to the multimedia panel via UART and has access to the car's main CAN bus.

A. CAN traffic analysis

To understand which data is acquired and exchanged between ECUs, it is vital to understand *who* sends a message and what its meaning is. By passively listening on the bus, one can only see the frame ID and up to 64 bits of payload. To this end, we designed a *CAN Interceptor* that uses two CAN transceiver chips on each side and forwards the received packet instantly to the opposite CAN transceiver. The interceptor forwards the original CAN traffic bidirectionally and is therefore transparent for all ECUs connected to the bus. Given that, we physically located the TCU and used the *CAN Interceptor*, which we placed between the TCU and the rest of the CAN, to examine the communications. While controlling the car via Renault's App, we could see which commands are sent to the CAN network by the TCU and which sensor values are actively queried by the TCU before transmitting them to the cloud. From the experiments we understood that: (1) the TCU is generally passive, but it periodically broadcast a single

[6] Since 2019, Tesla model 3 became the most sold electric car [23].

frame to inform the bus of its status, (2) upon certain events (i.e. ignition ON/OFF, charging ON/OFF or at least every 15 minutes), the TCU initiates an OBD session with the battery controller, from which it retrieves various values (cell voltages, temperatures, etc.), and (3) if a remote command is received, the TCU sends a CAN frame to the bus (e.g., to tell the climate control to start heating the car).

B. *GSM traffic analysis*

We also examined the communication between the TCU and the nearest mobile base station. For that, we used openBSC, an open source implementation of the GSM architecture developed by the osmocom community. If paired with a GSM capable hardware (e.g. sysmocom sysmoBTS [25]), it allows to completely simulate a GSM network [26]. We focused on the GSM standard because (1) devices connect to the base tower with the strongest signal belonging to the operator of the SIM card, and (2) the GSM standard doesn't provide means to authenticate the remote operator. This implies it is possible to force a GSM device to connect to a fake base station by simply simulating the correct operator while having a stronger signal than the original cell tower (which is not an issue if the BTS is near the car) [27].[7] Via OpenBSC, we forced the TCU to connect to our base station instead of the real network operator. The attack succeeded because the TCU preferred to connect to the GSM network instead of using the more secure 3G/4G. We assume this decision has been taken considering the higher presence of 2G cells in rural areas. Nevertheless, it is possible to force the downgrade by testing the car in a shielded environment.

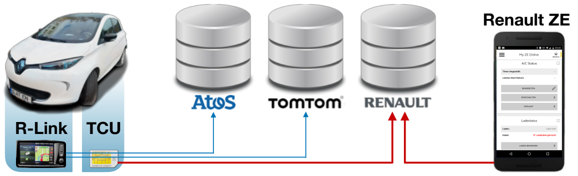


Figure 14: Flows between R-Link, TCU, and Renault ZE.

From the GSM logs, we identified three traffic flows: (1) an HTTPS connection to ATOS, an IT service provider, (2) various HTTP/S requests to TomTom, and (3) sporadic HTTPS connections to a Renault-owned domain. Between these three, only the latter was correlated with the activity of the TCU on the CAN bus, which suggested that the TCU was acquiring data before sending it to the cloud. We verified that the requests were performed even when the multimedia panel was disconnected: this implied they originated from the TCU itself.

From the packet dumps, we saw that both TCU and remote server used X.509 certificates to create an SSLv2 channel. Even though SSLv2 is outdated, it prevented us from inferring which data was being transferred from just the TCP/SSL dumps. Nevertheless, by checking the activity periods, we understood that the commands from the mobile app were not coming over HTTPS. Instead, they were sent using ordinary SMSes.

C. *UART*

The R-LINK infotainment system and the TCU communicate over UART (serial bus) using standard AT modem commands. The standard AT commands (e.g. `AT+CCLK?` to get the clock) are extended by manufacturer-specific ones, accessible via `AT+BREWAPP=(cmd)`[8]. Out of all commands, the most useful was `AT+BREWAPP=$DBG,1` which enables debug mode. When in debug mode, the TCU prints debug messages, such as:

- Which event triggered the data transmission.
- Path and length of the HTTPS data being transmitted.
- Firmware version and producer of the TCU.
- Some of the data encoded in the messages.

By analysing the logs, we realised that packets are encapsulated using the *ACP-245* protocol, a public standard that describes how communication between a TCU and a remote server should be handled [28]. Since we couldn't find any open source implementation, we wrote a basic parser ourself. We obtained clear-text messages by making use of another AT command: AT+BREWAPP=$SET_URL, which overwrites the manufacturer's default server. This URL also includes the protocol, so by putting `HTTP` instead of `HTTPS` one can disable SSL completely. Thanks to this command, the car was sending the first payload to the server without authenticating/encrypting the data (although the server didn't reply). By joining the logged data, HTTP data, and our ACP-245 parser, we discovered that the car always embeds the GPS coordinates and the ACP-245 vehicle descriptor element, which contains the VIN number, DCM ID and version, IMEI, SIM ID and Battery ID, in every payload it sends to the manufacturer.

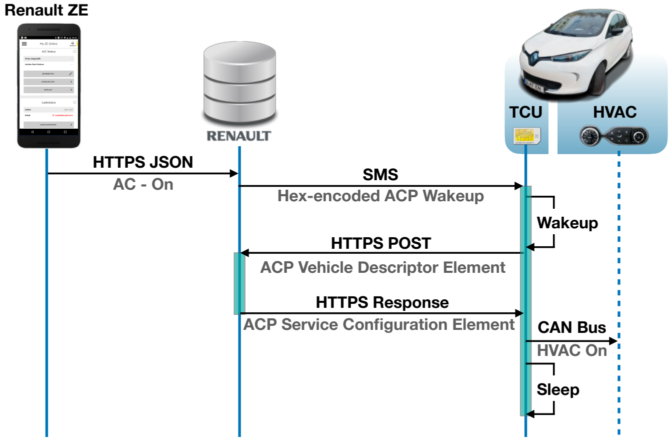


Figure 15: Remote command execution flow.

[7] It is sufficient to know the *Mobile Country Code* and *Mobile Network Code* of the operator (which are public) to overwrite a real GSM cell tower with openBSC.

[8] We found out about `AT+BREWAPP` and related commands by analysing the firmware.

D. Firmware

From the specifications, it is clear that the ACP-245 protocol supports an extensive array of functionalities, both in terms of which data the car sends to the server, and which commands the server sends to the car. That raised questions about which other data the car is sending and which other commands can be executed (possibly only accessible to the manufacturer).

To answer these questions, we inspected the TCU's firmware, which we obtained by dumping the entire flash memory over UART. The UART flashing protocol is proprietary, but it could be reverse engineered as follows. The TCU is based on the ARM-based AirPrime SL6078 chip [29]. The SIM card is soldered to the main board and directly wired to the AirPrime. For AirPrime development, the vendor (Sierra Wireless) provides an Eclipse IDE to develop firmware. As it is open source, we examined the code and found out how the module communicates with the device. We reversed how the IDE writes/reads the device's memory and developed a script to query the device for arbitrary memory locations. Since there isn't any memory protection or keyed developer access, we were able to dump the whole firmware and volatile memory during runtime. The firmware is approximately 3.6 MB in size, with 2.4 MB occupied by the boot-loader, the drivers, and the operating system. The remaining 1.2 MB consists of the Renault/Nissan application-specific code developed by Ficosa, an automotive supplier. The operating system, called OpenAT and developed by Sierra Wireless, is a complete software framework for M2M applications and encompasses a M2M-specific operating system, a range of software libraries, and a developing environment [30]. We focused primarily on the Renault/Nissan firmware. Thankfully, the binary code contained debug strings. Given the size of the firmware, this heavily simplified the reverse engineering task. As most methods were practically self-documenting, we focused our attention on the ones which contained sentences with relevant words like "HTTP(S)/SSL", "command/upload/encode". Our analysis revealed that the code structure is mostly event-based, in which the TCU registers a series of callbacks via OpenAT. The callbacks are triggered either by an event (e.g. SMS received) or by a timer (e.g. every 15 minutes).

E. Discovered "services"

Within the firmware, we found functionalities that the manufacturer labels *"services"* – as they are not always in the interest of the driver/owner, we keep the quotation marks when referring to them. They can be divided into four categories: (1) *alert and notification* such as theft notification, speed alert, geo-fencing, low-battery alert, etc., (2) *remote monitoring* like battery charge monitor, fleet and pay as you drive services, remote diagnosis, probing, etc., (3) *functionality control* such as remote door lock/unlock, charge scheduling, **battery charge blocking**, and (4) *configuration* which can essentially enable/disable each service on-demand. In Table V, we list the most relevant "services" we identified. Some common points:

- All can be enables/disabled and configured remotely via ACP service provisioning.
- All upload data to the cloud. These data always contain the car's VIN, battery state and GPS position.

Also, due to the *remote diagnosis* and *probe* services, Renault appears to be able to retrieve arbitrary information from the CAN bus. This implies that any data could be queried in real-time without the user noticing.

Besides the obvious security concerns, these *"services"* have also massive privacy implications: most of them constantly trigger data exchanges with the cloud, and always encode at least the VIN, IMEI, SIM-ID, battery state, and GPS position. This allows the receiving party to extrapolate very complex information like driving behaviour, locations, and habits. What is also worrying is that Renault can remotely enable, disable and configure these functionalities without the user's consent and knowledge. Furthermore, even the infotainment has full control of the TCU via AT commands over the UART: this means that a compromised infotainment might reconfigure the TCU. This is not unlikely, as the infotainment system is based on **Android 2.2** even though the car is only two years old and could theoretically receive over-the-air software updates. The SSL version of the TCU is similarly outdated – in fact it does not support TLS as it is based on **OpenSSL-0.9.6** from 2003, which has been proven to be vulnerable years ago.

F. Data Access

We do not know how (secure) the data is stored on the other end in the OEM's cloud and *who* has access to it. However, to get proof that people have access to that data, we set up an experiment. We went to a Renault dealer under pretence that something was wrong with the battery. The assistant just asked for the VIN and went to a computer to check the battery data. He did *not* go physically to the car. We asked for a screenshot of their proprietary program and obtained a print-out that lists all the trips the car ever did. It contains a table with the following columns: (I) Odometer, (II) Trip length (km), (III) State of Charge (SOC) beginning and end (used energy), (III) Used energy for driving, climate, lights (in kWh each), (IV) Outside temperature, (V) Time, and (VI) State (charging, driving). This clearly states that the manufacturer does not only accumulate data for statistical evaluation but that the personnel working at a dealer has detailed access to each car's history based on the VIN of the car.

VIII. Security considerations

Due to the incremental development nature of the automotive industry, the head unit was implemented using a very outdated version of Android, even though it is the only component which interacts with the outside world. Similarly disconcerting was the discovery that the head unit has full debug access to the TCU via UART. An attacker able to compromise the head unit can easily reconfigure the TCU via AT commands or simply flash a new firmware. Moreover, since the TCU has direct access to the car's power train CAN, an

Service type	Description
Fleet & Pay As You Drive	Huge set of services that seem to be used for handling fleet and Pay As You Drive (PAYD) services. When enables, they constantly exchange data with the cloud.
Eco	Send driving style, and eco-related statistics (e.g., trip statistics) to the cloud.
Remote diagnosis	Possibly used for remote diagnostics: it allows to remotely query an ECU via OBD.
Probes 1-10	Set-up periodic probes that retrieve data from the CAN and upload it to the cloud.
Battery blocking	Allows to remotely disable the battery.
Low battery	It sends to the cloud the battery state whenever it goes below a configurable threshold.
Charge History	Periodically send the battery state (i.e., voltages, temperature, health, etc.) with GPS position to Renault. This happens whenever the engine/battery state changes and every 15 minutes.
Remote door, start, horn & light	Allows to remotely control the doors, lighting, horn, and ignition.
Bulgar/Tow notification	Trigger a transmission whenever towing or burglary conditions are detected.
Maintenance alert	It triggers a transmission whenever a maintenance of the car is required. Simply compares the age of the vehicle to the current timestamp.
Speed/curfew/geo-fence alert	These services trigger a transmission whenever the car doesn't respect some constraints in terms of speed, time, or distance from a GPS position.

Table V: List of "services" provided by the TCU.

attacker can inject frames and control the vehicle. To verify our assumption, we successfully implemented attacks against some of the car's functionalities. More complex attack which completely take over the vehicle have already been demonstrated by researchers [8]–[10]. The only barrier that prevents an attacker from easily attacking a vehicle this way is the obscurity of CAN payloads. An attacker first needs to invest time, effort and money to manually reverse engineer these payloads, a process that until now required physical access to the vehicle. However, with `AutoCAN`, we demonstrate that this is not necessarily true and that an attacker can *passively* log the frames and analyse them offline – greatly lowering the cost and expertise needed for such attacks.

IX. Related Work

Recently, the topic of CAN traffic reverse engineering has gained popularity. The *READ* algorithm proposed by Marchetti and Stabili [21] is able to partially reverse engineer CAN signals. READ has several limitations, the most prominent being that it only tries to find signal boundaries and label counters and checksums. For a detailed comparison, refer to section VI. Another independent work tackling the same problem is the dissertation of Brent Stone, recently published by the US Air Force Institute of Technology [31]. The author employs machine learning methods based on Agglomerative Hierarchical Clustering computer over a Transition Aggregation N-Gram, which is comparable to our `HAMD`. The work focus is the segmentation of the frames, with very little signal correlation. Nevertheless, the proposed segmentation metrics are promising and we expect to integrate them in our toolset to improve its overall efficiency. Huybrechts et al. [32] propose to use LSTM and CNN networks to identify known signals. As it is based on supervised learning, their approach is unable to cope with unknown signals and requires access to external data sources of the vehicle (like GPS position) in order to work. Finally, pesé et al., in their just accepted work, propose LibreCAN [33]. LibreCAN, although novel, is limited by the fact it requires OBD ground truth data to work. This means the algorithm requires the attacker to actively inject OBD traffic into the CAN, and only works for data available via OBD.

In the field of automotive *privacy*, only little work has been published over the years. Researchers demonstrated what could be inferred from CAN traffic [6], [7]. However, none examined how difficult it is to gain those information and whether the manufacturer has already access to them. To our knowledge, our work is the first examining that topic.

X. Conclusion

In this paper, we investigate privacy in modern cars. To this end, we present an initial methodology to extract interpretations of raw CAN dumps. We demonstrate that an attacker can use our methodology to simplify the process of interpreting CAN frames – all without accessing the vehicle. We argue that even the knowledge of few signals can lead to privacy issues such as driver de-anonymization, derivation of driving patterns, location, and so on so forth. To complement our findings, we investigate the TCU of a modern electric car. We demonstrate how the lack of well-implemented security mechanisms makes the device, and thus, the whole vehicle, exploitable. We show how the TCU tracks the vehicle and discuss how this can be abused by the maker to extract private information and enforce contractual clauses, all without the user's consent and knowledge. We also show that car dealers have access to detailed, personal information about each trip a driver makes. Future work will focus on improving the current reversing methodology. The aim is to develop an open-source toolset able to automatically extract, label and correlate interpretations of cyber-physical system network traffic.

To conclude, we do not claim that data collection is dangerous per se. However, users should have the right of knowing *what*, *when* and *why* data are being collected and should have the right of denying this collection. This concept, although well understood by mobile and web providers, does not seem to be appropriately implemented by the automotive industry. With this paper, we hope to raise awareness on the issue and to motivate more researchers and authorities to look into the matter.

REFERENCES

[1] Insure The Box, “What is telematics insurance?” https://www.insurethebox.com/telematics.

[2] Money Super Market, “Telematics Car Insurance,” https://www.moneysupermarket.com/car-insurance/telematics/.

[3] Zubie, “Zubie device,” https://zubie.com/.

[4] TESLA Motors, “Blog: Most Peculiar Test Drive,” https://www.tesla.com/blog/most-peculiar-test-drive.

[5] District Court Cologne), “Bmw provides evidence of precise telemetry data in verdict 113 kls 34/15.”

[6] M. Enev, A. Takakuwa, K. Koscher, and T. Kohno, “Automobile driver fingerprinting,” *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 34–50, 2016.

[7] Y. Nishiwaki, K. Ozawa, T. Wakita, C. Miyajima, K. Itou, and K. Takeda, “Driver identification based on spectral analysis of driving behavioral signals,” in *Advances for In-Vehicle and Mobile Systems*. Springer, 2007, pp. 25–34.

[8] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, “Comprehensive experimental analyses of automotive attack surfaces.” in *USENIX Security Symposium*. San Francisco, 2011, pp. 77–92.

[9] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, “Experimental security analysis of a modern automobile,” in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 447–462.

[10] BBC Online, “Fiat Chrysler recalls 1.4 million cars after Jeep hack,” https://www.bbc.com/news/technology-33650491.

[11] ISO 11898-1:2015, “Road vehicles – Controller area network (CAN),” International Organization for Standardization, Standard, 2015.

[12] ISO 15765-2:2016, “Road vehicles – diagnostic communication over controller area network (docan) – part 2: Transport protocol and network layer services,” International Organization for Standardization, Standard, 2016.

[13] BMW Group, “BMW ConnectedDrive.”

[14] Volkswagen Group, “Volkswagen Connect app,” https://www.vwconnect.com/.

[15] European Commission, “eCall,” https://ec.europa.eu/transport/road_safety/specialist/knowledge/esave/esafety_measures_unknown_safety_effects/ecall_en.

[16] TESLA Motors, “Tesla App Support,” https://www.tesla.com/support/tesla-app.

[17] Clean Energy Wire, “Dieselgate – a timeline of Germany’s car emissions fraud scandal.”

[18] Official Journal of the European Communities, “Eu directive 97/39/ec: Adapting to technical progress council directive 75/443/eec of 26 june 1975 relating to the reverse and speedometer equipment of motor vehicles.”

[19] “A Hacker’s Look at Dieselgate,” https://debugmo.de/2015/12/dieselgate/.

[20] Robert Bosch GmbH, “Funktionsbeschreibung edc15+ p127-pea,” 2000.

[21] M. Marchetti and D. Stabili, “READ: Reverse engineering of automotive data frames,” *IEEE Transactions on Information Forensics and Security*, 2018.

[22] Renault Group, “Battery hire lease agreement,” http://myrenaultzoe.com/Docs/BatteryHireLeaseAgreement.pdf.

[23] Business Insider, “The 10 best-selling electric vehicles in the US this year so far.”

[24] Renault Group, “Renault R-LINK.”

[25] Sysmocom, “Sysmo NITB 2G starter kit,” http://www.sysmocom.de/products/lab/2Gstarterkit/index.html.

[26] Osmocom project, “OpenBSC,” http://osmocom.org/projects/openbsc.

[27] D. Strobel, “IMSI catcher,” *Chair for Communication Security, Ruhr-Universitat Bochum*, vol. 14, 2007.

[28] DENATRAN – Departamento Nacional de Trânsito, “ACP-245 – Application Communication Protocol,” http://new.denatran.gov.br/download/ACP%20245%20V%201.2.2_%2023_11_10_WITH_SMS.pdf.

[29] Sierra Wireless, “Airprime sl6087,” https://source.sierrawireless.com/devices/sl-series/sl6087/.

[30] ——, “Open AT Application Framework,” https://source.sierrawireless.com/resources/airprime/software/open-at-application-framework/.

[31] B. C. Stone, “Enabling Auditing and Intrusion Detection of Proprietary Controller Area Networks,” 2018.

[32] T. Huybrechts, Y. Vanommeslaeghe, D. Blontrock, G. Van Barel, and P. Hellinckx, “Automatic reverse engineering of can bus data using machine learning techniques,” in *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer, 2017, pp. 751–761.

[33] M. D. Pesé, T. Stacer, C. A. Campos, E. Newberry, D. Chen, and K. G. Shin, “Librecan: Automated can message translator,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

Appendix
Discovered signals

The Table summarise most signals we were able to derive from the vehicles under test. The knowledge of these signals and how they relate allows an attacker to compromise the vehicle or the users' privacy.

ECU	Signal	How derived	Lifetime	Description
Electric Vehicle Computer	Driving efficiency	Calculated	Forever	
	Average consumption	Calculated	Until Reset	
	Remaining range	Calculated	Forever	
	Battery serial number	Stored at production	Forever	
	Current gear	Lever position	HW switch	
	Motor RPM	RPM Sensor	Ephemeral	RPM motor + "fake" value (810)
	Torque(s)	Calculated	Ephemeral	Measured, requested, negative, etc.
	Throttle pedal	Pedal position	Ephemeral	
	Ignition on/off	Switch	Ephemeral	
	Brake pedal pressed	Pedal pressed	Ephemeral	
	12V battery amperage	Ammeter	Ephemeral	
	Traction battery consumption	Ammeter	Ephemeral	
	State of charge	Calculated	Forever	
	Traction battery voltage	Voltmeter	Ephemeral	
	Remaining battery capacity	Calculated	Forever	
	Max possible charging power	Calculated	Forever	Current limit for charging
	Traction battery temperature	Calculated	Ephemeral	Average from individual cells
	Charging cable plugged in	Switch on socket	Ephemeral	If charing cable is plugged in
	Time until 100% charged	Calculated	Ephemeral	
	Battery Health	Calculated	Forever	
	Cruise control target	Calculated	Forever	
	Cruise control exceeded	Calculated	Ephemeral	If speed exceed cruise limit
	Steering angle	Steering angle sensor	Ephemeral	Steering wheel angle in degrees
	Steering angle change	Calculated	Ephemeral	Speed of the steering wheel
	Battery cell voltage(s)	Voltmeter per cell	Ephemeral	Voltage for each cell (96x)
	Battery cell temperatures(s)	Temp. sensor per cell	Ephemeral	Temperature for each cel (96x)
	Energy recuperated	Calculated	Ephemeral	Energy gained by recuperation
Instrument Cluster	Action counter	Switch	Forever	Count when radio is switched on/off
	Speed in cluster	Calculated	Ephemeral	Displayed speed (ca. +5km/h)
	Trip distance (km)	Calculated	Until reset	
	Trip average consumption	Calculated	Until reset	
	Trip average speed	Calculated	Until reset	
	Outside temperature	Temperature sensor	Ephemeral	Displayed
	Hand brake pulled	Hand brake	HW switch	Displayed
Body Control Module	Age of vehicle	Stored at production	Forever	Seconds from first start
	Ignition counter	Calculated	Forever	How many times ignition on
	VIN	Stored at production	Forever	
	Indicator(s)	Steering column	Ephemeral	One switch/signal per indicator
	Lights(s)	Steering column	Ephemeral	One switch/signal per light
	Doors(s)	Door(s) switch column	Ephemeral	One switch/signal per door
Electronic Stability Control	Odometer	Calculated	Forever	Driven distance in Km
	Real speed	Calculated	Ephemeral	Real speed in km/h
	Longitudinal acceleration	ESP-accel sensor	Ephemeral	
	Transversal acceleration	ESP-accel sensor	Ephemeral	
	Yaw rate	ESP-accel sensor	Ephemeral	
	Wheels(s) RPM	RPM sensor on wheel	Ephemeral	One sensor/signal per wheel
	ABS warning light	Calculated	Ephemeral	ABS warning in cluster
	Odometer (precise)	Calculated	Ephemeral	In meter per seconds
Uncoupled Brake Pedal	Brake active	Calculated	Ephemeral	
	Torque recuperation	Calculated	Ephemeral	
	Brake pedal position	Pedal position	Ephemeral	
	Brake requested on UBP	Pedal	Ephemeral	Disk brakes requested pressure
Clima	Ventilation stage	Switch	Forever	Intensity of ventilation
	Evaporator threshold	Switch	Forever	
	Evaporator temperature	Temperature sensor	Ephemeral	
Airbag	Seat belt(s) buckled	Seat belt switch	HW switch	One switch/signal per seat

Table VI: Detailed list of discovered signals.