Automated Fuzzing of Automotive Control Units

Timothy Werquin, Roos Hubrechtsen, Ashok Thangarajan, Frank Piessens and Jan Tobias Mühlberg
KU Leuven, Dept. of Computer Science, imec-DistriNet
B-3001 Leuven, Belgium
⟨firstname.lastname⟩@cs.kuleuven.be

***Abstract*—Modern vehicles are governed by a network of Electronic Control Units (ECUs), which are programmed to sense inputs from the driver and the environment, to process these inputs, and to control actuators that, e.g., regulate the engine or even control the steering system. ECUs within a vehicle communicate via automotive bus systems such as the Controller Area Network (CAN), and beyond the vehicles boundaries through upcoming vehicle-to-vehicle and vehicle-to-infrastructure channels. Approaches to manipulate the communication between ECUs for the purpose of security testing and reverse-engineering of vehicular functions have been presented in the past, all of which struggle with automating the detection of system change in response to message injection. In this paper we present our findings with fuzzing CAN networks, in particular while observing individual ECUs with a sensor harness. The harness detects physical responses, which we then use in a oracle functions to inform the fuzzing process. We systematically define fuzzers, fuzzing configurations and oracle functions for testing ECUs. We evaluate our approach based on case studies of commercial instrument clusters and with an experimental framework for CAN authentication. Our results show that the approach is capable of identifying interesting ECU states with a high level of automation. Our approach is applicable in distributed cyber-physical systems beyond automotive computing.**

***Index Terms*—automotive control networks, CAN, security, testing, fuzzing**

I. Introduction

Modern cars are largely controlled by software. This software forms a distributed mixed-criticality system that executes on a number of interconnected Electronic Control Units (ECUs). Jointly, these ECUs govern the vehicle's behaviour – from convenience and infotainment functions to safety-critical functionality. ECUs are connected via automotive bus systems that facilitate the exchange of messages, most of which communicate sensor readings and control instructions. The control software then interprets these messages and reacts to events by triggering the relevant actuators, e.g., brakes, airbags, or steering gear. In 2016, the Ford Motor Company reported that their latest models are running on 150 million lines of code. Given the enormous complexity of these systems, they are notoriously hard to test, for safety as well as for security properties.

Since around 2004, researchers have been expressing their concerns with respect to the security limitations of communication standards, including the widely used Controller Area Network (CAN) [1]–[3]. Since 2010, a series of high-profile attacks [4]–[7] illustrate that with increased vehicular connectivity even remote adversaries can take control of critical functions of a vehicle. These risks have been acknowledged and are partly addressed in emerging industry standards [8], [9] that encompass authentication and software security for control systems, and prototypes that showcase secure system designs for automotive computing based on software attestation and Trusted Computing primitives [10] have been proposed. Meanwhile, more and more low-level vulnerabilities in these communication systems are being revealed (e.g., [11] and [12]), guidance for the reverse-engineering and penetration testing of vehicular communications and control systems becomes readily available [13], and the need for advanced testing methodology for these systems is generally acknowledged. A testing approach that promises a particularly high level of automation is fuzzing.

Fuzz testing [14], [15] is a well established methodology to expose software and systems to unexpected conditions, for example by providing random input streams that may crash the target. Yet, the approach does not easily apply to embedded software [16] and few approaches have been made to fuzz embedded control systems or automotive ECUs in particular [5], [17], [18]. A key difficulty to overcome here is the definition of oracle functions that define when a fuzzer has potentially triggered a bug or at least an "interesting" system state, and to automatically evaluate these functions.

1) Our Contributions.: In this paper we discuss fuzz testing in the context of automotive control networks. Specifically, our research investigates the use and automation of fuzzing so as to find vulnerabilities and to reverse engineer ECU functionality in CAN networks. We make the following contributions:

1) We systematically define fuzzers, fuzzing configurations and oracle functions for testing automotive ECUs through their CAN interface.
2) We develop a sensor harness to automatically evaluate fuzzing oracles for ECUs with physical outputs.
3) We evaluate our approach, taking commercial automotive instrument clusters and an experimental setup for testing AUTOSAR-compliant message authentication as case studies.

To the best of our knowledge, this paper is the first to largely automate a methodical fuzzing approach (e.g. following [15]) for automotive ECUs. Although our implementation is targeting CAN components, our approach can be generalised to cyber-physical systems with any underlying communication technology. Our fuzzer implementation, instructions to build the sensor harness and to repeat our experiments are available under an open-source license at https://github.com/timower/caringcaribou/tree/autoFuzz.

II. Background

In this section we briefly introduce the CAN bus, which is commonly used to facilitate communication between automotive ECUs but also in industrial control systems. We further introduce the CaringCaribou penetration testing framework, which our fuzzing toolchain is integrated with.

A. Controller Area Network (CAN) & Security

The CAN bus is the most commonly used broadcast network in modern cars. A CAN message consists of an 11-bit arbitration ID, followed by an optional 18-bit extended ID, and up to 8 bytes of data payload (cf. Fig. 1). Dedicated transceiver hardware implements a protocol for message acknowledgement and bus arbitration for sending/receiving data frames. CAN requires a fixed data transmission rate, and allows recessive bits (one) to be overwritten by dominant bits (zero) during transmission. Message acknowledgement can thus simply be implemented by overwriting the ACK bit at the end of the data frame in real-time. Likewise, to implement bus arbitration, CAN transceivers are required to continuously listen on the bus while sending the message ID at the beginning of the data frame, and to back off when one of their ID bits has been overwritten. This scheme ensures that messages with lower IDs effectively have higher priorities. Finally, each CAN frame features a 16-bit CRC field to detect transmission errors.

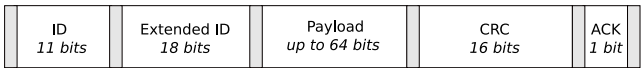

Figure 1. Extended data frame standardised by CAN 2.0B.

CAN was originally developed in 1983, when cyber attacks were of no concern. Thus, the protocol does not provide any form of message authentication. Any ECU connected to the network can spoof messages with arbitrary sender ID and payload, which forms the basis of many attacks [4]–[7]. As a response, the AUTOSAR [9] standardisation body published industry guidelines for backwards-compatible message authentication in vehicular networks. The VulCAN framework [10], which we study in Sect. V-A is one implementation of these authentication extensions.

B. CaringCaribou and Automotive Penetration Testing

We built our implementations of CAN fuzzers as modules for the open-source penetration-testing framework CaringCaribou. Our fuzzing extensions are freely available for further experimentation and follow-up research. Since fuzzers are widely used as a means to perform black-box testing in the regular penetration testing industry, it is our belief that a similar tool could prove useful in the automotive penetration testing community.

CaringCaribou[1] is a tool developed for the purpose of being the "nmap" of automotive security. Cyber security research in the automotive industry is a new field that is rapidly expanding. Yet, it still lacks the mature tooling available to the mainstream security community. Tools like CaringCaribou aim to fill that gap. CaringCaribou has a modular architecture that allows developers of penetration testing techniques for automotive systems to easily write new modules for their specific purpose, and deploy these modules using a unified tooling infrastructure. Thus, CaringCaribou provides the developer with a layer of abstraction which protects them from the specifics of CAN and other automotive communication protocols, allowing them to focus on writing the actual penetration testing tool instead of dealing with the lower-layer interactions. CaringCaribou is supposed to be a zero-knowledge tool that can be deployed on any CAN network regardless of its specific configuration.

[1]https://github.com/CaringCaribou/caringcaribou

III. Fuzzing CAN Networks

In this section we lay out our approach to define a fuzzing tool for CAN-based automotive control systems. We follow the approach of Manes et al. [15] and dissect the tool into an oracle component, the actual fuzzer and the run-time configuration for the fuzzer for a particular run.

A. Bug Oracles for ECUs

According to [15], a (bug) oracle is a program that determines whether a given execution of the target system violates a specific (security) policy. In Sect. V we outline two very different case studies for our system: In one of these we have a partial specification of security properties of the network available, and where we are looking for violations of this specification. In the second case study we have no reliable specification but we are interested in reverse-engineering such a specification. Both case studies are characterised by not being able to observe software interactions directly (as opposed to software fuzzing with code instrumentation in a simulator [15]). Instead, we are looking at black-box systems to which our fuzzer can provide an input stream, while any observable communication output, physical output (actuation of a LED or a relay) or even the timing or absence of such outputs (e.g., due to a software crash) may indicate that an "interesting" system state has been reached.

Recent related work in the field of intrusion detection for industrial control system (e.g., [19] and [20]) suggests that machine-learning approaches can be used to train detectors that then report anomalies in the communication behaviour of vehicular networks. We have not implemented such oracle functions.

Physical outputs of control units can certainly be observed by human operators. They can also be sensed and electronically reported through sensor networks, or in our case a sensor harness that is attached to the target system. In the following sections we emphasise on this form fuzzing oracle, where a state change in the target is defined by a sensor (de-)activation or sensor threshold.

Most difficult is certainly the detection of system failure which results in the absence of an observable response from the target. Thus, inputs that lead to failures are easily misinterpreted by a fuzzing tool as inputs that have no effect. For example, our work deals with ECUs that need to regularly receive

certain messages or they will fault, effectively rendering the continuation of a fuzzing campaign ineffective. Detecting these messages and constructing traffic that satisfies the input requirements of ECUs in the absence of a specification, is difficult. An oracle to identify these kinds of system faults requires the use of recorded traffic that periodically triggers a known observable output. An oracle function will then fire when the periodic signal is absent due to, e.g., message omission. We apply this method in our *omission fuzzer* below.

B. Defining CAN Fuzzers

Fuzzing is the execution of the target system using input(s) sampled from an input space (the "fuzz input space") that protrudes the expected input space of the target system [15]. With fuzzing we aim to enumerate and exercise a large subset of this fuzz input space to find system behaviour that triggers an oracle function. ECUs that process CAN messages are an interesting target since the frame size of CAN messages is at most 110 bits. This fuzz input space is certainly huge, but much smaller than, e.g., Ethernet frames, WiFi frames, or multimedia streams. Still, even for CAN networks, this fuzz input space is prohibitively large for being exhaustively exercised. Furthermore, with a maximum bandwidth of 1 MBit, and most ECUs using 500 MBit as a fixed transfer rate, data transmission to a target network of ECUs represents a bottleneck.

Starting with the idea of *random fuzzing*, where arbitration IDs and message payloads are selected randomly, we devise three additional fuzzing strategies, *brute-force fuzzing*, *mutation fuzzing* and *identify fuzzing*, to narrow down the fuzz input space and explore interesting ECU behaviour more efficiently. These strategies are based on the observation that an ECU typically accepts inputs on a relatively small number of IDs only, that also the number of payload bits that result in a observable state change is limited, and that several consecutive messages may be required to trigger an observable state change. We then integrate these approaches in an *automated exploration* mode, where inputs from a sensor harness (cf. Sect. IV), which is attached to a target ECU, guide input generation. We have implemented our approach in two modules for CaringCaribou, namely `fuzzer` and `autoFuzz`, which can be invoked as `./cc.py <module> <parameters> [-f <file>]`. Here, `./cc.py` refers to the CaringCaribou main script, `<module>` to a fuzzer module, and `<parameters>` to a fuzzer configuration which we discuss below. `-f <file>` can be used to store a message trail on disk. For example, `./cc.py fuzzer random` will generate entirely random messages and dispatch them over the configured CAN interface.

1) Brute-Force Fuzzing.: This method aims to exhaustively enumerate selected hexadecimal digits in a message, specifically in the message's ID field and the payload. For example, the fuzzer can be invoked as `./cc.py fuzzer brute 0x123 12ab..78`, where the 5th and 6th octet of the message payload will be enumerated and sent, while the message ID `0x123` and all other payload octets remain constant.

2) Mutation Fuzzing.: This strategy can be used to systematically explore a larger fuzz input space through mutating selected hex digits in arbitration ID and message by means of individual random bit flips. An example use for this strategy is `./cc.py fuzzer mutate 7f.. 12ab....`; the syntax follows the example given for *brute-force fuzzing* above.

3) Identify Fuzzing.: Once a fuzzing run resulted in an event of interest, e.g., a change of an indicator LED on a target ECU, the *identify* method can be used to replay and identify a minimal set of messages that caused the event. The syntax for invoking this method is `./cc.py fuzzer identify log.txt`, where `log.txt` refers to a log file previously recorded with the `-f` parameter. The method relies on human input – i.e., key presses – to gather information about the timely occurrence of events, and aims to prune the set of recorded messages in `log.txt` so that the event still occurs when the pruned set is replayed.

4) Automated ECU Exploration.: Our `autoFuzz` module implements the above strategies so that system change can be detected directly through our sensor harness (cf. Sect. IV). Sensor observations can then be used to guide the generation of the next inputs and to automatically identify message bits that lead to observable system change, depending on the fuzzing strategy. The module further features the generation of J1939-compliant messages and the fuzzing of J1939 function group addresses (PGNs and SPNs).

When fuzzing an ECU with a single sensor attached to one of the ECU's actuators, it is possible to immediately run the identify fuzzer when a change in the sensor state is detected without relying on recorded traffic. This requires that the actuator can be triggered with a predictable payload. When using multiple sensors, the log file can be filtered to keep a number of messages preceding the activation of a specific sensor which can then be used as input to the identify fuzzer.

While experimenting with fuzzing strategies, we observed that an ECU's response to a message is often delayed. During the delay period, other messages are being sent by the fuzzer, which makes identifying the CAN messages specifically responsible for an response more difficult. One possible solution is to increase the delay between sending messages. Yet, this will increase the time required to cover the fuzz input space. Another option is the resend only a subset of the messages preceding a sensor activation with increased delays, which we implemented in our identify method.

Our experiments further revealed that some ECUs expect certain messages to be received regularly. The absence of these messages will lead to a shut-down or render the ECU unresponsive and to indicate a failure. These behaviours prevent our identify method from working as sending the complete traffic log can keep the ECU responsive but sending parts of the recorded traffic will cause the ECU to fault. To address this, we developed an approach that we refer to as *omission fuzzing*. This strategy sends the complete recorded traffic but omits some messages in order to identify which message cause

specific state changes. The identified arbitration IDs or payloads can then be added to a "blacklist" to inform other methods. E.g., any arbitration ID in the blacklist will never be omitted during *identify fuzzing*.

After collecting logs from the fuzzer, various analysis can be applied. For example, if a specification of the target ECU is available, which would detail the expected actuator responses to specific groups of messages, sensor activations can be check against this specification to detect bugs or undocumented behaviour.

C. Target-Specific Fuzzer Configuration

A fuzz configuration of a fuzzer comprises the parameter value(s) that control(s) the fuzz algorithm [15]. In the context of our approach, these parameters involve *message generation*, *message timing*, *message omission*, and the configuration of the *sensor harness*. As outlined before fuzzing entire CAN messages makes little sense as it results in an extremely large fuzz input space. Thus, configurations will typically restrict the fuzz space to specific octets in (extended) arbitration IDs and message payload. Message timing is typically configured to schedule a new message every 3 ms to 20 ms to avoid message collisions and to leave enough time for actuators to be engaged and sensed. Message omission and baseline traffic are to be set up to simulate typical bus traffic in a car so as to make target ECUs function normally. The sensor harness offers a wide range of configuration options that involve the type of sensors, sampling rates, the number of sensors and their placement on the target ECU.

IV. A Sensor Harness to Automate ECU Fuzzing

In this section we describe an inexpensive and extensible experimental sensor harness to automate the analysis of automotive ECUs. The intuition behind the setup is that fuzzing communication in an automotive control network, or in cyber-physical systems in general, can cause a range of interesting responses beyond network communication. Thus, to use these responses as inputs to fuzzing oracles (cf. Sect. III-A) they must be automatically measured at an appropriate sampling rate. Previous approaches to consider these responses typically rely on human observation and human interaction during the fuzzing process. For example, a fuzzing tool may require the user to press a key if they observe a change in the system, e.g., a flashing indicator light on a control panel. Our work improves over this by detecting physical responses of ECUs automatically, with negligible delays, and at a configurable granularity.

Fig. 2 gives an overview of the sensor harness. The system in action is depicted in Sect. V-B. The harness connects multiple sensors together and provides a Universal Serial Bus (USB) interface for a PC to control the setup. The depicted configuration contains only light and colour sensors which can be placed over the various indicators LEDs on a target ECU. No general-purpose microcontroller is used in the harness, which allows the entire setup to be programmed and configured from the PC in a higher-level language, Python in our case.

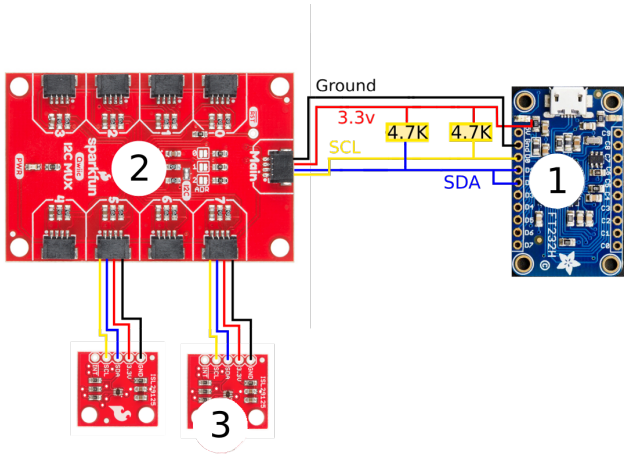

Figure 2. Sensor harness connection schema showing *(1)* the FT232H I2C-to-USB converter, *(2)* an TCA9548A I2C Multiplexer, and *(3)* two ISL29125 light & colour sensors.

The light sensors are connected through an I2C bus, a communication interface which is present on many low-cost sensors. The use of I2C make the harness extensible with various other sensors, such as sound sensors to monitor auditory alerts, motion sensors to monitor steering wheel movement, or current sensors to detect an engine start. As the I2C bus supports multiplexing, multiple sensors can be connected to the same bus as long as they have a different I2C address. The low-cost sensors used in the harness often have a fixed address, which implies that an I2C multiplexer must be used to connect multiple sensors of the same type in the harness. In order to interface with the sensor harness from a PC an USB to I2C adapter is used.

1) Light & Colour Sensors: ISL29125.: The ISL29125 colour sensor is used to measure the status of visual indicators on automotive equipment. The sensor provides a simple RGB light level readout and has a number of configuration options which determine the sensitivity and precision of the sensor. The sensitivity can be configured between 10k lux or 375 lux and the sensor has a built in infrared light filter which can be configured separately. The precision of the sensor can be configured to be either 12 or 16 bits. Changing the precision also changes the integration time, meaning a higher precision (16 bits) will require the Analogue-Digital Converter (ADC) to sample the sensor longer resulting in slower measurements. In addition to the I2C interface the sensor has an interrupt pin which can be triggered by a configurable light level on either of the red, green or blue channel. Currently the harness does not use the interrupt functionality but works by polling each sensor individually. Our colour sensors have a single fixed I2C address, requiring a multiplexer to connect multiple sensors on the same I2C bus.

2) I2C Multiplexer: TCA9548A.: As the colour sensor has a fixed I2C address, the TCA9548A multiplexer is used to connect multiple colour sensors in the harness. The TCA9548A multiplexer has eight I2C channels which allows eight colour sensors to be connected on the same bus. As the multiplexer

has a three bit configurable address multiple multiplexers can be daisy chained allowing up to 64 I2C busses on a single connection.

3) I2C-to-USB: FT232H.: In order to connect the I2C bus to a PC the FTDI FT232H adapter is used. This adapter supports a number of different bus protocols such as UART, SPI and I2C in addition to a GPIO interface which can be used to write to eight digital IO pins.

4) Programming, Calibration & Use.: To obtain readings from the light sensors, the Arduino library[2] for reading ISL29125 sensors was adapted. The resulting library uses the Adafruit[3] library for communicating with the FT232H adapter. Our current sensor library exposes functions for initialising a new sensor, and for reading the sensor's red, green and blue values.

The initialisation function resets the sensor and configures it in RGB mode which enables all three colour channels, puts it in 10k-lux mode for bright environments and enables high IR light adjustment. Reading a colour value is done by reading from the relevant device register, which returns a colour value of either 12 or 16 bits depending on the chosen precision. The time the sensor needs to take a reading varies depending on the ADC integration time which in turn depends on the chosen precision: At 16 bit precision, each reading takes about 110 ms while at 12 bit each reading takes only 7 ms. As the sensor's output registers are double-buffered, reading out these registers between sensing operations will result in outdated readings.

In order to use multiple light sensors a library that interfaces with the TCA9548A multiplexer was created, this library allows virtual I2C ports to be created for each channel of the multiplexer. These virtual I2C ports can then be used in the sensor library instead of the default FT232H I2C port. In order to switch I2C channels the number of the requested channel is written before any commands, this adds a delay before every I2C command, which is negligible in comparison with the integration time of the light sensor and does not impact fuzzing performance.

In initial experiments we use the colour light sensors to detect whether the various indicators on an automotive dashboard are changing state, effectively converting the red, green and blue light levels into a binary input signal for the fuzzer. As the sensors are sensitive enough to detect (even reflected) movements behind the sensor while duct-taped to a dashboard in both sensitivity configurations (up to 375 lux and 10k lux) a simple threshold is not sufficient to distinguish state change. We devise a calibration method that involves taking a reading when the indicator is on and when the indicator is off. This results in red, green and blue light level triples to which any new measurement can be compared, if the new measurement is closer to the on-value the indicator is detected as on and vice versa. This method assures that a uniform increase or decrease in ambient light does not change the detected indicator value. The method further requires a calibration with the indicator both on and off, which may not be feasible when the indicator trigger is unknown. When the indicator cannot be triggered during calibration, a simple threshold may be used to detect the indicator state. More elaborate calibration methods may be required to operate the sensor harness in noisy environments.

[2]https://github.com/sparkfun/SparkFun_ISL29125_Breakout_Arduino_Library

[3]https://github.com/adafruit/Adafruit-FT232H-Breakout-PCB

V. Evaluation and Discussion

We have applied our fuzzer implementation and the sensor harness to a number of case studies that include the ICSim automotive instrument cluster simulator[4], a demo setup for illustrating and implementing message authentication in CAN networks with the VulCAN [10] framework, as well as real instrument clusters. In this section we focus on our experience and lessons learned from the latter two case studies. We compare our findings with earlier manual approaches to discover bugs and explore proprietary functionality in these scenarios.

A. Case Study 1: VulCAN

We evaluated the effectiveness of our fuzzer to find implementation bugs and security vulnerabilities on a demo implementation of VulCAN [10], a generic design for CAN message authentication. VulCAN provides efficient and AUTOSAR-compliant [9] authentication plus software component attestation based on lightweight trusted computing technology. We used the same test bench as described in [10] to test the abilities of the fuzzer.

In brief, the demo consists of a number of ECUs with keypads as input devices and LED displays as actuators. A distributed control application which simulates a traction control system is executing on the ECUs. Application components communicate via cryptographically authenticated CAN messages with freshness guarantees: only messages that are successfully validated to be fresh and to originate from unmodified and integrity-protected remote component should ever be able to trigger output events. The application communicates only a few valid payloads at fixed intervals. Thus, deviation from expected behaviour would be easy to detect. Yet, since it is unlikely for a random or mutation-based fuzzer to "guess" a valid payload, nonce, and authentication tag triple, and since the system was designed with security in mind, we did not expect the security properties of the system to be broken easily. This part of the evaluation is conducted without using the sensor harness but by relying on visual observation on the demo's LED displays. The fuzzer is executing on desktop PC, which is connected to the demo setup via a USB to CAN interface.

To our surprise, with the help of the fuzzer, we detected and traced several unique vulnerabilities in the system in a fairly short period of time. Below we focus on two particularly subtle discoveries.

The first vulnerability was discovered nearly instantaneously in a fuzzer configuration where messages with extended CAN arbitration IDs are generated. Such messages resulted in system

[4]https://github.com/zombieCraig/ICSim

states where the injected messages could lead to actual display outputs, breaking the security properties of VulCAN entirely. Extended CAN IDs are not being used in the VulCAN demo, and thus, the components were not tested in environments where these messages occur. Most likely, a misconfigured driver for the CAN controller on an ECUs – "untrusted" software in VulCAN's attacker model – together with an incomplete rejection condition in a secure application module, allowed an attacker to arbitrarily adjust the displays of the test bench without having to pass authenticity checks.

The second vulnerability was found within the implementation of one of the authentication protocols in VulCAN, specifically VatiCAN [21]. This implementation turned out to be particularly vulnerable to denial-of-service attacks when being flooded with specific traffic patterns, allowing an attacker to desynchronise nonces and render trusted components unresponsive even to dedicated re-synchronisation messages. The bug was discovered in a timespan of several minutes when fuzzing the test bench in a configuration where both, the fuzzer as well as an ECU, are simultaneously attempting to send messages to a target ECU. Interestingly, due to the configuration error in CAN drivers described above, messages with extended CAN IDs are effectively interpreted as broadcast messages. Application components are thus subject to receiving a mix of fuzzer-generated payloads and authenticated messages which results in denial-of-service.

B. Case Study 2: Instrument Clusters

In the context of automotive security research and for building demos such as the VulCAN [10] setup, instrument clusters are commonly used as easily accessible off-the-shelf components with many visible indicators (speed needle, turning indicators, display, etc.), most of which can be controlled through CAN messages. Yet, the specific arbitration IDs and payloads to control these functions are not publicly documented. Literature on car hacking (e.g., [13]) suggests manual approaches to reverse-engineer these details, which may require hours or even days of try-and-error, even for a skilled engineer. By using our sensor harness, we expect a substantial speed-up of these processes, on top of being able to largely automate the process.

We have been experimenting with a number of clusters from passenger cars and commercial vehicles. As illustrated in Fig. 3, components of our sensor harness are duct-taped to indicators of the cluster. The instrument cluster is connected to a desktop PC with the fuzzer via a USB to CAN interface and there are no other ECUs present on the CAN.

In order to test the instrument cluster's basic functions, we developed a controller application that would send a number of documented [22] CAN messages to the dashboard. Only some these control messages worked in combination with our dashboard. We then applied our fuzzer in *identify* mode to filter the traffic data for messages that trigger physical functions, then applied *brute-force* and *mutation* mode to explore arbitration IDs and payloads to trigger further functionality.

By *brute-forcing* the entire 11-bit arbitration ID fuzz-space with a fixed payload `0xffffffff`, most indicators LEDs in the dashboard could be activated. Some of these indicators are switched on by default when the instrument cluster is powered up. These indicators could be triggered using a fixed payload of `0x00000000`. Control of the speedometer and engine RPM needles could also be triggered. The fuzzer takes about 30 s to enumerate the 11-bit address space using a delay of 10 ms between messages. Running the *identify* method after finding a state changes takes an extra 30 s per activation in order to identify the message responsible for an indicator activation.

Using *mutation-based* fuzzing on any of the messages identified with the brute-force method, we were able to reverse engineer the semantics of most payload bits. For example, by starting from the messages that activated the left turning indicator, the fuzzer was able to not only identify the bit responsible for triggering the indicator. It was also able to identify the bit responsible for the right turn signal, the headlights indicator and a number of other status LEDs. Enumerating eight payload bits takes about 20 s with a delay of 3 ms between messages. Using the *identify* method requires an additional 5 s per indicator. The delay between messages is critical when using mutation-based fuzzing. If the delay is too short, the next message, which will have another bit flipped, will overwrite the previous message. This will cause more indicator activations to be missed or some indicators to not even activate.

Omission fuzzing is not necessary for analysing this particular instrument cluster. Yet, we heavily relied on this method method when working with more modern clusters from commercial vehicles.

C. Discussion & Lessons Learned

The two case studies outlined above show that our fuzzers can efficiently reveal undocumented functionality, intricate bugs and security vulnerabilities in ECUs connected to CAN networks. Ultimately, our experiments provide further evidence for fuzzing to be a useful tool in testing and reverse-engineering, which is due to the technique's ability to cover an enormous range of possible combinations of system inputs, in our case arbitration IDs and payloads. Many of these inputs may not even occur in normal and benign operation, and are difficult to consider in static test cases.

Above we describe two critical security vulnerabilities in an experimental system design, which are based on intricate implementation and configuration bugs. Previously undetected, these vulnerabilities became apparent within minutes with the use of a fuzzer, even without relying on a sensor harness. The harness could be used to reduce human interaction and to improve the duration for detecting and tracing these bugs, but non-trivial extensions of the harness would be required to sense the state of actuators such as the attached displays. Alternatively, the demo setup could be modified to feature simpler actuators (i.e., individual LEDs or relays) that allow for an easier detection of conditions that satisfy our bug oracles. We used our fuzzer, specifically the identify and replay

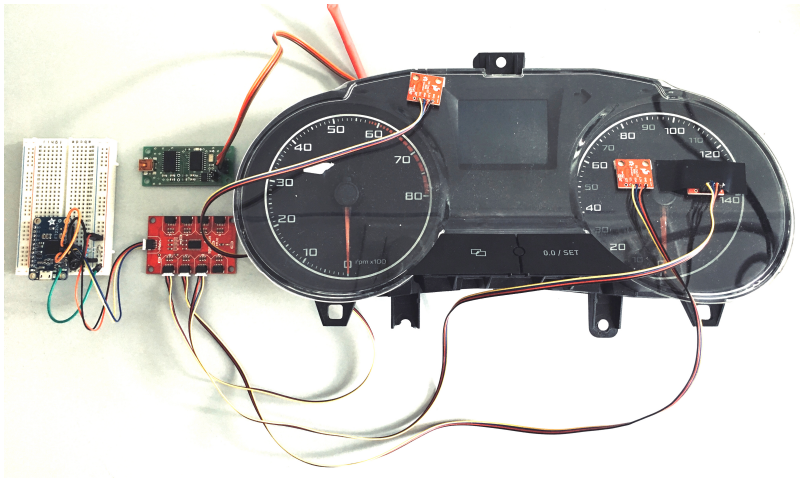

Figure 3. An automotive instrument cluster with (part of) our sensor harness attached. The cluster originates from a 2014 Seat Ibiza model.

functionality, to trace bugs in source code and fix vulnerabilities. The resulting fixes are not straight-forward as they require consideration of rather involved network states. Our findings highlight the need for thorough testing and verification on top of strong cryptographic primitives and Trusted Computing technology when designing distributed control systems that are potentially exposed to malicious interactions.

We further described how the sensor harness in combination with our fuzzing techniques can be used to largely automate the process of reverse-engineering communication protocols of proprietary ECUs. Manually reversing a substantial subset of the functionality of, e.g., an instrument cluster, can easily be an effort of several days or even weeks. With our approach, this can be achieved within hours. Additional sensors (e.g., audio, power consumption, vibration) could further extend the harness' abilities. We believe that our approach can be used to identify bugs and unintended functionality when being applied to components for which a specification is available. This specification could be integrated in a bug oracle such that responses outside of the specified behaviour are detected as errors. In this context, our approach may be useful to automate activities such as integration testing and to achieve a high input-space coverage in these activities.

Fuzzing ECUs under realistic conditions, i.e., while being connected to a vehicle's CAN network with many other "noisy" ECUs, may also be feasible but requires fuzzing strategies that aim at noise reduction by exploring the effects of individual messages or sequences of messages in different system states. In this context it may be useful to also consider CAN responses of ECUs. We may borrow from recent approaches in anomaly detection in control systems (e.g., [19] and [20]) to define oracle functions that detect changes in the response stream of an individual ECU, or even to detect state change throughout the network.

VI. Related Work

Fuzzing has a long history and is still actively developed, in particular in the domain of security- and penetration testing of software systems [14], [15]. Recent work [16] elaborates on the difficulties of employing fuzzing in embedded systems. Specifically for automotive systems, Smith states in [13] that, while fuzzing can certainly be useful in discovering undocumented services or crashes, it is rarely useful beyond that, e.g., to find and exploit vulnerabilities. Our experience report disagrees slightly with this observation: We discovered that fuzzing is more efficient in finding subtle vulnerabilities and configuration errors than monitoring or reverse engineering the firmware and communications. Fuzzing exposes substantially more of the system's unintended states than what one would be able to explore manually, due to the sheer amount of pseudo-random message combinations that are generated and dispatched by the fuzzer. This allows testers to focus on tracking down and responding to vulnerability reports instead of having to manually probe the system. With automated oracle function, as discussed in Sect. III-A and Sect. IV, fuzzing becomes even more efficient. While our approach mostly relies on black-box fuzzing where very little knowledge of the system is assumed and oracle functions must rely on system outputs rather than observing the system's internal state, our approach can certainly be combined with more advanced reverse-engineering and firmware inspection tools. This would lead to more powerful and also much more intricate oracle functions.

Related research investigates the extent to which fuzzing can be applied be in automotive systems [17], [18], [23]–[25]. Our work aims to improve over this state of the art by not only defining a fuzzer for CAN networks, but by developing an entire methodology that defines fuzzing objectives, oracle functions and fuzzing strategies, and substantially improves the automation of testing cyber-physical systems. We report on experiments and lay out our experience from applying this methodology to two realistic systems, one of which being a prototype for an automotive security system. While related work reports mixed results on the usefulness of fuzzing automotive networks, we judge our results as largely positive since we were able to identify a few subtle vulnerabilities and dramatically

speed up reverse-engineering activities.

Further related research investigates the use of fuzzing to explore stateful communication protocols [26], [27], also in the context of cyber-physical systems such as smart-grid communications [28]. Our toolchain does currently not employ such techniques. We believe that these could be a sensible addition to the current stateless exploration approach. Specifically, these techniques may be used to brute-force intricate CAN protocols that trigger, e.g., software updates on ECUs.

VII. Summary & Conclusions

Automotive control networks are highly complex safety-critical and security-critical systems which have been shown to be vulnerable to adversarial interactions. In this paper we devise a largely automated approach for fuzz-testing these systems. We discuss how bug oracles for automotive ECUs can be described, define a number of fuzzing strategies, and develop a sensor harness to allow oracle functions to detect interesting system behaviour in an automated fashion. We have implemented our fuzzing approach in CaringCaribou, an open-source automotive penetration-testing toolkit, and we report on two sets of experiments where we apply our fuzzer to find vulnerabilities and to reverse-engineer proprietary ECU functions. To the best of our knowledge, our approach is the first to achieve a high degree of automation for these activities, and we see future applications of our fuzzing approach in, e.g., penetration testing but also in integration- and compliance testing. While we have been focusing on CAN networks, we believe that the approach is applicable to other types of control networks and beyond the domain of automotive computing. In the future we will work towards a more rigorous evaluation of our approach, following the methodology of [29].

1) Acknowledgements.: This research is partially funded by the Research Fund KU Leuven. This research is partially funded under SErVO, "Secure and Economically Viable V2X Solutions", by the Flemish Agentschap Innoveren & Ondernemen. We thank the developers of CaringCaribou for their sport and ideas, and for integrating parts of our fuzzer into their platform.

References

[1] M. Wolf, A. Weimerskirch, and C. Paar, "Security in automotive bus systems," in *Workshop on Embedded Security in Cars*, 2004.

[2] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks – practical examples and selected short-term countermeasures," in *Computer Safety, Reliability, and Security (SAFECOMP '08)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 235–248.

[3] O. Henniger, L. Apvrille, A. Fuchs, Y. Roudier, A. Ruddle, and B. Weyl, "Security requirements for automotive on-board networks," in *9th International Conference on Intelligent Transport Systems Telecommunications, (ITST)*, 2009, pp. 641–646.

[4] 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, 2010 IEEE Symposium on*. IEEE, 2010, pp. 447–462.

[5] 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.

[6] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *Black Hat USA*, 2014.

[7] ——, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, 2015.

[8] SAE International, "J3061: Cybersecurity guidebook for cyber-physical vehicle systems," 2016, http://standards.sae.org/j3061_201601/.

[9] AUTOSAR Specification 4.3, "Specification of module secure onboard communication," https://www.autosar.org/standards/classic-platform/release-43/software-architecture/safety-and-security/, 2016.

[10] J. Van Bulck, J. T. Mühlberg, and F. Piessens, "VulCAN: Efficient component authentication and software isolation for automotive control networks," in *ACSAC '17*. ACM, 2017, pp. 225–237.

[11] S. Fröschle and A. Stühring, "Analyzing the capabilities of the CAN attacker," in *ESORICS '17*, ser. LNCS, vol. 10492. Heidelberg: Springer, 2017, pp. 464–482.

[12] A. Palanca, E. Evenchick, F. Maggi, and S. Zanero, "A stealth, selective, link-layer denial-of-service attack against automotive networks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 185–206.

[13] C. Smith, *The car hacker's handbook: a guide for the penetration tester*. No Starch Press, 2016.

[14] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy*, vol. 3, no. 2, pp. 58–62, 2005.

[15] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *CoRR*, vol. abs/1812.00140, 2018. [Online]. Available: http://arxiv.org/abs/1812.00140

[16] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[17] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim, "Fuzzing can packets into automobiles," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*. IEEE, 2015, pp. 817–821.

[18] S. Bayer, T. Enderle, D.-K. Oka, and M. Wolf, "Automotive security testing – the digital crash test," in *Energy Consumption and Autonomous Driving*. Springer, 2016, pp. 13–22.

[19] A. Taylor, S. Leblanc, and N. Japkowicz, "Anomaly detection in automobile control network data with long short-term memory networks," in *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 2016, pp. 130–139.

[20] C. Wressnegger, A. Kellner, and K. Rieck, "Zoe: Content-based anomaly detection for industrial control systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 127–138.

[21] S. Nürnberger and C. Rossow, "– vatiCAN – Vetted, authenticated CAN bus," in *Cryptographic Hardware and Embedded Systems – CHES '16: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–124.

[22] L. Bataille, "Volkswagen can bus gaming," URL: https://hackaday.io/project/6288-volkswagen-can-bus-gaming.

[23] S. Bayer and A. Ptok, "Don't fuss about fuzzing: Fuzzing controllers in vehicular networks," *13th escar Europe*, p. 88, 2015.

[24] R. Nishimura, R. Kurachi, K. Ito, T. Miyasaka, M. Yamamoto, and M. Mishima, "Implementation of the can-fd protocol in the fuzzing tool bestorm," in *2016 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, 2016, pp. 1–6.

[25] D. S. Fowler, J. Bryans, and S. Shaikh, "Automating fuzz test generation to improve the security of the controller area network."

[26] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *International Conference on Information Security*. Springer, 2006, pp. 343–358.

[27] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[28] H. Dantas, Z. Erkin, C. Doerr, R. Hallie, and G. v. d. Bij, "efuzz: A fuzzer for dlms/cosem electricity meters," in *Proceedings of the 2nd Workshop on Smart Energy Grid Security*. ACM, 2014, pp. 31–38.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.