

Vehicle Safe-Mode: An After-Market Proof-Of-Concept

Tsvika Dagan
School of Electrical Engineering
Tel-Aviv University
Tel-Aviv, Israel
TDagan02@gmail.com

Yuval Montvelisky
Hamachvan Speedometer LTD
Holon, Israel
pniel_m@netvision.net.il

Avishai Wool
School of Electrical Engineering
Tel-Aviv University
Tel-Aviv, Israel
yash@eng.tau.ac.il

Abstract—The *vehicle safe-mode (VSM)* concept was designed to limit the potential damage of an identified cyber-attack on a vehicle. Unlike other defense mechanisms, that try to block the attack or simply notify of its existence, the VSM mechanism responds to a detected intrusion by limiting the vehicle’s functionality to safe operations, and optionally activating additional security counter-measures. This is done by adopting ideas from the existing mechanism of *Limp-mode*, originally designed to limit the damage of a mechanical or an electrical malfunction: it lets the vehicle “limp back home” in safety, in case a cyber-breach is detected, while reducing the potential damage of such an attack to the vehicle, its driver, the passengers, and its surroundings.

In this paper we demonstrate that *vehicle safe-mode* can be implemented as an after-market add-on, by developing a proof-of-concept system, and actively testing it on an operating Skoda Octavia vehicle. Once activated, our VSM system restricts the vehicle to *Limp-mode* behavior by guiding it to remain in low gear, taking into account the vehicle’s speed and the driver’s actions. The system overrides some of the normal gear-shifting logic by careful manipulation of the relevant CAN bus messages. Our system does not require any changes to the ECUs, or to any other part of the vehicle, beyond connecting the *safe-mode* manager to the correct bus.

As part of the VSM development we analyzed the Skoda’s CAN bus topology, and discovered that it is possible to connect our system to the Powertrain bus from inside the passenger compartment, without opening the hood. We reverse-engineered some of the vehicle’s CAN messages and their timing characteristics, which allow the *safe-mode* manager to identify the vehicle’s condition and construct the overriding control messages. We implemented the *safe-mode* manager to work in real-time when connected to the vehicle’s CAN bus, and tested it in multiple driving scenarios, taking the VSM-augmented vehicle onto the roads and successfully demonstrating its functionality.

I. INTRODUCTION

A. Motivation

Modern vehicles are susceptible to cyber-attacks: this is since they are controlled by multiple dedicated computers (electronic control units - ECUs) that are typically connected to each other over a CAN bus, and also to the outside world—often by wireless protocols (WiFi, Bluetooth, Cellular, etc.). These conditions, and the introduction of new technologies that allow remote access to the vehicle internal systems, make vehicles vulnerable to new attack vectors of increasing number. Researchers have already shown that these attacks can be both feasible and severe (e.g., attacks on Jeep [17] and Tesla [32]).

Many defense mechanisms have been offered to block the attacks. However, none of them are perfect—and may never

be—motivating the need for a solution to limit the potential damage of an attack that already passed the vehicle’s first line of defense. The recently suggested concept of a vehicle *safe-mode* [9] can be a good candidate for this purpose.

B. Related Work

1) *Attacks on Vehicles*: Research into vehicle cyber-security has been growing since the first publication of Koscher et al. [22] in 2010. Using sniffing, fuzzing and reverse engineering of ECU’s code, the authors succeeded in controlling a wide range of vehicle functions, such as disabling the brakes, stopping the engine, etc. Checkoway et al. [4] showed that a remote attack, without physical access to the vehicle, is also possible (via Bluetooth, cellular radio, etc.). Valasek and Miller [30] demonstrated actual attacks on Ford Escape and Toyota Prius cars via the CAN bus network. They affected the speedometer, navigation system, steering, braking and more. In 2015 it was reported [16], [17] that they remotely disabled a Jeep’s brakes during driving, and caused Jeep to recall 1.4M vehicles. Foster and Koscher [14] have also reported of the potential vulnerabilities in relatively new commercial OBD-II dongles (such as those used by insurance companies to track one’s driving) which support cellular communication and may be even exploited via SMS. In 2016, a team of researchers from Keen Security Lab demonstrated a successful attack on the Tesla electrical vehicle [32], taking control over the vehicle through a bug in the Infotainment unit’s browser, forcing the company to release an over-the-air software update.

2) *Defense Mechanisms*: Several ideas were offered to secure vehicles against cyber-attacks, including both active and passive solutions. One approach is to try and secure the internal communication of the vehicle - typically a CAN bus, by adding authentication to the messages (e.g., by using a cryptographic Message Authentication Code (MAC)). Several ideas were suggested, ranging from adding part of a MAC tag to the actual message’s data field, to splitting the MAC into several pieces and layers as offered by Glas and Lewis [15]. Van Herrewege et al. [39] suggested to use a new light-weight protocol to better fit the CAN bus limitations.

A similar approach was adopted by the AUTOSAR standard, as defined by the Secure Onboard Communication (SecOC) mechanism [2], to add some authentication and replay prevention to the vehicle’s internal networks. Note that all these

cryptographic solutions require secret keys and/or random nonces—hence they rely on a good source of randomness to produce the keys cf. [8].

A different, non-cryptographic, family of solutions is based on destroying non-legitimate spoofed messages. These include suggestions by Matsumoto et al. [29], Kurachi et al. [23], [24], Ujiié et al. [37], and the *Parrot* system of Dagan and Wool [7], [13].

Another approach is to try and identify un-authorized access to the internal network of the vehicle, by using Anomaly or Intrusion Detection Systems (IDS). Markovitz and Wool [28] demonstrated the ability to classify the traffic over the CAN bus, where Marchetti et al. offered some anomaly detection mechanisms, based on an information theoretic algorithm [27] and on inspection of sequences of IDs [26]. Hamada et al. [19] offered to implement an IDS system that relies on the traffic density of some periodic messages.

Newer works offered to rely on some unique characteristics of the ECU to build an IDS for the CAN bus. Lee et al. [25] used the time of arrival of Remote-frames reply packets to identify potential attackers; whereas both Cho and Shin [5], and Choi et al. [6] used the voltage characteristics of an ECU to identify attacks.

Some leading manufacturers, such as NXP [31] and Bosch [3] offer a variety of products to secure the vehicles, ranging from Hardware Secure Modules (HSMs) to full fledged secure gateways. The existence of these products fits the wide-spreading holistic (in-depth / layered) approach for vehicle cyber-security, as described by Van Roermund et al. [40].

3) *Vehicle Safe Mode*: Augmenting all the above-mentioned defense mechanisms is the need to notify the driver of potential attacks [20], using different methods according to the notification severity. Note that the recently released UN-ECE Resolution on the Construction of Vehicles [38] Annex 6 (4.3.3) in fact requires driver notification in case a cyber-attack is detected.

The *vehicle safe-mode* (VSM) concept [9] was designed to limit the potential damage of an identified cyber-attack on a vehicle. Unlike other defense mechanisms, that try to block the attack, in addition to notifying the driver of its existence, the VSM mechanism responds to a detected intrusion by limiting the vehicle’s functionality to safe operations, and optionally activating additional security counter-measures. This is done by adopting ideas from the exiting mechanism of *Limp-mode*, originally designed to limit the damage of a mechanical or an electrical malfunction: it lets the vehicle “limp back home” in safety, in case a cyber-breach is detected, while reducing the potential damage of such an attack to the vehicle, its driver, the passengers, and its surroundings.

C. Contribution

In this paper we demonstrate that *vehicle safe-mode* can be implemented as an after-market add-on, by developing a proof-of-concept system, and actively testing it on an operating Skoda Octavia vehicle. Once activated, our VSM system restricts the vehicle to *Limp-mode* behavior by guiding it to remain in low gear, taking into account the vehicle’s speed and the

driver’s actions. The system overrides some of the normal gear-shifting logic by careful manipulation of the relevant CAN bus messages. Our system does not require any changes to the ECUs, or to any other part of the vehicle, beyond connecting the *safe-mode* manager to the correct bus.

As part of the VSM development we analyzed the Skoda’s CAN bus topology, and discovered that it is possible to connect our system to the Powertrain bus from inside the passenger compartment, without opening the hood. We reverse-engineered some of the vehicle’s CAN messages and their timing characteristics, which allow the *safe-mode* manager to identify the vehicle’s condition and construct the overriding control messages. We implemented the *safe-mode* manager to work in real-time when connected to the vehicle’s CAN bus, and tested it in multiple driving scenarios, taking the VSM-augmented vehicle onto the roads and successfully demonstrating its functionality.

Organization: In the next section we provide some background on the vehicle’s *Limp-mode*. In Section III we provide a short description of the *vehicle safe-mode* mechanism. Section IV describes some preliminaries and the setup for our PoC, followed by Section V that describes our initial findings. Section VI describe the actual PoC. We conclude with Section VII.

II. PRELIMINARIES

A. *Limp-mode*

Limp-mode (also known as *Fail Condition*) was originally designed as a safeguard to limit the potential damage of either a mechanical or an electrical malfunction, and let the vehicle “limp back home” for treatment, without risking further damage and without forcing the vehicle to a complete stop. In modern vehicles, *Limp-mode* is activated automatically after an ECU detects a malfunction in one or more vehicle subsystems.

It is possible to distinguish between two different types of *Limp-modes*: a *local limp-mode* that is limited to the operation of a single ECU; and a *global limp-mode* affecting the global state of the vehicle.

Local limp-mode is a feature often supported by micro-controllers used to implement ECUs. It is usually provided as a physical pin that, when activated by applying the proper voltage, makes it possible to override the normal behavior of the micro-controller and drive the output pins directly to pre-configured settings (see as an example the technical documentation of the DRV8305-Q1 automotive micro-controller [35]). *Local limp-mode* can be easily deactivated by restoring the normal voltage to the *Limp-mode* pin, thus restoring the normal operation of the micro-controller.

Global limp-mode is activated when one of the central ECUs connected to the in-vehicle network, usually the Body Control Module (BCM), or the Engine Control Module (ECM) detects possible fail conditions by analyzing the values of the messages received from the CAN bus (e.g., see the Central BCM produced by Infineon [21]). For instance, *global limp-mode* may be activated if the coolant temperature rises above safe values [18] or if the Powertrain control module detects a failure (or near-failure) condition in the transmission [36].

Depending on the type and on the severity of the failure, the central ECU triggers a set of operations that restrict the vehicle to a limited set of failsafe states. As an example, when in *Limp-mode* the vehicle speed might be electronically limited to a set threshold, the transmission might be fixed in a second gear or, if an issue related to the engine is detected, *Limp-mode* can shut the engine off and gradually reduce the vehicle speed to a complete stop. The exact counter-measures deployed when in *Limp-mode* depend on the specific settings defined by the car manufacturer.

Depending on the car maker and model, *global limp-mode* may be implemented by activating the *local limp-mode* of some peripheral ECUs, letting the main ECU directly control them.

Deactivation of the *global limp-mode* also depends on the nature and severity of the detected failure. For example, *Limp-mode* that is activated due to the detection of transient failure conditions, is usually reset automatically after restarting the vehicle, or after a predefined amount of time. In some cases, the car owner can perform a sequence of operations that resets the *Limp-mode* for non-severe failures, such as switching the car ignition on, and pressing and releasing the throttle pedal for a given number of times [1]. On the other hand, more severe failures may require a manual reset of the *Limp-mode*, which is usually performed by operators of authorized car services by physically connecting to the OBD-II port and executing proprietary diagnostic protocols.

In the remainder of this paper we use the term *Limp-mode* to refer to the *global limp-mode*.

III. THE VEHICLE SAFE-MODE MECHANISM

A. Overview

The concept of vehicle *safe-mode*, as described in [9], is similar in principle to the *Limp-mode* mechanism (recall Section II-A): The *safe-mode* mechanism is offered to let the vehicle “limp back home” in case a cyber-breach is detected, while reducing the potential damage of such an attack to the vehicle, its driver, the passengers, and its surroundings.

The Vehicle *safe-mode* system operates as follows: When a cyber-attack is detected, a *safe-mode* manager (*SMManager*, see Section III-C) puts the vehicle into a *safe-mode* condition—in which several operations are limited or disabled, by broadcasting an alert triggering message (*TMessage*) to other ECUs. The *SMManager* bases its decision on any existing IDS-like systems, that flag suspicious cyber-related events. This decision should typically include the recommended level of alert and the chosen reaction. See Figure 1 for the system overview.

Two modes of operation were suggested by [9]: *Transparent-mode* and *Extended-mode*. In *Transparent-mode* (Section III-B) the *SMManager* only causes the neighboring ECUs to enter into their pre-configured *Limp-mode* state, in order to limit the functionality of the vehicle and reduce the potential danger. This is done by broadcasting the relevant messages that put the chosen ECUs into their pre-configured state of *Limp-mode*.

The main advantage of this mode is its immediate applicability to virtually all modern vehicles, since the introduction of the *SMManager* is transparent with respect to all other ECUs. In this paper we focus on this mode. Ideally the system can be

TABLE I
A SKETCH OF PRE-EXISTING *TMessages* THAT CAN TRIGGER *Limp-mode*

ECU	Msg ID	Data
ECM	014	“Dangerous high engine temperature”
ECM	014	“Major engine malfunction”
ABS	004	“Dangerous low oil pressure”
TPM	020	“Dangerous low air pressure”

deployed by adding a single OBD-connected entity to include the *SMManager*, with optional IDS capabilities (however as we shall see, an OBD-II connection is not always sufficient).

Alternatively, the *Extended-mode* requires adding a novel software component, called *safe-mode* client (*SMClient*) to chosen ECUs. The purpose of the *SMClient* is to process and react to the custom *TMessages* sent by the *SMManager*. Using this mode adds more flexibility to the system, by making it possible to design and implement customized reactions per individual ECU, and state of alert.

Special care should be given to the recovery options - that make it possible to exit from *safe-mode* and to restore full vehicle functionality. This is required to make sure that the attacker will not have an easy way to take the vehicle out of *safe-mode*, while simultaneously ensuring that the driver will not have a too-difficult time to restore normal operation.

B. Transparent-mode

In this mode of operation, the *SMManager*’s goal is to put the relevant ECUs into *Limp-mode* in order to reduce the potential damage of an identified attack, by triggering the pre-existing *Limp-mode* mechanism of each relevant ECU.

For this purpose, the *SMManager* should maintain a list of all the CAN bus messages that cause each ECU to enter *Limp-mode*. This list can be maintained by a simple updatable table of the relevant *TMessages* per ECU (see Table I). Note that this table can include several different lines per ECU, in case there are multiple *TMessages* per ECU (in this case the *SMManager* can decide, per ECU, whether to broadcast all, or only some of the available *TMessages*).

The properties of this mode potentially make the *Safe-mode* protection applicable to any existing vehicle, e.g., by connecting an after-market device (to include the *SMManager* and some anomaly-detection component) to its OBD-II port

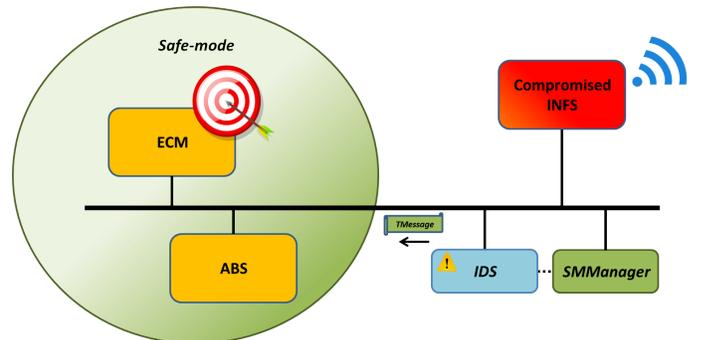


Fig. 1. The VSM system overview. Note that the *SMManager* can be connected directly to the IDS system, or alternatively, get its feedback over the bus.

or directly to the appropriate CAN bus. A more sophisticated after-market device (e.g., one using a smart-phone) can include additional notification and recovery options to the vehicle’s driver, see [9].

The drawback of this mode is that *safe-mode* reactions are bound to be the same reactions that the car maker already planned for the *Limp-mode*. Hence counter-measures that are designed specifically against cyber-breaches cannot be implemented.

Special care should be taken under this mode to make sure that no collision will occur between the *SMManager’s* *TMessages* and genuine messages of the original responsible ECU.

C. The Safe-mode Manager

The *SMManager* is responsible to process the IDS alerts, calculate the vehicle alert-level (*AL*), decide on the relevant reaction-level (*RL*), and finally put the vehicle into, and out of, *safe-mode*, by broadcasting the relevant *TMessages*.

The *SMManager* may also be responsible for any related key-management aspects in case of using cryptography for either the protection of the *TMessages* themselves, or the transfer into secure-communication when under attack.

Regardless of the selected configuration, the *SMManager* should be able to receive the IDS alerts, either directly from the bus, or from its hosting ECU (which can also comprise of both the *SMManager* and the IDS).

We also note that the *SMManager* can be implemented in either software or hardware—a hardware implementation should increase the cyber-resistance of the suggested mechanism, while possibly increasing its cost and making deployment more challenging.

1) *The vehicle Alert-level*: Independent of the chosen implementation (Transparent or Extended), the *SMManager* is responsible for evaluating the vehicle’s Alert-level (*AL*). Different levels of alert reflect different threat levels and imply the deployment of appropriate reactions, as will be further discussed in Section III-C2.

To evaluate the current *AL*, the *SMManager* relies upon any anomaly detection system deployed within the vehicle. In particular, intrusion detection systems (IDS) represents the main source of information useful for *AL* evaluation. IDS for in-vehicle networks of modern vehicles have already been proposed in the literature [19], [26], [27]. All these systems analyze different features of the messages broadcast over the CAN bus and issue alerts whenever evidence of an attack is found. The *SMManager* collects and analyzes all of the security related alerts (or lack thereof) and modifies the current *AL* accordingly.

For concreteness, as an example we suggest that the *AL* can be comprised of five different levels to represent the severity of the alert, denoted by *AL1* (low severity) to *AL5* (critical severity).

2) *Reaction-Matrix*: The *Reaction-Matrix* is the structure used by the *SMManager* to determine the reaction-level (*RL*) that encodes the required protective steps and notifications. The calculation of the *RL* depends on two metrics:

- the current Alert-level (*AL*, recall Section III-C1)
- the current Vehicle-condition (*VC*).

The value of the current *VC* represents the current conditions of the vehicle dynamics, including speed, yaw, roll, pitch, lateral acceleration and outputs of the ABS and ESC (‘Electronic stability control’) systems. Intuitively, it is important to consider the current vehicle conditions to make sure that reactions decided by the *SMManager* are appropriate, and do not cause more harm than the attack itself.

As an example, if counter-measures were to be deployed without considering the vehicle conditions, the *SMManager* might decide to exclude the electronic stability protection or other advanced driving assistance systems. While this decision may be the most appropriate for a vehicle running at low speed on a straight road, it may cause severe safety risks to a vehicle in dangerous driving conditions (e.g., at high speed, or under high lateral accelerations). To prevent similar situations, the reaction-matrix makes it possible to react to the same *AL* by deploying different counter-measures based on different vehicle conditions.

The calculated *RL* is used to determine the most appropriate reaction, aiming to bring the vehicle to a safe state that nullifies or limits the safety consequences of the detected cyber-breach.

An example for a Reaction-Matrix is depicted in Figure 2 to include the different *RL* to match every possible combination of a given 5x5 *AL/VC* structure.

IV. PRELIMINARY SETUP FOR THE PROOF OF CONCEPT

A. Overview

Modern vehicles typically have several CAN buses, interconnected by a filtering gateway. Therefore, to develop an after-market *Transparent-mode safe-mode* mechanism, a basic understanding of the vehicle’s CAN bus topology is required: connecting the *SMManager* to the wrong bus would render it useless.

Furthermore, a good understanding of the syntax, semantics, and frequency of the relevant CAN messages is necessary. This knowledge help us identify which messages carry the Vehicle Condition (*VC*); How to construct, and when to broadcast the

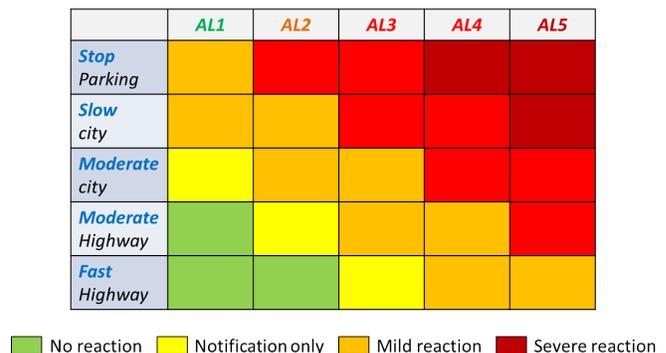


Fig. 2. An example for a 5x5 *Reaction-Matrix*. Note that the rows represent the current *VC* (speed/location), the columns represent the chosen *AL* (1-5), and the colors of the internal blocks represent the chosen *RL* (nothing to severe).

Tmessages, in order to make the relevant ECUs limit their operation under the desired concept of *safe-mode*.

Clearly the vehicle manufacturer has all the above-mentioned knowledge. However, manufacturers generally do not provide the specifications of their ECU’s CAN messages, nor do they provide much detail about the CAN bus topology. Hence a developer of an after-market *SMManager* needs to resort to some trial-and-error, and to identify and parse the key CAN messages by reverse-engineering sniffed traffic.

In the next sections we describe the topology discovery we went through, and the results of our traffic analysis. Our findings are specific to the vehicle we experimented with—a 2015 Skoda Octavia—however we believe that they are indicative of other modern vehicles.

B. Equipment and Software

In our experiments we used two *Peak-system* PCAN-USB devices [33]. Both PCAN-USB devices were controlled via USB connections by a Dell laptop running Windows, using the *PCAN-View* control software.

The first USB device was used primarily as a sniffer: we used the software to capture the traffic over the vehicle’s CAN bus into *.trc* trace files (which include the messages’ ID, Data, Type, and time-stamp). We also used this device to broadcast some messages during the initial stage of our analysis.

The second USB device was added for the *SMManager* testing, where the first device served as an observer, capturing all the traffic over the CAN bus. The two devices were connected to the same location on the CAN bus through a *Peak T-connector* (two-to-one D9 connector).

We developed the *SMManager* in Python, using the PCAN-Basic software libraries and DLL (PCANBasic.dll) to access the devices drivers. For voice notifications we used the *pyttsx3* Text-to-speech python library, that relies on Windows’ SAPI5 (Microsoft Speech Application Programming Interface) synthesizer.

C. Bus Topology and Connection Choices

In order to identify the relevant CAN messages we needed to sniff the CAN bus traffic, which entails connecting our USB-based sniffing device to the CAN bus.

Our first attempt was to connect the sniffer through the OBD-II port, which provides a fully legitimate connection to a CAN bus with minimal tinkering. Unfortunately, after connecting to the OBD-II port, we failed to observe the CAN traffic we were looking for: hardly any messages were received. This lead us to conclude that traffic reaching the OBD-II port is heavily filtered, so connecting through it is not suitable for our purposes.

Our second attempt was to connect the USB device to the dashboard unit’s CAN line. Our intuition was that since the dashboard displays the speed and current gear position, it should also receive the CAN messages we care about. This turned out to be the case, and sniffing CAN traffic from the dashboard CAN line allowed us to observe the desired messages and begin with our analysis, (see Section V below).

However, when we tried to re-broadcast messages from our USB device onto the dashboard unit’s CAN line, we

discovered that some of our messages have no effect. We concluded that additional bus segmentation and filtering exists: for instance, the shift-gear messages we tried to re-broadcast were not allowed to cross over from the dashboard unit’s bus to the relevant Powertrain bus. Additional investigation taught us that the Skoda Octavia indeed has separate Convenience and Powertrain buses.

This made us look yet again for a suitable connection point, that provides access to the Powertrain bus. We discovered that we can achieve this access *from inside the passenger compartment*, without opening the hood: by tapping the gear selector unit’s CAN bus lines (see Figure 3). In the Skoda the gear selection is electronic, so clearly the gear selector is able to broadcast gear-shifting messages onto the Powertrain CAN bus, without filtering. In all the following experimentation we used this connection point.

V. MESSAGE AND FIELD IDENTIFICATION

A. Overview and notations

The next stage of our experiments focused on analyzing the CAN bus traffic in the search of messages that bring the vehicle (or at least some of its key ECUs) into a predefined *Limp-mode*. Finding these messages would have allowed us to implement the *Transparent-mode* version of the *safe-mode* mechanism, exactly as described in Section III-B. Note that *Limp-mode* is typically activated when the vehicle senses a serious mechanical or electric failure—yet we needed to trigger it’s activation without damaging the vehicle. Under this constraint, our attempts to trigger *Limp-mode* included disconnecting fuses and unplugging various sensors. Unfortunately, we were unable to find messages related to any *Limp-mode* activation: either we did not manage to simulate a serious-enough fault (e.g., disconnecting the fuel pressure sensor), or we caused failures that completely prevented driving the vehicle (e.g., disconnecting the gearbox’s main fuse).

Failing to find CAN messages that put the Skoda in a real *Limp-mode* (e.g., permanent low gear / limited RPM), we

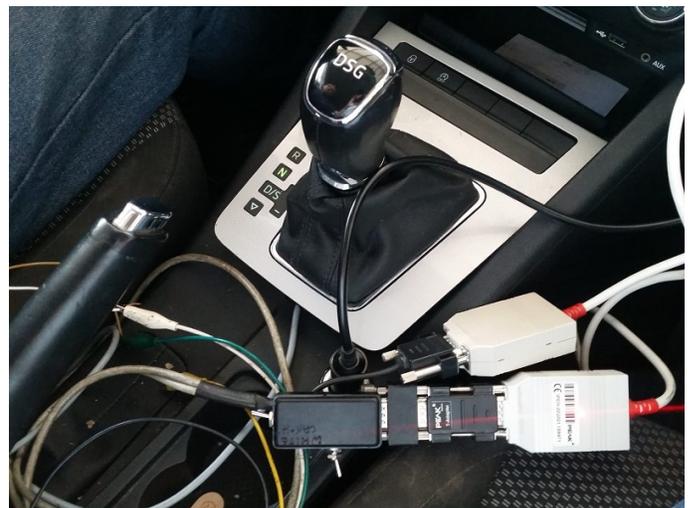


Fig. 3. CAN bus tapping. Note that both USB devices are connected to the same line by using the black T-connector.

decided to mimic the behavior of *Limp-mode* from the safety perspective, by actively guiding the vehicle to stay in low gear, while taking into account its speed as an indicative ‘VC’ parameter.

To implement this alternative of “simulated Transparent-mode”, we needed to identify the messages that carry the vehicle’s speed and current gear, in addition to the message that makes the gear change its position. For each message we had to analyze the data field, understand its semantics, and learn its typical pattern of broadcast. The following Section V-B describes these identified messages and their analysis.

Recall that every CAN message is characterized by its ID (an 11-bit value). In our analysis of each message we also record its length (we only observed 4-byte or 8-byte messages), its frequency (which we denote by the message’s typical inter-arrival time in milliseconds), its assumed field structure, and the level of its semantics we understand. Following [28] we found that each of the messages we analyzed is constructed of several bit fields that carry semantic meaning, surrounded by seemingly ignored “don’t care” bit fields. All the semantic fields we identified are either 4- or 8-bit wide and are nibble-aligned. Below we use ‘x’ to indicate a don’t-care nibble, a single lower-case letter to indicate a semantic 4-bit field, and two Upper-Case letters to indicate an 8-bit field. For each field we also indicate its classification into the categories: Sensor / Counter / Multi-Value, following [28].

B. Identified Messages

We identified and analyzed the following messages:

- **ID: 0x0AF, gear selector position**

Message frequency: 10 msec

Length: 4 bytes

Data field structure: [gb \bar{g} c xx xx]

Data field analysis: In the left (most significant) byte, the ‘g’ digit is a multi-value field representing the gear selector’s position; see Table II for the semantic meaning of the values we identified. The ‘b’ digit is a multi-value field which seems to represent whether the brakes are applied (b=2), or not (b=3).

In the second byte, the ‘ \bar{g} ’ digit represents a bitwise-complement digit of ‘g’ (where $g + \bar{g} = 0xF$ always holds). The ‘c’ digit is a 4-bit increasing cyclic counter field. The remaining 2 bytes seem unimportant.

For example, two consecutive 0x0AF messages indicating the gear selector set to the Parking ($g=0x8$) position, with the brakes applied (b=2), can appear as: [82 70 xx xx], and [82 71 xx xx].

- **ID: 0x394, gear position**

Message frequency: Typically 160 or 200 msec. Can appear at higher frequency after a gear shift event.

Length: 8 bytes

Data field structure: [xx xc xx GP xx MG xx xx]

Data field analysis: The ‘c’ digit is a 4-bit increasing cyclic counter field. ‘GP’ is an 8-bit multi-value field which seems to represent the gear position¹. We identified

the following values: 0x1A and 0x0A: Neutral; 0x0i: gear in position i, where $i = 1$ to 7. We are uncertain of the difference between Neutral values 0x1A and 0x0A: they seem related to whether the brakes are on or not. ‘MG’ has similar values to those of the ‘GP’ field, and appears in addition when in Manual state.

- **ID: 0x30B, vehicle speed**

Message frequency: 50 msec.

Length: 8 bytes

Data field structure: [10 2c 00 00 00 00 SP 10]

Data field analysis: The ‘c’ digit is a 4-bit increasing cyclic counter field. ‘SP’ is an 8-bit sensor field which seems to represent 3×the vehicle’s speed in Km/h. For example, the ‘SP’ field for 30 Km/h is 0x5A (since $3 \times 30 = 90 = 0x5A$). All the remaining message bits are fixed.

- **ID: 0x391, gas pedal position**

Message frequency: 100 msec.

Length: 8 bytes

Data field structure: [xx xx xx xx xx PD xx xx]

Data field analysis: ‘PD’ is an 8-bit sensor field which seems to represent the gas pedal position, ranging from 0x26 (idle position) to 0xD0 (fully open throttle).

C. Gearbox Manipulation

After identifying the relevant messages and understanding their usage, we turned to testing the ability to broadcast them ourselves (using the PCAN-View software) and observing the results. Since our goal is to guide the vehicle to a low gear (to simulate *Transparent safe-mode* behavior), we focused on manipulating the gear selector, 0x0AF messages.

For this purpose we tried to broadcast the 0x0AF message with its ‘g’ field set to 0xB (see Table II)—spoofing the gear selector’s request to reduce the gear by one position.

We discovered that this is not enough: the cyclic counter value (‘c’), and the complement-to-0xF (‘ \bar{g} ’) fields cannot carry arbitrary values. Failing to provide the right value for the complement-to-0xF (which needs to be 4, since $0xB+0x4=0xF$), or using a wrong cyclic-counter value (with a value more than 7 off from the expected value of +1), made the target ECUs (probably the gearbox ECU) ignore our message; Whereas, using the right values, allowed us to actually make the gearbox reduce the gear by one, as desired.

We note that even using the right content in the spoofed message, as explained above, does not guarantee success—the gear did not always obey the instruction. We assume that our control over the gearbox was not perfect since our attempts failed to pass some internal logic-checks implemented by the gearbox ECU. Such a check may consist of using internal analog magnetic speed sensors to make sure the chosen gear fits the current speed of the vehicle, as a self-protection mechanism, preventing mechanical damage to the gearbox itself (e.g., the gearbox may ignore instructions to switch to a too-low gear at high speed).

In addition, we found that broadcasting the spoofed reduce-gear-message more slowly, at an interval of 50 msec, instead of

¹We found the message to be occasionally unreliable after broadcasting spoofed gearbox manipulation messages.

TABLE II
VALUES OF THE GEAR SELECTOR POSITION: MESSAGE ID 0x0AF, MOST SIGNIFICANT NIBBLE ('g')

Selector's position	Park	P/N intermediate	Reverse	Neutral	Drive	S Drive	Manual	Manual up	Manual down
The value of 'g'	0x8	0x9	0x7	0x6	0x5	0xC	0xE	0xA	0xB

broadcasting it at the 0x0AF message's original 10 msec cycle, improves the likelihood that the gearbox obeys our instructions.

The outcome of this investigation gave us the ability to control the chosen gear: i.e., the ability to reduce the gear almost at will. The only caveat is that this worked as long as the instruction didn't contradict other conditions such as when the speed was too high or when the gas pedal was applied during our attempt to reduce the gear.

Additional experiments showed us that instructing the gearbox to shift to the Neutral position (by using the 'g' field value 0x6) was more resilient to the internal gearbox safety mechanisms, which gave us some additional choices in our *SMManager* PoC.

VI. THE SAFE-MODE PROOF OF CONCEPT

A. *SMManager* implementation

Identifying the relevant messages, and learning how to use them correctly, allowed us to continue into the final stage of our PoC — implementing and testing the *SMManager*. Algorithm 1 shows the pseudo code of our *SMManager*.

Based on the ability to (at least partially) control the chosen gear, and to identify the vehicle's speed and current gear, we chose the following tactics. We define two parameters (maxSafeGear and maxSafeSpeed). If the gear position is above maxSafeGear and the speed is above maxSafeSpeed , and an attack is detected, then the *SMManager* attempts to reduce the gear to the desired level (of maxSafeGear).

In our PoC we used $\text{maxSafeGear}=1$, and $\text{maxSafeSpeed}=40\text{Km/h}$, under the assumption that lower values are relatively safe and acceptable, even when under attack.

Both the current gear and current speed are continuously monitored by the *SMManager* to allow proper reaction. We achieve this by a background process that analyzes incoming messages by ID, and processes the relevant ones (e.g., 0x394 for the gear, and 0x30B for the speed); We also needed to monitor the gear selector's 0x0AF messages to ensure we have the latest cyclic counter value ('GSCounter') ready when the *SMManager* needs to broadcast its own 0x0AF messages (procedure `analyzeMsgs` in Algorithm 1).

Beyond controlling the gear we also demonstrate a driver notification capability (recall section III-C): when *safe-mode* is engaged we activate an existing Skoda functionality, which consists of showing a "Press brakes" text and a warning symbol on the dashboard display (Figure 4). To do so the *SMmanager* broadcasts a message triggering this dashboard feature at a (configurable) rate of about once every 5 seconds, to create a 'blinking-warning effect'. The triggering message itself is the gear position 0x394 message with a special data field of: [25 4D 00 50 00 00 00 00]. Note that this special message does not follow the structure we identified in other 0x394 messages,

Algorithm 1 The *SMManager* pseudo code

```

1: procedure APPLYSAFEMODE()
2:   beginTimer()
3:   beginMsgAnalysisProcess(analyzeMsgs())
4:   notifyDriver("Intrusion Alert")
5:   while underAttack do
6:     if above (maxSafeSpeed) then
7:       ENGAGE()
8:     end if
9:   end while
10: end procedure
11:
12: procedure ENGAGE()
13:   waitForTheRightTime(50 msec)
14:   if gasPedalPressed then
15:     broadcastMessage(PutInNeutral)
16:     voiceMessage="Please do not accelerate"
17:   else
18:     if above (maxSafeGear) then
19:       broadcastMessage(ReduceGear)
20:       voiceMessage="Shifting gear down"
21:     end if
22:   end if
23:   if (timer % notificationCycle = 0) then
24:     broadcastMessage(WarningMsg)
25:     notifyDriver(voiceMessage)
26:   end if
27: end procedure
28:
29: procedure ANALYZEMSGS
30:   for incoming Message m do:
31:     id = getMessageID(m)
32:     if id in IdList then:
33:       processMsg(m)
34:       updateState(speed, gear, gasPedal, GSCounter)
35:     end if
36:   end for
37: end procedure

```

nor does it bear any clear relationship with a gear position; we have no explanation for this.

In addition, we added voice notifications, to keep the driver aware of both the identified attack, and the chosen reaction (lines 4 and 25 in Algorithm 1). Doing so improves the effectiveness of the system, by allowing the driver to cooperate (e.g., when the system asks the driver not to accelerate when the car is above the chosen safe speed), and reduces the probability of undesired driver reactions.

The initial testing showed us that the gearbox tends to ignore our request to reduce the gear at high speed or when the

gas pedal is pressed (recall Section V-C)—thus making our system effective only with ‘cooperative drivers’ that do not try to accelerate when *safe-mode* is engaged; i.e., drivers that obey the displayed “Press brakes” warning and voice notifications.

This made us modify our tactics: for demonstration purposes, we added the option to switch the gear to Neutral, when the gas pedal is pressed. For this purpose we also monitored the gas pedal’s position (message 0x391). The combined control appears in procedure Engage() in Algorithm 1: When the vehicle speed and gear position are above the predefined safe limits, we reduce the gear by one when the gas pedal is not pressed, and shift the gear to Neutral otherwise. This allows us to demonstrate *SMManager*’s ability to override insecure actions of a ‘non-cooperative’ driver (or a more aggressive attacker). We note that switching to Neutral is generally an unsafe action. We believe that in a real system, additional, or different actions are merited to safely handle scenarios in which *safe-mode* is engaged and down-shifting the gear is not possible, e.g., when the vehicle is on a downhill.

B. Live Tests

After implementing the *SMManager* PoC we connected it to our vehicle and took it for some test drives. We observed the desired behavior when we simulated an attack detection trigger: the *SMManager* gradually reduced the gear down to the first gear, slowing the car down to 40Km/h, while blinking the warning message on the dashboard: and if the driver tried to press the gas pedal—the gear switched to Neutral.

Testing with a ‘Cooperative Driver’

A short video of one of our test drives, with a ‘cooperative driver’, can be found at [11]. The video begins with a voice notification to the driver, letting him know that the system is up and running, and that the chosen maxSafeSpeed value was set at 40Km/h. A few seconds later (at timestamp 0:51), when the vehicle is driving at 77Km/h in D6 (sixth gear), the *SMManager* alerts the driver of a detected intrusion (manually simulated by the experimenter).

From this moment on, and as long as the vehicle is above the chosen safe speed of 40Km/h, the *SMManager* asks the driver to slow down using both visual and voice notifications (e.g., see timestamps 0:56 and 1:00). During this time, the *SMManager* also tries to slow the vehicle down by manipulating the gear (recall procedure ENGAGE, in Algorithm 1). Since the driver is ‘cooperating’ and does not try to accelerate after the attack

was detected, the *SMManager* chooses to shift the gear down (line 19 in Algorithm 1) as long as it is above the chosen maxSafeGear (which is set to first gear in this video). Examples of this behavior can be observed at timestamps 0:52 and 0:56, where the RPM jumps up, and the gear position indicator (in the upper-middle part of the dashboard’s display) switches into 3, and 2, accordingly.

Note that during this entire drive, the gear selector was fixed in the Drive position, and that the driver manually activated the hazard lights for safety reasons.

The *SMManager* stops interfering with the gear operation when the vehicle reaches the desired safe speed of 40Km/h (timestamp 1:16), letting the driver ‘limp back home’, or as advised by the *SMManager*—try to “find a safe place to stop and reset the vehicle”, under the assumption that a proper reset could help end the attack (e.g., non-volatile components were not effected). The video ends when the driver stops the vehicle.

Testing with a ‘Partially-Cooperative Driver’

The second video [12] captures a more complex scenario, in which the driver tries to accelerate after the intrusion is detected, making the *SMManager* choose different tactics. To show this, we captured the screen of a secondary laptop (connected through the OBD-II port of the vehicle), running a VCDS diagnostic software [34] tracking the RPM and Gas Pedal levels. The two graphs can be seen from timestamp 0:35 at the bottom right corner of the video.

On this drive, we manually triggered the intrusion detection at timestamp 0:54 (vehicle at 70kph, D6). This time, the driver ignored the manager’s initial request and continued to apply the gas pedal for the first 18 seconds (from timestamp 0:54 to 1:12), causing the *SMManager* to issue a voice notification of “Please do not accelerate” (timestamp 1:06). During this time the *SMManager* also tried to slow down the vehicle by switching the gear to the Neutral position, to mitigate the driver’s acceleration (recall line 15 in Algorithm 1).

When the driver eventually leaves the gas pedal (timestamp 1:12), the *SMManager* switches to the action observed in the first video [11], shifting the gear down toward the chosen maxSafeGear of one. This driver action can be observed in the overlaying video, when the green line of the gas pedal graph levels with its idle position marked line. We observe the gear down-shifts at timestamps 1:14 and 1:21: the RPM gauge jumps up, and the engine noise increases.

The vehicle reaches the chosen safe speed of 40km/h a few seconds later. From this point onward, the *SMManager* does not try to manipulate the gear any more, since the vehicle is driving below the chosen safe speed limit. An example of that can be seen after timestamp 1:23, with a normal car action of up-shifting from D2 to D3, leaving the driver with the ability to ‘limp back home’, or stop for a reset as requested by the *SMManager*.

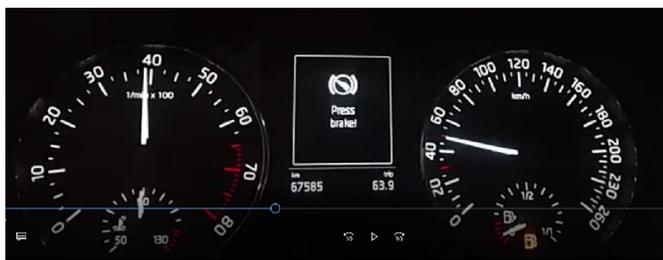


Fig. 4. The driver notification alert triggered by a transmission of a pre-existing CAN message: picture taken from one of our test drive videos [10].

VII. CONCLUSION

In this paper we demonstrated that *vehicle safe-mode* can be implemented as an after-market add-on, by developing a proof-of-concept system, and actively testing it on an operating Skoda

Octavia vehicle. Once activated, our VSM system restricts the vehicle to *Limp-mode* behavior by guiding it to remain in low gear, taking into account the vehicle's speed and the driver's actions. The system overrides some of the normal gear-shifting logic by careful manipulation of the relevant CAN bus messages. Our system does not require any changes to the ECUs, or to any other part of the vehicle, beyond connecting the *safe-mode* manager to the correct bus.

As part of the VSM development we analyzed the Skoda's CAN bus topology, and discovered that it is possible to connect our system to the Powertrain bus from inside the passenger compartment, without opening the hood. We reverse-engineered some of the vehicle's CAN messages and their timing characteristics, which allow the *safe-mode* manager to identify the vehicle's condition and construct the overriding control messages. We implemented the *safe-mode* manager to work in real-time when connected to the vehicle's CAN bus, and tested it in multiple driving scenarios, taking the VSM-augmented vehicle onto the roads and successfully demonstrating its functionality.

We believe that the VSM concept, using ideas from vehicle safety such as the *Limp-mode* in conjunction with cyber-defense ideas of intrusion detection and prevention, is a strong combination. In a safety oriented automotive domain, any type of reaction to a cyber-incident must balance safety considerations with the attack severity. Our proof-of-concept demonstrates that such a balanced combination is realistic, and can be built even without the cooperation of the vehicle or ECU manufacturers. However, if ECU manufacturers incorporate "VSM-ready" capabilities, and vehicle manufacturers include the VSM logic when integrating intrusion- or anomaly-detection technologies, much better solutions can be developed.

REFERENCES

- [1] AnthonyJ350, "Nissan 350z / infiniti g35 ecu reset (check engine light)," <https://www.youtube.com/watch?v=5tQm28K32SQ>, 2016.
- [2] AUTOSAR, "AUTOSAR secure onboard communication (SecOC), version 4.3," <https://www.autosar.org/standards/classic-platform>, 2016.
- [3] Bosch mobility solutions, 2018. [Online]. Available: <http://www.bosch-mobility-solutions.com/en>
- [4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *Proceedings of the 20th USENIX Conference on Security (SEC'11)*, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2028067.2028073>
- [5] K.-T. Cho and K. G. Shin, "Viden: Attacker identification on in-vehicle networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1109–1123.
- [6] W. Choi, K. Joo, H. J. Jo, M. C. Park, and D. H. Lee, "Voltageids: Low-level communication characteristics for automotive intrusion detection system," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 2114–2129, 2018.
- [7] T. Dagan and A. Wool, "Testing the boundaries of the Parrot anti-spoofing defense system," in *5th Embedded Security in Cars (ESCAR USA'17)*, Ypsilanti, MI, USA, Jun. 2017.
- [8] —, "Woodpecker, a software-only true random generator for the CAN bus," in *16th Embedded Security in Cars (ESCAR'18)*, Brussels, Nov. 2018.
- [9] T. Dagan, M. Marchetti, D. Stabili, M. Colajanni, and A. Wool, "Vehicle safe-mode limp-mode in the service of cyber security," in *15th Int. Conf. on Embedded Security in Cars (ESCAR 2017)*, Berlin, Germany, Nov. 2017.
- [10] T. Dagan and Y. Montvelisky, "VSM PoC Video," 2018. [Online]. Available: <https://youtu.be/S7g8iGSA50E>
- [11] —, "VSM PoC Video V2.13," 2019. [Online]. Available: <https://youtu.be/ihGR4jweeF8>
- [12] —, "VSM PoC Video V2.15," 2019. [Online]. Available: <https://youtu.be/5RYX9afZ34U>
- [13] T. Dagan and A. Wool, "Parrot, a software-only anti-spoofing defense system for the CAN bus," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2016)*, Munich, Germany, Nov. 2016.
- [14] I. Foster and K. Koscher, "Exploring controller area networks," *USENIX ;Login: magazine*, vol. 40, no. 6, 2015.
- [15] B. Glas and M. Lewis, "Approaches to economic secure automotive sensor communication in constrained environments," in *11th Int. Conf. on Embedded Security in Cars (ESCAR 2013)*, 2013.
- [16] A. Greenberg, "After Jeep hack, Chrysler recalls 1.4m vehicles for bug fix," 2015. [Online]. Available: <http://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/>
- [17] —, "Hackers remotely kill a Jeep on the highway with me in it," 2015.
- [18] Hagens Berman Sobol Shapiro LLP, "Ford shelby gt350 mustang overheating," <https://www.hbsslaw.com/cases/ford-shelby-gt-mustang-overheating>, 2005.
- [19] Y. Hamada, M. Inoue, S. Horiata, and A. Kamemura, "Intrusion detection by density estimation of reception cycle periods for in-vehicle networks: A proposal," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2016)*, Munich, Germany, Nov. 2016.
- [20] T. Hoppe, S. Kiltz, and J. Dittmann, "Adaptive dynamic reaction to automotive IT security incidents using multimedia car environment," in *2008 The Fourth International Conference on Information Assurance and Security*, Sept 2008, pp. 295–298.
- [21] Infineon Technologies AG, "Driving the future of automotive electronics, automotive application guide," 2014. [Online]. Available: <https://www.infineon.com/dgdl/AutomotiveApplicationGuide.pdf?fileId=db3a30431ed1d7b2011edf68bea53e77>
- [22] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy (SP)*, May 2010, pp. 447–462.
- [23] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horiata, "CaCAN—centralized authentication system in CAN (controller area network)," in *12th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.
- [24] R. Kurachi, H. Takada, T. Mizutani, H. Ueda, and S. Horiata, "SecGW secure gateway for in-vehicle networks," in *13th Int. Conf. on Embedded Security in Cars (ESCAR 2015)*, 2015.
- [25] H. Lee, S. H. Jeong, and H. K. Kim, "Otdis: A novel intrusion detection system for in-vehicle network by using remote frame," in *PST*, 2017.
- [26] M. Marchetti and D. Stabili, "Anomaly detection of can bus messages through analysis of id sequences," in *28th IEEE Intelligent Vehicle Symposium (IV2017)*, June 2017, pp. 1–6.
- [27] M. Marchetti, D. Stabili, A. Guido, and M. Colajanni, "Evaluation of anomaly detection for in-vehicle networks through information-theoretic algorithms," in *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, Sept 2016, pp. 1–6.
- [28] M. Markovitz and A. Wool, "Field classification, modeling and anomaly detection in unknown CAN bus networks," *Vehicular Communications*, vol. 9, pp. 43–52, 2017.
- [29] T. Matsumoto, M. Hata, M. Tanabe, K. Yoshioka, and K. Oishi, "A method of preventing unauthorized data transmission in controller area network," in *IEEE Vehicular Technology Conference (VTC Spring)*. IEEE, 2012, pp. 1–5.
- [30] C. Miller and C. Valasek, "Adventures in automotive networks and control units," 2014. [Online]. Available: http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf
- [31] NXP automotive, 2018. [Online]. Available: <http://www.nxp.com/applications/automotive>
- [32] D. Pauli, "Hackers hijack Tesla Model S from afar, while the cars are moving," 2016. [Online]. Available: http://theregister.co.uk/2016/09/20/tesla_model_s_hijacked_remotely
- [33] PEAK-System, "PCAN-USB: CAN interface for USB," 2015. [Online]. Available: http://www.peak-system.com/produktcd/Pdf/English/PCAN-USB_UserMan_eng.pdf
- [34] Ross-Tech, "VCDS: Windows-based Diagnostic Software for VW / Audi / Seat / Skoda," 2019. [Online]. Available: <https://www.ross-tech.com/vag-com/>

- [35] Texas Instruments Incorporated, "DRV8305-Q1 three-phase automotive smart gate driver with three integrated current shunt amplifiers and voltage regulator," <http://www.ti.com/lit/ds/symlink/drv8305-q1.pdf>, 2016.
- [36] Transmission Repair Cost Guide, "What is limp mode? causes and what to do," <https://www.transmissionrepaircostguide.com/limp-mode/>, 2015.
- [37] Y. Ujii, T. Kishikawa, T. Haga, H. Matsushima, T. Wakabayashi, M. Tanabe, Y. Kitamura, and J. Anzai, "A method for disabling malicious CAN messages by using a centralized monitoring and interceptor ECU," in *13th Int. Conf. on Embedded Security in Cars (ESCAR 2015)*, 2015.
- [38] UN-ECE, "UN-ECE Resolution on the Construction of Vehicles (RE.3) ECE/TRANS/WP.29/78/Rev.6 Annex 6 (4.3.3)," July 2017. [Online]. Available: <https://www.unece.org/trans/main/wp29/wp29wgs/wp29gen/wp29resolutions.htm>
- [39] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus," in *ECRYPT Workshop on Lightweight Cryptography*, vol. 2011, 2011.
- [40] T. van Roermund, A. Birnie, R. Moran, and J. Frank, "Securing the in-vehicle network of the connected car," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2016)*, Munich, Germany, Nov. 2016.