Security Analysis of Vehicle Instrument Clusters by Automatic Fuzzing and Image Acquisition

Alfred Anistoroaei, Bogdan Groza, Pal-Ștefan Murvay, Horațiu Gurban
Faculty of Automatics and Computers, Politehnica University of Timisoara, Romania
Email: {alfred.anistoroaei, bogdan.groza, pal-stefan.murvay, eugen.gurban}@aut.upt.ro

***Abstract*—Automatic testing and reverse engineering of in-vehicle components have become topics of significant interest as they allow industry professionals to get a better understanding of component behaviour and attack surfaces. In this work we pursue the automatic testing of in-vehicle instrument clusters. These devices may be subject to attacks that may mislead the driver regarding the current state of the car. Understanding the feasibility of such attacks requires the ability to determine the behaviour of the cluster in response to specific instructions received from the CAN bus. For this purpose we use fuzz testing and a camera in order to remove manual intervention and set room for automatic learning of the commands that can be sent to the cluster. This enables the automatic detection of the commands with little or no human intervention. We test our framework on three clusters from real world vehicles and determine in an automatic manner a sufficiently large number of commands and responses that they trigger.**

***Keywords*-instrument cluster, fuzz testing, cybersecurity testing, CAN bus, automotive engineering**

I. Introduction and Motivation

In-vehicle components and networks are somewhat unprepared for malicious adversaries as recent investigations have proved, e.g., [1], [2], [3], etc. In order to discover potential vulnerabilities in CAN messages, to which the car instrument cluster responds, or to check how easy an attacker might discover the commands sent to the cluster, we pursue the design of an automated system which uses fuzzing over all possible standard CAN identifiers (IDs) on 11 bits and a range of payloads. Such an analysis will give a comprehensive idea of how easy is to reverse engineer an instrument cluster by adversaries.

In order to understand the relevance of instrument clusters, a few words on the history of these devices are useful. At first, rudimentary instrument clusters were used for monitoring oil level, water pressure or coolant temperature [4]. With technological advancements, instrument clusters also evolved. In the 50s, additional information on the speed, fuel level, oil pressure and coolant temperature were displayed, while warning lights were introduced only later replacing some of the gauges [5]. Currently, most clusters are equipped with LCD panels. Clusters produced for most low to mid-range vehicles use one central LCD display along with mechanical needle gauges, while in high-end cars a full LCD instrument cluster is preferred.

The instrument cluster is a relevant component for the safety and security of the whole car. This is because the driver is continuously informed on the state of the car by observing information displayed on the instrument cluster. This includes information on the speed at which the car is travelling, the need to refuel as well as various warning messages that indicate malfunctions. Clearly, an adversary manipulating such messages can misled the driver, possibly causing accidents making the driver drive at an unwanted speed or making the driver stop in undesired circumstances, e.g., on a highway, by injecting false messages on the CAN bus. It is well known and already proved by many works that the CAN bus has no intrinsic security and thus injecting messages on the bus is a simple task. The only problem is for an adversary to learn the instructions, i.e., the CAN identifiers and specific payloads, to which the cluster instrument will respond by changing the information displayed on the panel. For a better understanding of this task, Figure 1 depicts an overview of the system which we use. Briefly, we use a motion detection algorithm which processes the images recorded by a camera and checks for various events such as moving needles, steady or blinking lights. The camera is coupled to a laptop which uses a RaspberryPi board in order to communicate with the cluster and observe its behaviour. More details on this setup will be given in the forthcoming sections.

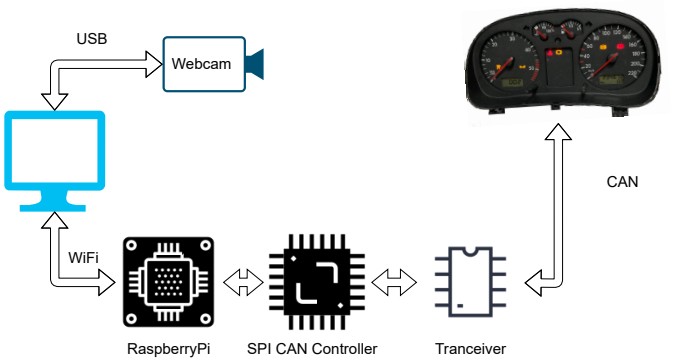

Fig. 1. Overview of the proposed system

The rest of our work is organized as follows. In Section II we discuss some related works on reverse engineering car components. Section III presents the laboratory setup in which we test three instrument clusters from common passenger cars. Section IV presents the design of the automatic fuzz testing mechanism detailing the concept, implementation and

978-1-6654-7933-2/22/$31.00 ©2022 IEEE

experimental results. Finally, Section V holds the conclusion of our work.

II. Related work

Reverse engineering CAN messages inside a car network was the target of multiple research works. Existing works on this subject use different fuzzing techniques, big data analysis or manual reverse engineering of messages. In what follows we try to briefly account for some of these works.

In [6], the authors send high volume of random or malformed data to the Display Interface ECU (Electronic Control Unit), i.e., the ECU controlling the instrument cluster, and reactions are recorded for later analysis. More precisely, the injected data consists of CAN messages with IDs in the 0–2047 range and payload bytes randomized over the full 64-byte range. Messages are sent at a rate of one packet per millisecond, using a PC based CAN fuzzer developed by the authors in a previous paper, but due to the system's processing time, the response to the message is delayed. The fuzzing procedure runs multiple times over the messages, with a binary search algorithm, to reduce the number of messages on each try, until the message producing the desired effect is found. Afterwards, the payload is analyzed in the same manner, sending the same message ID and varying only the bytes of the message payload. This work does not use a camera to automatically detects changes.

An automated method for reverse engineering the car's cluster is proposed in [7]. This is done through different fuzzing techniques. In order to capture the instrument cluster's output, the authors use light sensors. Using this approach the authors were successful in reverse engineering real and simulated instrument cluster and the time performance of several fuzz testing algorithms were measured. A simulated cluster is also used in [8], to prove the success of fuzz testing on an automotive component, and in [9], to show the tools and methodology needed to hack an instrument cluster.

Another approach is to analyze sniffed logs from the car in order to discover the CAN messages that are used for instrument cluster control. For example, in [10] the authors crashed a car in a controlled environment and recorded the CAN traffic preceding the accident. Afterwards, the authors analyzed the traffic knowing the behavior before the crash (speed, time of the crash, breaking time, etc.) and pieced together all these information to obtain meaningful information about the employed messages. In [11], the authors performed a man-in-the-middle attack and spoof CAN messages in order to take control of the cluster. In [12], collected CAN bus traffic was analyzed by different message filtering techniques for several car models.

Emulating part of instrument cluster's firmware is also an option for reverse engineering, exploited in [13]. Demonstrations of how different vulnerabilities can lead to malicious attacks on clusters were shown in [14], where an Android head unit accessible with root privileges was used for injecting malicious messages. Also, in [15] an OBD2-to-Bluetooth device and an Android malicious app were used to insert malicious CAN frames. Attacks on the entire car and attempts of reverse engineering CAN messages were also performed in [2] or in [16]. Procedures for automated fuzzing CAN frames were not used only for attacking or reverse engineering purposes, but also for testing and checking the behavior of clusters in [17], [18] or [19].

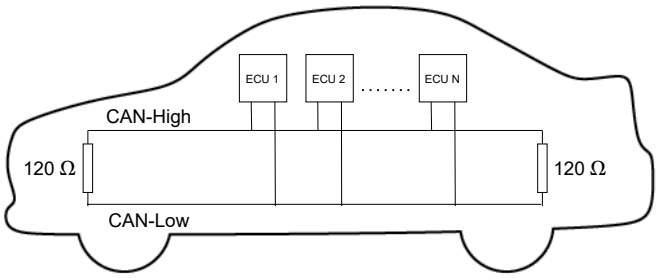

Fig. 2. CAN bus network representation

Fig. 3. CAN frame structure

As potential protection mechanisms, one way to circumvent the automated fuzzing of the instrument cluster would be to use identifier randomization. This procedure has been proposed by a number of recent works, e.g., [20], [21], [22], [23], as a protection against frame injection attacks. However, such a procedure, while effective, is not yet adopted by manufacturers.

III. Laboratory setup

In this section we first discuss some background on CAN buses, then we proceed to the description of our experimental setup in which we test the three in-vehicle clusters.

A. CAN buses background on instrument clusters

Within the car network architecture, the instrument cluster represents a critical component. It may be connected to one or several CAN networks where rogue nodes may reside, or where an On-board diagnostics (OBD) port exists and can be used as an attack surface [24]. The Bosch Group started the development of the Controller Area Network (CAN) protocol in 1983 and its specification was released three years later, in 1986. Since then it has become an automotive standard, published in 1993 as ISO 11898. Due to its low cost, robustness, speed and flexibility the CAN bus is widely used by ECUs found in automotive networks, including the instrument cluster.

The CAN bus is a serial communication protocol with a physical layer implemented using two dedicated wires, CAN-High and CAN-Low. An 120 ohm resistor is used as a terminator at each end of the bus. Each node, usually referred as Electronic Control Unit (ECU), which has to communicate on this bus, must be linked to these two wires. A basic

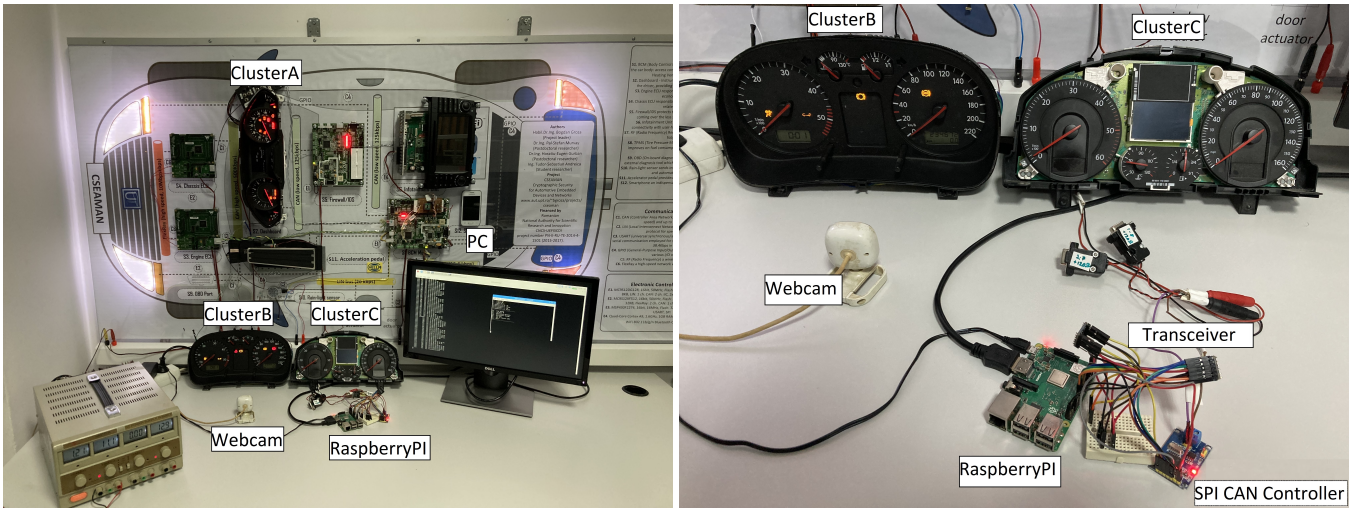

Fig. 4. Experimental setup with the components near a car-on-bench setup (left) and detailed view of the setup with two instrument clusters, the camera and a RaspberryPi used to send messages to the cluster (right)

representation of the CAN bus architecture can be seen in Figure 2. At the physical layer, bits are represented as voltage differences. When one recessive bit, i.e., logical 0, is sent, both lines have 2.5 volts, having a differential voltage of about 0 volts. When a dominant bit is sent, i.e., logical 1, CAN-H has about 3.5 volts, while CAN-L has about 1.5 volts, resulting in a differential voltage of about 2 volts. The CAN bit rate is limited to 1 Mbit/s while its newer embodiment with flexible data-rates, CAN-FD, allows up to 8 Mbit/s for the data-field. All the tested clusters in our experiments employed the same bit rate, i.e., 500kbit/s.

All the CAN data is structured in frames as represented in Figure 3, where Start-of-frame denotes the start of frame transmission, CAN ID is a unique identifier representing the message priority, RTR the remote transmission request, Control specifies data size and message ID length, Data is the actual data sent, CRC stands for cyclic redundancy check, ACK is the acknowledgement of a valid frame and End-of-frame marks the end of the frame transmission. In our work we make use of the two blocks of bits marked with gray, i.e., the ID and Data fields, as we explain later in the paper. The range for the ID field is from 0 to 2047 for standard 11-bit IDs, while the Data field can hold up to 8 bytes. Extended IDs are 29 bits long which would make fuzzing more difficult. However, this type of IDs are commonly used in heavy-duty vehicles and remain uncommon for passenger cars.

B. Setup

In this section we present the clusters that were used in our setup and their connectivity. For our setup we used the following components: a laptop with USB and WiFi connection, an external webcam, a Raspberry Pi development board and three distinct instrument clusters. The laptop is connected via USB to an external webcam and via WiFi to the Raspberry Pi board. Further, the Raspberry Pi board is directly connected to the instrument cluster. For taking pictures we used the webcam and the C# VideoCaptureDevice class.

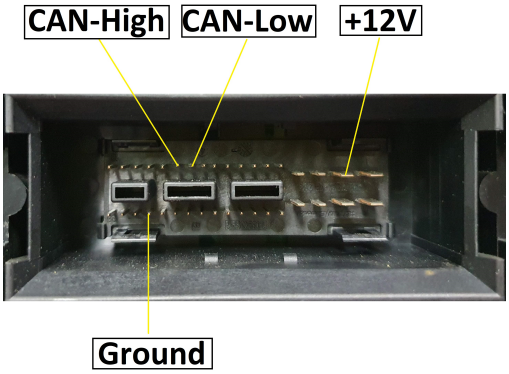

Fig. 5. The connection pinout for Cluster B

In our experiments we used three clusters. In order to avoid disclosing information that could be used for any potential harmful actions we will keep the manufacturers anonymous. We will refer these clusters as Cluster A, B and C. The first and the second clusters are from more recent cars, produced after 2010. The third cluster was from an older car model, produced in 2001. The entire setup with all the three clusters can be seen in Figure 4.

To determine the pin-out, we had to search the internet for specific manuals and instructions. These were easy to retrieve and follow as there were basically four pins that need to be connected: the power supply, ground, CAN-High and CAN-Low wires [25], [26]. Further the Raspberry PI was linked to a CAN controller since it does not natively supports CAN bus communication. A CAN transceiver was employed to interface the bus with the CAN controller. The power supply, CAN-Low and CAN-High wires were linked to the cluster as indicated in Figure 5.

IV. Designing the automatic fuzz testing mecanism

This section presents the procedure that we designed for the automatic fuzz testing along with some of the difficulties encountered at the design time.

A. Concept and implementation

A C# application, that was deployed on a laptop, is in control of the entire fuzz testing process. The application communicates with a server implemented on the RaspberryPi board through TCP/IP sockets. This just acts as a proxy and is used for sending CAN message to the cluster using SocketCan utilities[1]. We choose to send the CAN messages using a RaspberryPi instead of using another dedicated USB-to-CAN converter from the PC since the RaspberryPi CAN extension provided a finer grain control for message transmission on the CAN bus.

The fuzz testing procedure starts with taking an initial photo of the cluster that will be used as a baseline for comparison. This photo, like subsequently taken photos, is converted to grayscale for easier processing. The basic fuzz testing process consist in sending CAN frames starting from ID 0. The ID field is incremented for each subsequent transmission while the payload consists in 8 randomly generated bytes. A 200 millisecond waiting time is allowed for the cluster to react after each transmission. Afterwards, 3 pictures are taken in burst mode and their grayscale version is compared with the baseline image. If any change is identified, the new images are saved in a folder named after the ID of the message used to trigger the effect.

We improved on this basic approach with the aim of identifying specific bits that are responsible with changing indicators on the cluster. For each ID that was found to produce an effect we sent a series of messages each having a single bit (out of the 64 bits available in a CAN frame) set to 1. A total of 64 messages are sent in this step, one for each bit position in the CAN data field. This approach was inspired by the fact that most indicator lights displayed on a cluster are generated based on corresponding flags in the frame. A detailed flow chart our fuzzing system implementation is illustrated in the Figure 6

During the development phase we encountered several unexpected issues that we had to cope with. As presented in the beginning of this section, after we send the CAN message, we had to take three photos in burst mode. We do this because we observed that in the case of blinking indicators (e.g., turning lights), a single picture may miss the exact moment when the LED was on, and only capture the moment when it was off. This would falsely indicate that the message transmission had no effect and lead to skipping an ID that actually produces an effect. Another problem caused by blinking lights was that once they were activated by a CAN message and kept blinking, the system could falsely report subsequent CAN messages as producing an effect (e.g., stop blinking). A solution was to physically cover these lights once the corresponding message was identified.

[1]https://www.kernel.org/doc/html/latest/networking/can.html

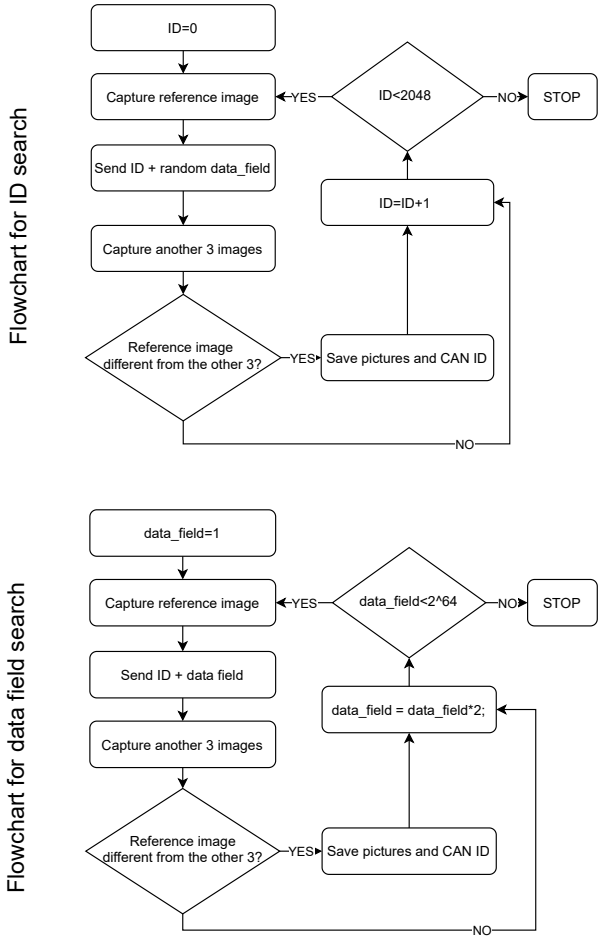

Fig. 6. The software flow chart for ID search (up) and the flowchart for algorithm adapted for message search (down)

B. Experimental results

This section sums up the experimental findings for each of the clusters that we tested.

All the results are summarized in Table I, where the first column indicates the cluster that was tested and the second one indicates the ID of the message that was found to generate an effect. The third column indicates whether we discovered the message automatically, using our algorithm, or manually using information publicly available on the internet. The fourth column shows the message that produces the effect described in the fifth column.

For Cluster A our automated approach led to the discovery of 12 IDs with 39 messages which produce multiple effects, from turning on different lights, e.g., seat-belt, high beam, low beam, etc., to changing the speedometer or RPM needles.

For Cluster B our algorithm discovered 7 IDs and a total of 14 messages. Moreover, by using information from various

sources such as [27], we managed to find combinations of messages which triggered different effects like glow plug or check engine lights. To test this we sent two consecutive messages with different IDs and random payload. We found two such cases, i.e., ID 15 followed by 08 and 13 followed by 0E as illustrated in Table I.

For Cluster C, due to the fact that it was from an older car model, the automated discovery only revealed an ID and two messages. Using additional information available on the internet we were able to validate another message that controls the built-in speaker and we also managed to control the speedometer by applying a rectangular signal at the input of a specific pin. This also led to changes in the mileage indicator according to the input speed.

As part of our analysis we also measured the automated test duration. All the operations involved with a single message required less than one second. These include: taking a picture, converting it to grayscale and byte array, sending the CAN message, waiting 200ms for the cluster to react, taking three new pictures with a 100ms delay and converting them to grayscale and byte array. Consequently, for analyzing the effect of all 2048 IDs it took approximately 30 minutes.

V. Conclusion

In this work we tried to design an automated fuzzing solution for reverse engineering the instrument cluster and we obtained reasonably good results for a first attempt. We tested the proposed approach on 3 different instrument clusters and identified between one and twelve messages IDs employed by these clusters. By further analyzing the effects of the payload content we discovered a series of functionalities that can be activated with these IDs. This proves that this methodology is feasible for automatic discovery of some of the commands for in-vehicle clusters. The development in this work gives a better idea on the difficulty of reverse engineering messages that target the instrument cluster and on designing automatic tests that may allow checking the resilience of such clusters against adversaries. As future work we may conduct similar analysis on more complex vehicle clusters.

TABLE I
Discovered IDs and messages for the three instrument clusters in the experiments

Dev.	ID	Test	Message	Description
A	01	A	xx 00 00 00 00 00 xx xx	Background light
A	03	A	00 00 xx 00 00 00 00 00	Speed needle
A	03	A	x0 00 00 00 00 00 00 00	RPM needle
A	04	A	00 00 xx xx x0 00 00 00	Mileage value
A	04	A	08 00 00 00 00 00 00 00	Needels check
A	05	A	00 00 00 00 00 00 00 40	Engine warning
A	05	A	00 00 00 00 00 00 00 40	Engine error (2 times)
A	05	A	00 00 00 00 00 00 10 00	Seatbelt
A	05	A	00 00 00 00 00 00 40 00	Seatbelt
A	05	A	00 00 00 00 00 02 00 00	Urea warning light
A	05	A	00 00 00 00 00 10 00 00	Seatbelt
A	05	A	00 00 00 00 00 40 00 00	Seatbelt
A	05	A	00 00 00 00 00 80 00 00	Fuel tank empty
A	05	A	00 00 00 00 40 00 00 00	Open doors warning
A	05	A	00 00 00 04 00 00 00 00	Automatic shift lock
A	05	A	00 00 00 10 00 00 00 00	Deactivated airbag
A	05	A	02 00 00 00 00 00 00 00	Left turning light
A	05	A	04 00 00 00 00 00 00 00	Right turning light
A	05	A	08 00 00 00 00 00 00 00	Front fog lights
A	05	A	10 00 00 00 00 00 00 00	Rear fog lights
A	05	A	20 00 00 00 00 00 00 00	High beam
A	05	A	40 00 00 00 00 00 00 00	Low beam
A	05	A	80 00 00 00 00 00 00 00	Headlights
A	06	A	00 00 00 x0 00 00 00 00	Fuel level
A	07	A	00 00 00 00 01 00 00 00	ECO
A	07	A	00 00 00 00 40 00 00 00	Low beam blinking
A	07	A	00 00 00 00 80 00 00 00	Empty fuel tank warn.
A	07	A	00 00 00 08 00 00 00 00	Low temp. warning
A	07	A	00 00 02 00 00 00 00 00	Battery warning
A	07	A	00 10 00 00 00 00 00 00	Engine warning
A	07	A	00 08 00 00 00 00 00 00	Auto wipers
A	07	A	08 00 00 00 00 00 00 00	Low oil pressure
A	07	A	80 00 00 00 00 00 00 00	Water temp. error
A	09	A	Random	Mileage value
A	0A	A	Random	Mileage blinking
A	0B	A	Random	Cruise control
A	0C	A	10 00 00 00 00 00 00 00	Battery indicator
A	0D	A	x0 00 00 00 00 00 00 0f	Middle panel watch
A	11	A	08 00 00 00 00 00 00 00	Key empty battery
B	02	A	00 00 04	Seatbelt light
B	08	A	00 01	ABS
B	08	A	00 00 00 00 00 00 00 10	ESP
B	08	A	00 00 00 00 00 00 00 02	ESP
B	0E	A	00 00 00 x0	RPM
B	0F	A	00 00 00 01	Steering error
B	0F	A	00 00 00 02	Steering warning
B	12	A	00 00 00 00 10	Bulb failure
B	14	A	00 00 01	Left turning light
B	14	A	00 00 02	Right turning light
B	14	A	00 00 03	Hazard lights
B	14	A	04	High beam
B	14	A	10	Fog lights
B	16	A	00 00 00 00 00 04	Break system
B	15+08	M	00 00 00 08	Low tyre pressure
B	15+08	M	00 xx xx	Speedometer
B	13+0E	M	00 02	Glow plug
B	13+0E	M	00 08	Check engine
B	13+0E	M	00 00 00 00 00 02	DPF light
C	10	A	00 02	Steering warning
C	10	A	00 01	Steering warning OFF
C	10	M	00 20	Buzzer

References

[1] 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*, vol. 4, no. 447-462. San Francisco, 2011, p. 2021.

[2] 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 *2010 IEEE symposium on security and privacy*. IEEE, 2010, pp. 447–462.

[3] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, no. S 91, 2015.

[4] M. Akamatsu, P. Green, and K. Bengler, "Automotive technology and human factors research: Past, present, and future," *International journal of vehicular technology*, vol. 2013, 2013.

[5] W. Ribbens, *Understanding automotive electronics: an engineering perspective*. Butterworth-heinemann, 2017.

[6] D. S. Fowler, J. Bryans, M. Cheah, P. Wooderson, and S. A. Shaikh, "A method for constructing automotive cybersecurity tests, a can fuzz testing example," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2019, pp. 1–8.

[7] T. Werquin, "Automated reverse engineering and fuzzing of the can bus," *KU Leuven*, 2019.

[8] D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson, "Fuzz testing for automotive cyber-security," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 239–246.

[9] B. R. Payne, "Car hacking: Accessing and exploiting the can bus protocol," *Journal of Cybersecurity Education, Research and Practice*, vol. 2019, no. 1, p. 5, 2019.

[10] J. Staggs, "How to hack your mini cooper: reverse engineering can messages on passenger automobiles," *Institute for Information Security*, 2013.

[11] R. Buttigieg, M. Farrugia, and C. Meli, "Security issues in controller area networks in automobiles," in *2017 18th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*. IEEE, 2017, pp. 93–98.

[12] D. Frassinelli, S. Park, and S. Nürnberger, "I know where you parked last summer: Automated reverse engineering and privacy analysis of modern cars," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1401–1415.

[13] A. Milburn and N. Timmers, "Efficient reverse engineering of automotive firmware," *escar USA*, 2018.

[14] G. Costantino and I. Matteucci, "Candy cream-hacking infotainment android systems to command instrument cluster via can data frame," in *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 2019, pp. 476–481.

[15] S. Woo, H. J. Jo, and D. H. Lee, "A practical wireless attack on the connected car and security protocol for in-vehicle can," *IEEE Transactions on intelligent transportation systems*, vol. 16, no. 2, pp. 993–1006, 2014.

[16] 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.

[17] Y. Huang, A. Mouzakitis, R. McMurran, G. Dhadyalla, and R. P. Jones, "Design validation testing of vehicle instrument cluster using machine vision and hardware-in-the-loop," in *2008 IEEE International Conference on Vehicular Electronics and Safety*. IEEE, 2008, pp. 265–270.

[18] M. A. Taut, M. Daraban, and D. Pitica, "Software instrument for testing cluster devices using high-speed can protocol," in *2017 40th International Spring Seminar on Electronics Technology (ISSE)*. IEEE, 2017, pp. 1–6.

[19] S. Nimara, D. B. Popa, and R. Bogdan, "Automotive instrument cluster screen content validation," in *2017 25th Telecommunication Forum (TELFOR)*. IEEE, 2017, pp. 1–4.

[20] B. Groza, L. Popa, and P.-S. Murvay, "Highly efficient authentication for can by identifier reallocation with ordered cmacs," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 6, pp. 6129–6140, 2020.

[21] S. Woo, D. Moon, T.-Y. Youn, Y. Lee, and Y. Kim, "Can id shuffling technique (cist): Moving target defense strategy for protecting in-vehicle can," *IEEE Access*, vol. 7, pp. 15 521–15 536, 2019.

[22] W. Wu, R. Kurachi, G. Zeng, Y. Matsubara, H. Takada, R. Li, and K. Li, "Idh-can: A hardware-based id hopping can mechanism with enhanced security for automotive real-time applications," *IEEE Access*, vol. 6, pp. 54 607–54 623, 2018.

[23] A. Humayed and B. Luo, "Using id-hopping to defend against targeted dos on can," in *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, 2017, pp. 19–26.

[24] E. H. Gurban, B. Groza, and P.-S. Murvay, "Risk assessment and security countermeasures for vehicular instrument clusters," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018, pp. 223–230.

[25] "Forum page 1 for cluster pinout," https://www.vwvortex.com/threads/mk4-cluster-in-mk3-diy-of-sorts.4651016/?id=4651016, accessed: 2021-11-27.

[26] "Forum page 2 for cluster pinout," https://mhhauto.com/Thread-golf-v-cluster-lights-on-table?pid=1195020, accessed: 2021-11-27.

[27] "Can bus gaming simulator," https://hackaday.io/project/6288-can-bus-gaming-simulator, accessed: 2021-11-27.