IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 19, 2022, accepted February 1, 2022, date of publication February 14, 2022, date of current version March 4, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3151358

Efficient ECU Analysis Technology Through Structure-Aware CAN Fuzzing

HYUNGHOON KIM[1], YEONSEON JEONG[2], WONSUK CHOI[3], DOON HOON LEE[2], (Member, IEEE), AND HYO JIN JO[1], (Member, IEEE)

[1]School of Software, Soongsil University, Seoul 06978, South Korea
[2]School of Cybersecurity, Korea University, Seoul 02841, South Korea
[3]Division of IT Convergence Engineering, Hansung University, Seoul 136-792, South Korea

Corresponding author: Hyo Jin Jo (hyojin.jo@ssu.ac.kr)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant through the Korean Government [Ministry of Science and ICT (MSIT)], AI-Based Cyber Attacks and Defense for Autonomous Vehicles, under Grant 2021-0-00111.

ABSTRACT Modern vehicles are equipped with a number of electronic control units (ECUs), which control vehicles efficiently by communicating with each other through the controller area network (CAN). However, the CAN is known to be vulnerable to cyber attacks because it does not have any security mechanisms. To find vulnerable CAN messages that can control safety-critical functions in ECUs, researchers have studied CAN fuzzing methods. In existing CAN fuzzing methods, fuzzing input values are generally generated at random without considering the structure of CAN messages, resulting in non-negligible CAN fuzzing time. In addition, existing fuzzing solutions have limited monitoring capabilities of the fuzzing results. In this paper, we propose a Structure-aware CAN Fuzzing protocol, in which the structure of CAN messages is considered and fuzzing input values are systematically generated to locate vulnerable functions in ECUs. Our proposed Structure-aware CAN Fuzzing system takes less time to run than existing solutions, meaning that problematic CAN messages that may have originated from SW implementation errors or CAN DBC (database CAN) design errors can be found quickly and, subsequently, appropriate action can be taken. Finally, we evaluated the performance of our Structure-aware CAN Fuzzing system on two real vehicles. We proved that our proposed method can find CAN messages that control safety-critical functions in ECUs faster than existing fuzzing solutions.

INDEX TERMS ECU, CAN, vehicle hacking, CAN fuzzing, structure-aware CAN fuzzing.

I. INTRODUCTION

Modern vehicles are equipped with a number of electronic control units (ECUs) that control electronic systems such as the engine, airbags, brake, and so on. ECUs communicate with each other through several in-vehicle networks (IVNs), such as the controller area network (CAN), FlexRay, local interconnect network (LIN), CAN with flexible data rate (CAN-FD), and automotive ethernet. In 1993, the International Organization for Standardization (ISO) adopted CAN as a de-facto standard (i.e., ISO 11898) for communication between ECUs [1] for its prime suitability with vehicle features.

A vehicle can be efficiently controlled by CAN communications, but the following security issues remain problematic. First, attackers can eavesdrop on all messages on the CAN-bus because the messages are broadcast without encryption. Second, attackers can gain control of a vehicle by injecting malicious messages because there are no access controls or authentication procedures on the network [2]–[8]. This can be illustrated by one key vehicle hacking result, in which Miller and Valasek were able to remotely control critical operations such as engine and brake function of a Jeep Cherokee [6]. In addition to remote attacks using wireless interfaces (e.g., Bluetooth or Wi-Fi), as in the aforementioned Jeep Cherokee attack, attacks based on wired interfaces (e.g., On-Board Diagnostics (OBD)-II port) have also been conducted [3].

To deal with cyber attacks, attack surfaces on a vehicle should be analyzed in advance before they can be exploited by a potential attacker. Analysis of attack surfaces can be divided into two parts: 1) vulnerability research on external networks (e.g., vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I), etc.) and 2) vulnerability research on

The associate editor coordinating the review of this manuscript and approving it for publication was Diana Gratiela Berbecaru.

in-vehicle networks. Vulnerabilities of external networks can be analyzed with several open-source tools, such as NMAP and ubertooth, and vulnerabilities of in-vehicle networks can be analyzed with reverse engineering methods [3], [4] that analyze ECU firmware and CAN fuzzing methods[1,2,3,4,5] that analyze CAN messages related to safety-critical ECU control.

While there is much active research that analyzes the vulnerabilities of external networks, research on the vulnerabilities of in-vehicle networks has several limitations. First, the reverse engineering of ECU firmware requires physical access to a target ECU to locate a debugging port and obtain the corresponding firmware. Second, CAN fuzzing requires non-negligible analysis time to randomly transmit fuzzing input values to the CAN-bus. In addition, it is difficult to monitor the responses corresponding to brute force fuzzing input values because CAN DBC (database CAN) is securely managed and undisclosed, so there is a limit to how much the meaning of CAN messages can be analyzed.

To overcome these limitations in analysis methods for in-vehicle networks, we herein propose a Structure-aware CAN Fuzzing system as a protocol that minimizes fuzzing time by considering the structure of a CAN message. In other words, our goal is not to identify all CAN messages that can cause misbehavior of the target vehicle but rather it is to identify some CAN messages that can cause misbehavior of the target vehicle in a short period of time. Additionally, the proposed method involves network-based fuzzing, which does not need to reverse engineer ECU firmware through a debugging port.

The proposed method consists of three phases: 1) CAN message analysis, 2) fuzzing input value generator, and 3) monitoring. In the CAN message analysis phase, the structure of the CAN message is analyzed. In the fuzzing input value generator phase, the fuzzing rules based on analysis results are defined, and structure-aware CAN messages are generated and injected into the vehicle. Finally, in the monitoring phase, the corresponding responses are monitored using side information.

The contributions of this paper are as follows.

1) We are the first to present a methodology that generates Structure-aware CAN Fuzzing input values based on the structure of a CAN message. The proposed method takes less fuzzing time than existing brute force CAN fuzzing methods. In our experiment, we found ten CAN IDs related to the engine of two real vehicles (i.e., we found four CAN IDs in one vehicle and six in the other). The number of fuzzing input values in our method is less than 2^8 while that of the existing brute force CAN fuzzing methods is greater than 2^{66} for each test vehicle.
2) By monitoring side information (e.g., IMU sensor values) of the fuzzing target vehicles, this method automatically finds misbehavior caused by injected fuzzing input values.
3) To validate the efficiency and practicality of the proposed method, we conducted experiments with two real vehicles and several vulnerable CAN messages (e.g., controlling steering and engine acceleration) that could be exploited as attack messages.

This paper is organized as follows. We introduce background knowledge and related works in Section II, and Section III describes the system design of our Structure-aware CAN Fuzzing system in detail. In Section IV, we evaluate our method through experiments on real vehicles and discuss the limitations of our proposal in Section V. Finally, we present our conclusion and offer direction for future research in Section VI.

SOF	ID	RTR	IDE	R	DLC	Data Field	CRC Field	ACK Field	EOF
1 bit	11 bit	1 bit	1 bit	1 bit	4 bit	0~8 byte	16 bit	2 bit	7 bit

FIGURE 1. The structure of CAN data frame.

II. BACKGROUND AND RELATED WORKS

A. BACKGROUND

1) CONTROLLER AREA NETWORK

Robert Bosch developed the CAN based on a bus topology (CAN-bus) and released the technology in 1986. In the CAN physical layer, wires are a twisted pair of cables to provide protection against electrical noise. The maximum speed of the CAN-bus is 1Mbit/s, but the actual baud-rate on a vehicle is 500Kbit/s. The maximum length of data is 8 bytes [9].

Fig. 1 shows the structure of a CAN Data Frame. Start of Frame (SOF) denotes the start of the frame, while Identifier (ID) is used to arbitrate the frame priority and Remote Transmission Request (RTR) identifies whether the frame is a data frame or a remote frame. The Identifier Extension Bit (IDE) distinguishes whether a frame is standard or extended, and Reserved (R) area is a pre-designation for an extended frame. Data Length Code (DLC) indicates the length of the data field, and the data field contains the data transmitted by ECUs. Cyclic Redundancy Check (CRC) field checks frame errors, and the Acknowledgement (ACK) field recognizes reception of a valid frame. Finally, End of Frame (EOF) denotes the end of a frame. In this paper, we focus on the ID, DLC, and data field of the CAN data frame.

2) OBD-II PID

OBD-II is a vehicle diagnostic service used to diagnose vehicle status. Parameter identifiers (PIDs) are codes that check the vehicle status, and all modern vehicles follow the OBD-II PID standard defined by SAE J1962.[6] However, not all

[1] [Accessed: Feb. 1, 2022] https://github.com/CANToolz/CANToolz
[2] [Accessed: Feb. 1, 2022] https://github.com/TianTianlove/ATG-python
[3] [Accessed: Feb. 1, 2022] https://github.com/bhass1/pyfuzz_can
[4] [Accessed: Feb. 1, 2022] https://github.com/zombieCraig/UDSim
[5] [Accessed: Feb. 1, 2022] https://github.com/CaringCaribou/caringcaribou
[6] [Accessed: Feb. 1, 2022] https://en.wikipedia.org/wiki/OBD-II_PIDs

vehicles support all standard PIDs, and vehicle manufacturers can create customize their own identifiers. Therefore, it is important to find the supported PIDs for each vehicle, which can be confirmed via a request and response process included in PID services. When PIDs make a request query using CAN ID $0 \times 7DF$ with PID, a code is sent to the CAN-bus of the vehicle connected to the OBD-II port, and the vehicle then sends a response query with the CAN ID in the range $0 \times 7E8$ to $0 \times 7EF$. The response query includes a return value that represents the vehicle status or supported PIDs.

B. RELATED WORKS

1) CAN MESSAGE ANALYSIS METHODS

Vehicle manufacturers keep Database CAN, called CAN DBC files, secret because attackers may cause a vehicle to misbehave using knowledge obtained from the DBC files. However, researchers have already discovered that DBC files can be extrapolated through a CAN message analysis method (e.g., CAN message reverse engineering [10]–[17].

References [10] and [11] proposed the ACA algorithm, which effectively analyzes CAN messages to solve the problem of CAN message reverse engineering taking non-negligible time. The ACA algorithm finds supported PIDs using the OBD-II PID standard, and then requests a PID diagnostic query from a test vehicle with the PID code that it wants to analyze. By analyzing the relationship between the response data of the PID query and the CAN messages, the ACA algorithm finds both the CAN IDs and the corresponding data fields related to the requested PID code.

Alternatively, an automatic reverse engineering method uses the READ algorithm [13] to extract the signal boundaries of the CAN data frame. READ logs CAN messages received from the vehicle and creates sub-files of CAN messages with the same ID. READ calculates the bit-flip rate, which indicates how often the bit value changes at each bit position in the data field (64-bit) of each sub-file. Finally, READ extracts signal boundaries and labels the signals (e.g., Counter, CRC, etc.) encoded in the data field using the bit-flip rate. In this research, a comparison of the formal CAN message specifications for the test vehicle confirmed the results of READ to be highly accurate.

LibreCAN [15] subsequently proposed a framework that analyzes CAN message signals of a vehicle's powertrain and body using the OBD-II PID protocol and the built-in Inertial Measurement Unit (IMU) sensor on a smartphone. LibreCAN adopts the methodology of READ [13] (i.e., bit-flip rate-based signal extraction) for signal extraction from CAN messages. After extracting signals from CAN messages, it identifies the signal boundaries more accurately than READ [13] via a heuristic method. Finally, LibreCAN translates the meaning of CAN messages by using the identified signals obtained from the OBD-II PID and the IMU sensor.

2) CAN FUZZING METHODS

Some studies on ECU vulnerability analysis methods based on CAN fuzzing do not need to reverse engineer the ECU firmware through a debugging port. In general, existing CAN fuzzing methods[1,2,3,4,5] run their own fuzzing algorithms using random input values without considering the meaning of the CAN IDs and the data fields [18]–[23].

Therefore, existing fuzzing methods tend to take non-negligible time because these methods are required to inject all possible values of the CAN data field (i.e., if DLC is 8, the number of input values that will be injected to CAN is 2^{64}). In addition, there is no published work on monitoring methods for the responses to fuzzing input values.

III. THE PROPOSED SYSTEM: STRUCTURE-AWARE CAN FUZZING

In this section, we introduce our proposed system that can systematically generate fuzzing input values and automatically monitor responses triggered by fuzzing input values.

A. OVERVIEW

Fig. 2 shows an overview of our Structure-aware CAN Fuzzing system, which consists of three phases. We outline these phases next.

1) PHASE 1: CAN MESSAGE ANALYSIS

Phase 1 is composed of three parts: bit-flip rate analysis, OBD-II PID analysis, and correlation analysis.

- Bit-flip rate analysis: Signal boundaries are identified by calculating the bit-flip rate of the CAN data field. Then, the identified signal boundaries are mapped to seven signal types–*Unused*, *Constant*, *Multi-value*, *Sensor*, *Counter*, *Checksum*, and *Undefined* field–via a heuristic method.
- OBD-II PID analysis: By checking the supported PIDs provided by the OBD-II PID standard, CAN IDs related to a specific function (e.g., engine RPM, throttle position, etc.) can be identified.
- Correlation analysis: The relationships between CAN IDs are analyzed by calculating the correlation of signals identified as being in the *Sensor* field, which is one of the defined seven signals.

2) PHASE 2: FUZZING INPUT VALUE GENERATOR

Phase 2 is composed of three parts: CAN fuzzing rule definition, CAN fuzzing input value generation, and CAN fuzzing input value injection.

- CAN fuzzing rule definition: By using the analysis results in Phase 1, the system can define the fuzzing rules that can cause vehicles to misbehave.
- CAN fuzzing input value generation: The fuzzing input values are generated by using fuzzing rules. In our proposed process, if a fuzzing input value is inferred to have the *Checksum* field, the checksum value corresponding to the fuzzing input should be generated in two ways. If the fuzzing input value to be injected exists in the CAN log recorded in a fuzzing target vehicle, the checksum value obtained from the CAN log can be used as is. Otherwise, a deep learning algorithm infers a new checksum value.

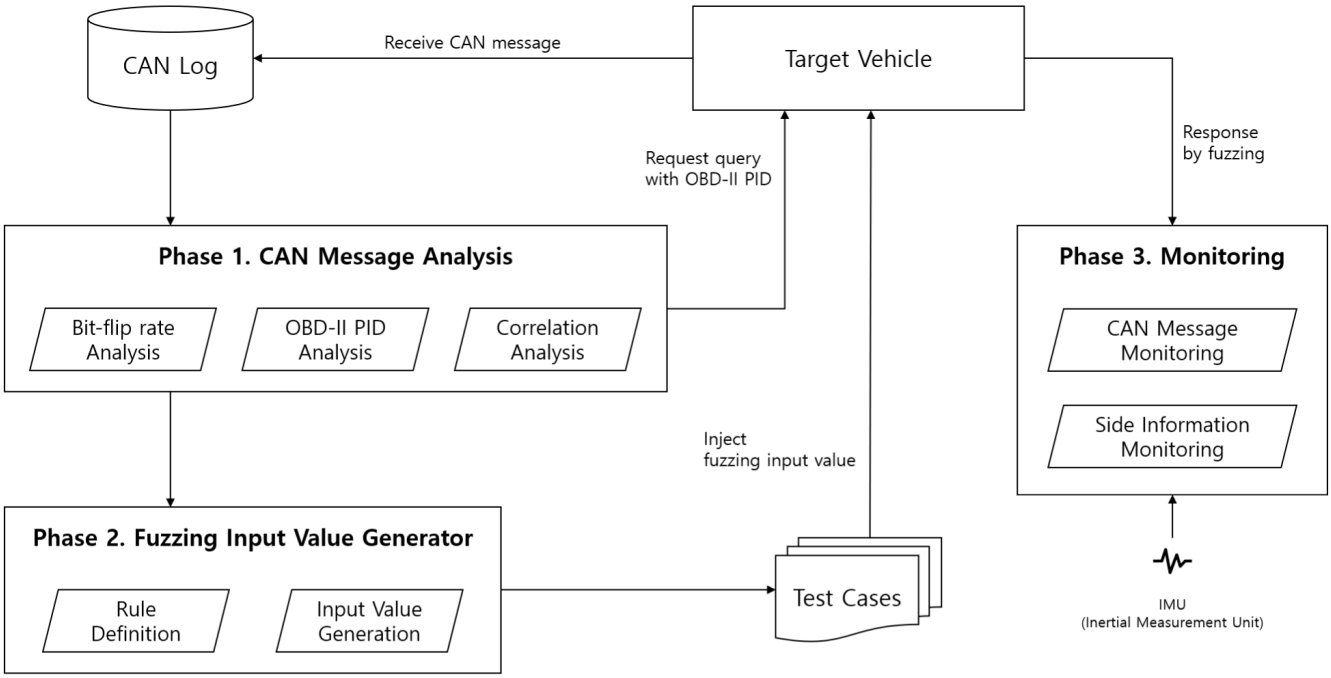


FIGURE 2. System design of our structure-aware CAN fuzzing system.

- CAN fuzzing input value injection: The generated input values are injected into CAN by considering the error states of the CAN-bus. Note that bus-off [9] issues occur due to message collisions caused by fuzzing input value injection.

3) PHASE 3: MONITORING

Phase 3 is composed of two parts: CAN message monitoring and side information monitoring.

- CAN message monitoring: After injection of the fuzzing input values, the system monitors violations of signal properties as defined in Phase 2.
- Side information monitoring: After injection of the fuzzing input values, the system monitors incidents of misbehavior in a fuzzing target vehicle caused by the injected input values using the side information obtained from the monitoring sensors installed on the target vehicle (e.g., IMU sensors).

Next, we provide a detailed explanation of each phase.

B. PHASE 1: CAN MESSAGE ANALYSIS

1) BIT-FLIP RATE ANALYSIS

Two existing studies, [13] and [15], analyze the signal of each CAN ID data field based on the bit-flip rate. Our proposed Structure-aware CAN Fuzzing method also adopts the bit-flip rate-based analysis approach. The bit-flip rate is the frequency of changes in each bit position (from 0 to 1, and vice versa) and can be calculated with the following Equation (1):

$$BFR_{c,k} = \frac{1}{n_c} \sum_{i=2}^{n_c} b_{c,k,i} \tag{1}$$

- $BFR_{c,k}$: The bit-flip rate for the k^{th} position in the data field of an arbitrary CAN ID (c).
- n_c: The number of CAN messages with CAN ID (c) in a set of CAN messages that has been logged over a predetermined time (e.g., 10 minutes).
- $b_{c,k,i}$: Among the n_c CAN messages with CAN ID (c), if the values of k^{th} bit position of the i^{th} and $(i-1)^{\text{th}}$ CAN message's data field are the same value, $b_{c,k,i}$ is 1. Otherwise, $b_{c,k,i}$ is 0.

The boundaries of the signals are identified by calculating the bit-flip rate of the CAN data field. Then, the identified signal boundaries are mapped to seven signal types by using the bit-flip rate-based heuristic rules as follows.

- *Unused* field: A signal with bit values of 0 and a corresponding bit-flip rate of 0, and has a size of 4 or 8 bits.
- *Constant* field: A signal with fixed bit values of a specific value that is not 0 and a corresponding bit-flip rate of 0, and has a size of 4 or 8 bits.
- *Multi-value* field: A signal with bit values of only a few constant values and a corresponding bit-flip rate that converges to 0, and has a size of 2 or 4 bits.
- *Sensor* field: A signal whose bit-flip rates decrease from the Least Significant Bit (LSB) to the Most Significant Bit (MSB), and has a size of 4, 8, or 16 bits. In general, the sensor field representing a physical value has a high bit-flip rate on the LSB, as the bit value changes frequently, while the bit-flip rate on the MSB is smaller than on the LSB. Therefore, we can infer that the field where the bit-flip rate decreases from the MSB to the LSB is related to the physical value generated by the sensor in the vehicle.

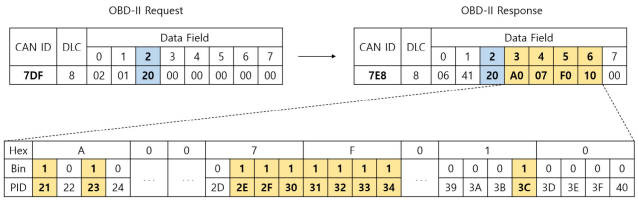


FIGURE 3. OBD-II PID diagnostic query request and response.

- *Counter* field: A signal with a 2, 4, or 8-bit value, and bit-flip rates decrease by 1/2 from the LSB to the MSB.
- *Checksum* field: A signal whose bit-flip rates of 2, 4, or 8-bit values are distributed between 0.3 and 0.6.
- *Undefined* field: Any signal that is not classified into the above six fields.

2) OBD-II PID ANALYSIS

a: FINDING SUPPORTED PIDs

In this step, we adopt the existing ACA-proposed method [10], [11], to find supported PIDs provided by a target vehicle. The OBD-II PID diagnostic query is requested through the OBD-II port, and supported PIDs are checked using response query data. When PID request queries with a CAN ID of $0 \times 7DF$ are transmitted to the target vehicle after setting the PID codes (e.g., 0×00, 0×20, 0×40, 0×60, 0×80, $0 \times A0$, $0 \times C0$), we can check the supported PIDs represented in the 4-byte return values of the PID requests. For example, as shown in Fig. 3, if the PID code of 0×20 is injected into a target vehicle as a PID request query, each bit position of the 4-byte return values indicates a supported PID of the vehicle. As such, the supported PIDs are 0×21 (Distance traveled with malfunction indicator lamp (MIL) on), 0×23 (Fuel Rail Gauge Pressure), $0 \times 2E$ (Commanded evaporative purge), $0 \times 2F$ (Fuel Tank Level Input), 0×30 (Warm-ups since codes cleared), 0×31 (Distance traveled since codes cleared), 0×32 (EVAP System Vapor Pressure), 0×33 (Absolute Barometric Pressure), 0×34 (Oxygen Sensor 1), and $0 \times 3C$ (Catalyst Temperature: Bank1, Sensor 1).

b: CAN ID-PID MATCHING

In order to analyze the relationship between the CAN IDs and the PIDs of a test vehicle, we adopted the existing methods proposed in [10] and [11] for the following matching step process. First, the diagnostic query with a PID is periodically requested from a test vehicle, and the corresponding diagnostic response packets along with normal CAN packets are recorded during a pre-defined time called a window time. Then, by comparing the diagnostic response packets and data field of the normal CAN packet recorded during the window time, the frequencies of CAN IDs that have the same value as the diagnostic response results are counted. By repeatedly performing the above described processes, the CAN ID-PID matching step can identify the CAN ID with the highest number of counts in all time windows as being highly correlated with the PID used in the diagnostic queries. For details on the CAN ID-PID matching algorithm, please refer to [10] and [11].

3) CORRELATION ANALYSIS

In order to identify correlating CAN IDs, we analyze the data field of each CAN ID. Since ECUs in a vehicle broadcast CAN messages to share a specific vehicle status, some *Sensor* signals representing the vehicle's physical status may have negatively or positively correlate. Therefore, we identify CAN IDs that have a high correlation with each other via a correlation analysis between all *Sensor* fields.

C. PHASE 2: FUZZING INPUT VALUE GENERATION

1) CAN FUZZING RULE DEFINITION

In this section, we define the Structure-aware CAN Fuzzing rules used to check a real vehicle that is misbehaving—a problem that could be originating from SW implementation errors or CAN DBC design errors. The rules include the following two violations.

- Signal violation: Signal violations generate fuzzing input values that include abnormal fields and consist of two violations.
  - U.C.M. violation: U.C.M. violations generate fuzzing input values that include abnormal *Unused*, *Constant*, and *Multi-value* fields by setting abnormal signal values that have not appeared in these fields. Through this rule, it is possible to check the misbehavior caused by abnormal values in *Unused*, *Constant*, and *Multi-value* fields that are not originally used or where only certain specific values are used. For efficient U.C.M. violation input value injections, a value in the form of 2^k is injected into the target field where the U.C.M. violation will be located. k is an integer that increases by 1 from 0 to l. (l is the bit length of the target field.)
  - Sensor violation: Sensor violations generate fuzzing input values that include an abnormal *Sensor* field that is greater than the maximum value or less than the minimum value of the *Sensor* field to observe the misbehavior causing abnormal *Sensor* signals; the maximum value and the minimum value of the *Sensor* field are obtained from the CAN log.
- Field violation: Field violations generate fuzzing input values that are set to the same value at each byte position of the data field to observe misbehavior caused by the abnormal structure of a CAN message that has not been inferred by bit-flip rates. For efficient field violation input value injection, a value in the form 2^k is injected into each byte of the CAN data field. k is an integer that increases by 1 from 0 to 7.

2) CAN FUZZING INPUT VALUE GENERATION

a: SIGNAL VIOLATION INPUT

Before input values corresponding to a U.C.M. violation or a sensor violation are generated, the target CAN message

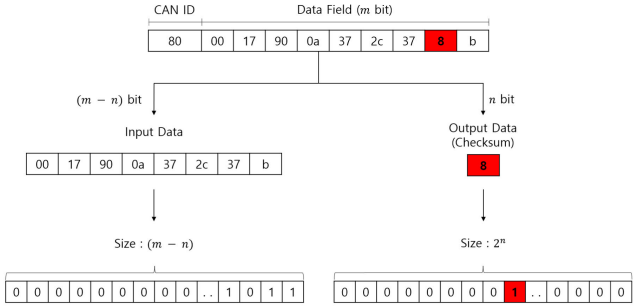


FIGURE 4. Pre-processing of the CAN log for checksum inference.

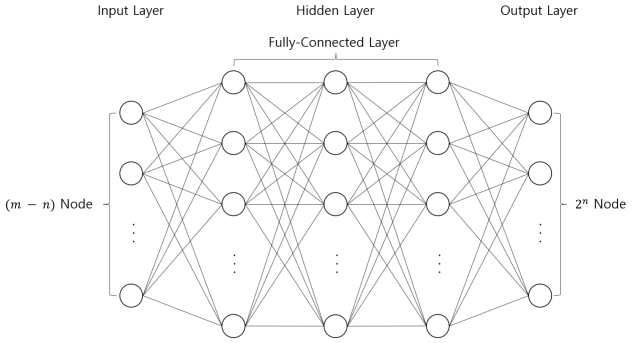


FIGURE 5. DNN model for checksum inference.

should first be checked to see whether it has a *Checksum* field in the data field. If a *Checksum* field exists in the target CAN message, the checksum of the injected input value should be set. According to [4], if a CAN message is injected into the vehicle without consideration to the checksum, the corresponding CAN message may be ignored because of the checksum verification process. In addition, some vehicle manufacturers use private keys stored in an ECU when generating the checksum.

In the proposed method, there are two ways that the checksum value can be obtained. These processes are known as defined checksum and inferred checksum. If the input value already exists in the CAN log, the checksum does not need to be inferred. In this case, the checksum is set to a defined value in the fuzzing input value that is equal to the existing CAN log. Otherwise, to set the checksum for new input values without reverse engineering the ECU, the checksum value is inferred using a deep learning method. The proposed checksum inferring process is described below.

- Pre-processing the CAN log for checksum inference: After extracting all CAN IDs that have values inferred by the *Checksum* field in the data field, values of the *Checksum* field are set as the output of the checksum inferring process, and the remaining values are set as input. Then, the input is converted from hexadecimal to binary, and the output is encoded by a one-hot vector. For example, if the data field is m-bit and the *Checksum* field is n-bit, the input data, as binary data, consists of an array of size $(m - n)$, and the output data consists of an array of size 2^n. The reason why the number of neurons in the output layers is 2^n is that the checksum value can be any value in a range of numbers that can be represented using the corresponding bit. Fig. 4 shows the process of encoding the CAN data field with a one-hot vector. If the *Checksum* field is 4 bits and the corresponding value is 8, a one-hot-vector with an array of size 2^4 sets 1 in the 8^{th} position of the array and the other positions of the array to 0.
- Designing a model for checksum inference: A checksum inference model is designed to adopt a deep neural network (DNN) [24]. For example, if the *Checksum* field is n-bit, the input layer consists of $(m - n)$ nodes and accepts the input data that has been converted to binary data. The output layer consists of 2^n nodes and takes the output data encoded with a one-hot vector. The hidden layer consists of a fully connected layer, and the design of the DNN model is shown in Fig. 5.

We use precision, recall, and F1-score to verify the performance of the model that infers checksum values. Experiments and evaluations of the model are described in Section IV in detail.

b: FIELD VIOLATION INPUT

In a field violation, input values are generated regardless of the *Checksum* field because the field violation ignores the identified signal boundaries.

3) CAN FUZZING INPUT VALUE INJECTION

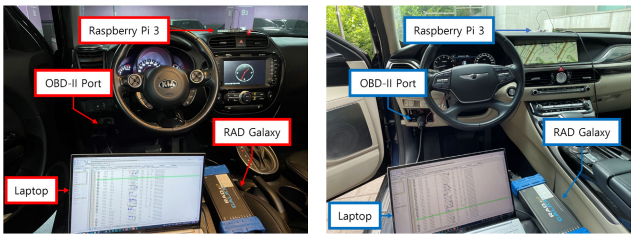
In this section, CAN messages are generated according to the Structure-aware CAN Fuzzing rules, and they are injected into a fuzzing target vehicle. When CAN messages are injected, the transmitter node may enter bus-off mode, which means that the node is unable to participate in the CAN-bus within a pre-defined time (i.e., bus-off recovery time) due to message collisions resulting from injected CAN messages and CAN messages generated from the test vehicle. Owing to this, our fuzzing input injection process is designed to reset the CAN controller of the transmitter when it detects the bus-off state and then performs the fuzzing process without any delay originating from bus-off recovery. In addition, if the fuzzing input injection process finds any fuzzing input that can trigger misbehavior in the target vehicle, this input is injected several times. We can then confirm through experiments that the target vehicle is responded to in the same way according to the fuzzing input, not the bus-off event.

D. PHASE 3: MONITORING

1) CAN MESSAGE MONITORING

In order to monitor misbehavior in a vehicle caused by CAN fuzzing input, we can monitor changes in the following values.

- CAN ID monitoring: Check for existence of invalid CAN IDs.



(a) 2014 Kia Soul (b) 2018 Genesis EQ900

FIGURE 6. Experiment environment for Structure-aware CAN Fuzzing.

- CAN DLC monitoring: Check for changes in DLC values of any CAN ID.
- U.C.M. violation monitoring: Check for increases in the bit-flip rates of *Unused*, *Constant*, and *Multi-value* fields defined in Phase 1 that exceed the U.C.M. violation threshold (e.g., 0.1).
- Sensor violation monitoring: Check whether or not the values of the *Sensor* field are outside the maximum or minimum values.
- Correlation monitoring: Check for changes in the correlation between the *Sensor* field values over a certain threshold.

2) SIDE INFORMATION MONITORING

An IMU sensor, an inertial measurement device that integrates accelerometers and gyroscopes, is used to monitor misbehavior caused by CAN fuzzing. The accelerometer measures velocity changes, and the gyroscope measures angular velocity for three axes: pitch (x-axis), roll (y-axis), and yaw (z-axis).

IV. EXPERIMENT AND EVALUATION

In this section, we evaluate the proposed Structure-aware CAN Fuzzing system on two different vehicles to consider the environment of the functions provided in a range of vehicles; According to the year of production and the price of the vehicle, the functions provided in each vehicle are different. *VehicleA* is a 2014 Kia Soul, a subcompact crossover SUV,[7] and *VehicleB* is a 2018 Genesis EQ900, a premium luxury sedan.[8] To send and receive CAN messages, we used a CAN monitoring tool called RAD GALAXY as shown in Fig. 6. We also used the Raspberry Pi 3 with an IMU sensor (e.g., MPU-6050) to test the Side Information Monitoring. The CAN monitoring tool and the IMU sensor collect data separately. RAD GALAXY collects CAN messages from the target vehicle, and Raspberry Pi collects the vehicle's vibration through IMU sensors. The data collected from the two devices helps monitor the vehicle's state transitions triggered by fuzzing input injection.

A. CAN MESSAGE ANALYSIS

1) BIT-FLIP RATE ANALYSIS

In order to analyze the signal of each CAN ID data field, we collected CAN logs of varying conditions (e.g., stop, normal driving, and driving with various vehicle functions operating). Fig. 7 shows the bit-flip rate analysis results and representative CAN ID signals in the two vehicles.

TABLE 1. Supported PIDs of *VehicleA* and *VehicleB*.

PID (Hex)	Supported PIDs of $VehicleA$ (Hex)	Supported PIDs of $VehicleB$ (Hex)
00	01, 03, 04, 05, 06, 07, 0B, 0C, 0E, 0F, 11, 13, 15, 1C, 1F	01, 04, 05, 0C, 0D, 11, 1C, 1F
20	21, 23, 2E, 2F, 30, 31, 32, 33, 34, 3C	21, 2E, 2F, 30, 31, 32, 33, 34, 38, 3C, 3D, 3E, 3F
40	41, 42, 43, 44, 45, 46, 47, 49, 4A, 4C, 56	41, 42, 43, 44, 45, 46, 47, 49, 4A, 4C, 4D, 4E, 55, 56, 57, 58, 59

2) OBD-II PID ANALYSIS

a: FINDING SUPPORTED PIDs

In this experiment, we can find supported PIDs using the OBD-II PID query. As shown in Table 1, *VehicleA* supports 37 PIDs and *VehicleB* supports 38 PIDs.

b: CAN ID-PID MATCHING

In this experiment, we identified supported PIDs related to the engine by checking the list of supported PIDs. With this information, we could identify which CAN IDs have a high correlation with engine-related PIDs. Table 2 shows the matching results.

In *VehicleA*, we discovered the following relationships: 1) CAN ID $0 \times A0$ seems to be related to PID 0×04 (Calculated Engine Load) and 0×05 (Engine Coolant Temperature); 2) CAN ID 0×80, $0 \times A0$, and 0×316 seem to be related to PID $0 \times 0C$ (Engine RPM); 3) CAN ID $0 \times A0$, 0×316, and 0×440 seem to be related to PID $0 \times 0D$ (Vehicle Speed); and 4) CAN ID $0 \times A0$ seems to be related to PID 0×11 (Throttle Position). In *VehicleB*, we find that: 1) CAN ID 0×556 seems to be related to PID 0×04 (Calculated Engine Load) and 0×05 (Engine Coolant Temperature); 2) CAN ID 0×316, 0×366, 0×368, 0×374, and 0×556 seem to be related to PID $0 \times 0C$ (Engine RPM); 3) CAN ID 0×112, 0×316, 0×366, 0×368, and 0×556 seem to be related to $0 \times 0D$; and 4) CAN ID 0×556 seems to be related to PID 0×11 (Throttle Position).

We verified our results using the open resource file[9] and confirmed that the identified results are correct except for offset 7 of CAN ID 0×440 in *VehicleA* and offsets 5, 6 of CAN ID 0×368 and offsets 0, 1 of CAN ID 0×374 in *VehicleB*. We organized these findings in Table 2.

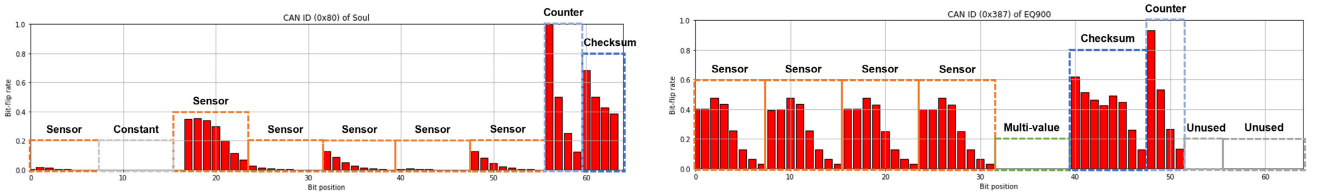
3) CORRELATION ANALYSIS

First, we calculated correlation using the *Sensor* fields of each CAN ID and used the Pearson correlation coefficient to measure the degree and strength of the positive or negative correlation. In this step, we set correlation thresholds

[7][Accessed: Feb. 1, 2022] https://www.kia.com/us/en
[8][Accessed: Feb. 1, 2022] https://www.genesis.com/kr/ko/main.html
[9][Accessed: Feb. 1, 2022] https://github.com/commaai/opendbc

TABLE 2. Matching results between PID codes and CAN IDs related to the engines of *VehicleA* and *VehicleB* (√: CAN signals analyzed by PID code requests, Red tagging: CAN signals provided by the open resource file[9]).

Vehicle	Description (PID code)	CAN ID (Frequency)	Data Field (byte offset)							
			0	1	2	3	4	5	6	7
A	Calculated Engine Load (04)	0xA0 (100ms)	√							
	Engine Coolant Temperature (05)	0xA0 (100ms)		√						
	Engine RPM (0C)	0x80 (10ms)			√	√				
		0xA0 (100ms)			√	√				
		0x316 (10ms)			√	√				
	Vehicle Speed (0D)	0xA0 (100ms)					√			
		0x316 (10ms)							√	
		0x440 (10ms)			√					
	Throttle Position (11)	0xA0 (100ms)						√		
B	Calculated Engine Load (04)	0x556 (100ms)	√							
	Engine Coolant Temperature (05)	0x556 (100ms)		√						
	Engine RPM (0C)	0x316 (10ms)			√	√				
		0x366 (10ms)		√	√					
		0x368 (10ms)						√	√	
		0x374 (10ms)	√	√						
		0x556 (100ms)			√	√				
	Vehicle Speed (0D)	0x112 (10ms)			√					
		0x316 (10ms)							√	
		0x366 (10ms)						√		
		0x368 (10ms)		√						
		0x556 (100ms)					√			
	Throttle Position (11)	0x556 (100ms)						√		



(a) CAN ID 0x80 of the Soul

(b) CAN ID 0x387 of the EQ900

FIGURE 7. Signal classification using bit-flip rate.

to 0.7 and −0.7 to indicate high correlation fields with the target *Sensor* field. If a correlation score of a *Sensor* field with the target *Sensor* field is greater than 0.7 or less than −0.7, we can infer that the *Sensor* field is highly correlated with the target *Sensor* field. Table 3 shows the results of our correlation analysis, in which the *Sensor* fields related to the engine were set to the target *Sensor* fields.

B. FUZZING INPUT VALUE GENERATOR

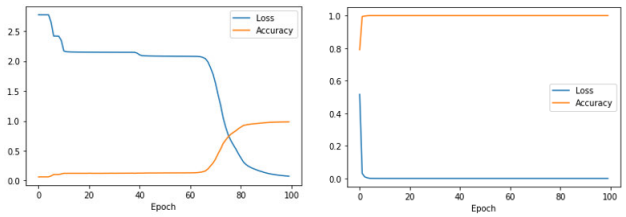
1) INPUT VALUE GENERATION AND CHECKSUM INFERENCE

In this section, we generated fuzzing input values for each CAN ID related to the engine using signal violation (i.e., U.C.M. violation and sensor violation) and field violation rules as defined in Section III.

While generating the fuzzing input values based on a signal violation, we must check whether or not a *Checksum* field exists. In our engine-related fuzzing experiments, we found that the *Checksum* field exists in the data field of CAN ID 0×80 of *VehicleA* and CAN ID 0×112 of *VehicleB*. Thus, checksum values of these fuzzing input values, which are related to CAN ID 0×80 of *VehicleA* and CAN ID 0×112 of *VehicleB*, were inferred using our checksum inference model. We confirmed that the identified checksum fields of the two CAN IDs are correct by comparing them with the open resource file[9]. The hyperparameters of our checksum inference model are listed in Table 4, and the input and output values for the checksum inference model become processed as follows. For example, since CAN ID 0×80 is expected to have a *Checksum* field of 4 bits, the input layer for the checksum inference model consists of 60 nodes and the output layer consists of 2^4 nodes. Likewise, since CAN ID 0×112 is expected to have a *Checksum* field of 2 bits, the input layer for the checksum inference model consists of 62 nodes and the output layer consists of 2^2 nodes. Then, in order to train and

TABLE 3. Sensor field value-based CAN ID correlation.

Vehicle	Description (PID code)	CAN ID (byte offset)	CAN ID with high correlation (byte offset)
A	Calculated Engine Load (04)	0xA0(0)	0xA0(5), 0xA1(1, 4)
	Engine Coolant Temperature (05)	0xA0(1)	0xA0(4), 0x18F(1)
	Engine RPM (0C)	0x80(2, 3)	0x80(0), 0x260(2, 5), 0x316(2, 3, 6), 0x329(6), 0x43F(6), 0x440(2, 6), 0x545(1)
		0xA0(2, 3)	X
		0x316(2, 3)	0x80(0, 2, 3), 0x260(2, 5), 0x316(6), 0x329(6), 0x43F(6), 0x440(2, 6), 0x545(1)
	Vehicle Speed (0D)	0xA0(4)	0x18F(1), 0xA0(1)
		0x316(6)	0x18F(5), 0x370(2), 0x43F(6), 0x440(2, 6)
		0x440(2, 7)	0x18F(5), 0x316(6), 0x43F(6), 0x440(6)
	Throttle Position (11)	0xA0(5)	0xA0(0), 0xA1(4)
B	Calculated Engine Load (04)	0x556(0)	0x556(0), 0x557(4, 5), 0x5CE(0, 1, 4, 5)
	Engine Coolant Temperature (05)	0x556(1)	X
	Engine RPM (0C)	0x316(2, 3)	0x111(5, 6), 0x112(5), 0x260(5), 0x316(5), 0x370(0, 1)
		0x366(1, 2)	0x260(5), 0x366(4), 0x367(6, 7), 0x368(6)
		0x368(5, 6)	0x260(5), 0x366(1, 2, 5), 0x367(6, 7), 0x368(4)
		0x374(0, 1)	0x111(5, 6), 0x112(5), 0x260(5), 0x316(2, 3, 5)
		0x556(2, 3)	X
	Vehicle Speed (0D)	0x112(2, 7)	0x111(5, 6), 0x112(5)
		0x316(6)	0x111(5, 6), 0x112(2, 5)
		0x366(5)	0x367(6, 7), 0x368(1, 6)
		0x368(1)	0x366(5, 6), 0x367(6, 7)
		0x556(4)	0x557(1)
	Throttle Position (11)	0x556(5)	0x556(0), 0x557(4, 5), 0x5CE(0, 1, 4, 5)



(a) CAN ID 0x80 (4 bits) (b) CAN ID 0x112 (2 bits)

FIGURE 8. Performance of our model for checksum inference.

TABLE 4. Hyperparameters of our model.

Hyperparameters	CAN ID 0x80 (4 bits)	CAN ID 0x112 (2 bits)
Number of nodes in hidden layer	120	80
Number of hidden layers	3	3
Batch size	128	128
Epochs	100	100
Activation function	ReLU, Softmax	ReLU, Softmax
Optimizer	Adam	Adam
Learning rate	0.001	0.001
Loss function	categorical_crossentropy	categorical_crossentropy

TABLE 5. Evaluation metrics for our model.

CAN ID	Precision	Recall	F1-score
0x80	98%	98%	98%
0x112	99%	99%	99%

evaluate the checksum inference model, we dedicated 70% of the CAN log for training, 20% for validation, and 10% for testing.

As shown in Fig. 8, our model achieves an accuracy rate over 98% and converges the loss to 0.01 in the training process for both CAN IDs 0×80 and 0×112. From Table 5, we can observe that the precision, recall, and F1-score for CAN IDs 0×80 and 0×112 are about 98% and 99%, respectively.

- Precision

$$= \frac{TruePositives}{TruePositives + FalsePositives} \tag{2}$$

- Recall

$$= \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3}$$

- F1-Score

$$= 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

C. MONITORING RESULTS

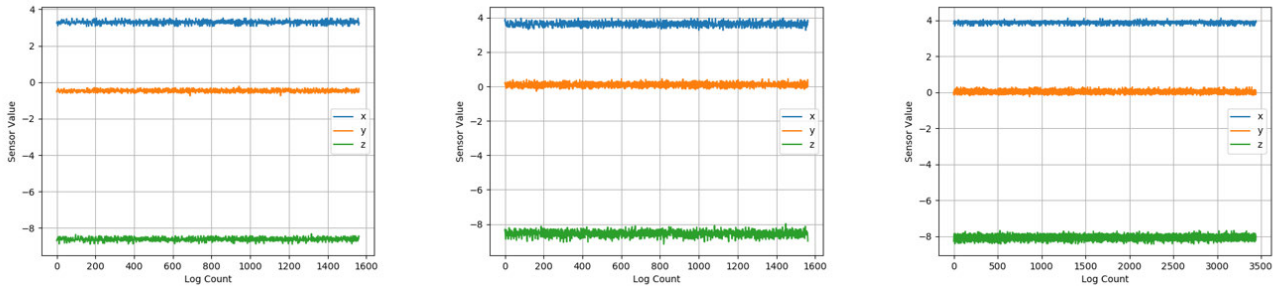
After injecting CAN fuzzing input values, the corresponding results are monitored via CAN logs and side information.

1) CAN MESSAGE MONITORING

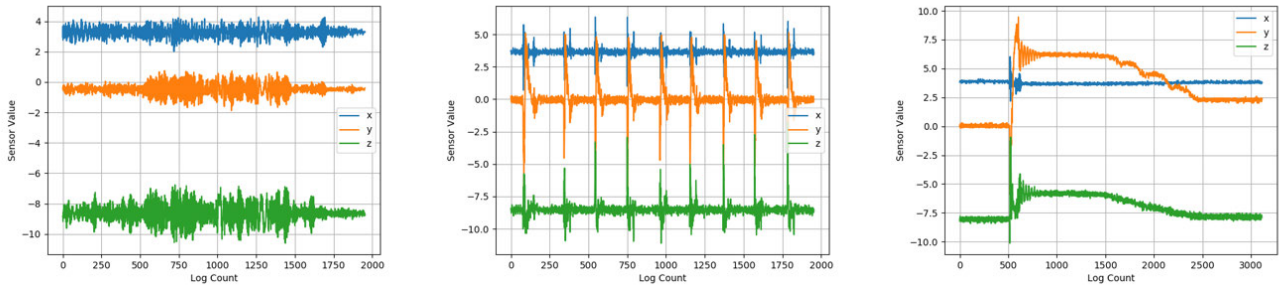
Since vehicle misbehavior can be reflected in CAN messages, these are monitored using the following rules: 1) CAN ID monitoring, 2) CAN DLC monitoring, 3) U.C.M. monitoring,

(a) RPM instrument panel (left) and Movement of steering (right) in the Soul (b) Activation of AEB (left) and Movement of steering (right) in the EQ900

FIGURE 9. Vehicle misbehavior caused by CAN fuzzing.



(a) Normal state of the Soul before injecting a U.C.M. violation with CAN ID 0x370 (b) Normal state of the Soul before injecting a field violation with CAN ID 0x164 (c) Normal state of the EQ900 before injecting a U.C.M. violation with CAN ID 0x340



(d) Fuzzing state of the Soul using a U.C.M. violation with CAN ID 0x370 (e) Fuzzing state of the Soul using a field violation with CAN ID 0x164 (f) Fuzzing state of the EQ900 using a U.C.M. violation with CAN ID 0x340

FIGURE 10. Side information monitoring results.

4) sensor monitoring, and 5) correlation monitoring. When monitoring CAN messages according to these five rules, we observed no noticeable changes.

2) SIDE INFORMATION MONITORING

In this step, we compared sensor values obtained from each vehicle's normal state without CAN fuzzing, which was done using sensor values obtained from each vehicle's fuzzing state with CAN fuzzing. By monitoring side information obtained from engine-related CAN fuzzing experiments, we were able to detect misbehavior caused by CAN fuzzing in the steering control in both vehicles, engine acceleration in *VehicleA*, and brake control in *VehicleB*.

Fig. 10 shows CAN IDs that have abnormal side information. In the figure, blue, orange, and green indicate x, y, and z axes of the IMU sensor value, respectively. When comparing graph (a) Normal state of the Soul with (d) Fuzzing state of the Soul using a U.C.M. violation with CAN ID 0×370, the amplitude increases for IMU sensor values found in (d). This vibration checked by the IMU sensor means that the test vehicle's engine accelerated. Also, we confirmed that the RPM of the dashboard changed as shown in Fig. 9 (a) RPM instrument panel. In addition, when the CAN fuzzing input value for CAN ID 0×316 was injected, the RPM of the dashboard changed. When comparing graph (b) Normal state of the Soul with (e) Fuzzing state of the Soul using the field violation role for CAN ID 0×164, a periodic change of IMU sensor values was found in (e). This periodical change is caused by the vehicle's steering control that actuates steering by $5 \sim 10°$ to the right as shown in Fig. 9 (a) Movement of steering in the Soul. Likewise, the difference between graph (c) Normal state of the EQ900 and (f) Fuzzing state of the EQ900 using

TABLE 6. Comparison of number of CAN fuzzing input values (engine-related CAN IDs: *VehicleA* (0 × 80, 0xA0, 0 × 316, 0 × 440), *VehicleB* (0 × 112, 0 × 316, 0 × 366, 0 × 368, 0 × 374, 0 × 556)).

		Brute force CAN fuzzing [18]–[23]		Structure-aware CAN Fuzzing									
		$VehicleA$	$VehicleB$	$VehicleA$				$VehicleB$					
		Engine-related CAN IDs		0x80	0xA0	0x316	0x440	0x112	0x316	0x366	0x368	0x374	0x556
Signal Violations	U.C.M. Violations	2^{64}		8	20	16	28	16	8	8	32	32	16
	Sensor Violations			12	8	10	6	10	12	10	6	2	10
Field Violations				8	8	8	8	8	8	8	8	8	8
Total number of cases		$2^{64} \times 4$	$2^{64} \times 6$	$140 < 2^8$				$226 < 2^8$					

a U.C.M. violation with CAN ID 0 × 340 shows the steering control for *VehicleB* as shown in Fig. 9 (b) Movement of steering in the EQ900.

Furthermore, brake control was monitored as shown in Fig. 9 (b) AEB, in which the autonomous emergency braking (AEB) is actuated by CAN ID 0 × 48A. Although the brake control was not monitored via the IMU sensor in the vehicle's stop status, it can be monitored during the vehicle's driving status.

D. NUMBER OF CAN FUZZING INPUT VALUES

As shown in Table 6, we compared the number of fuzzing input values between existing brute force CAN fuzzing methods and our proposed Structure-aware CAN Fuzzing method. For a fair comparison, we used CAN IDs that we identified as being related to the engine in the test vehicles (e.g., CAN ID 0 × 80, 0xA0, 0 × 316, and 0 × 440 in *VehicleA*, and CAN ID 0 × 112, 0 × 316, 0 × 366, 0 × 368, 0 × 374, and 0 × 556 in *VehicleB*). The process of calculating the number of fuzzing input values for each CAN ID is as follows.

- Number of signal violations
  - Number of U.C.M. violations: If the *Unused*, *Constant*, and *Multi-value* are 4 bits, the number of input values for each field is 4 because each field is injected in units of 2^k, meaning that k increases by 1 from 0 to 4. By using this fact, the number of fuzzing input values related to U.C.M. violations can be calculated in the following way: if the number of *Unused*, *Constant*, and *Multi-value* fields is n_1 in the data field of the specific CAN ID, the corresponding number of fuzzing input values is $(4 \times n_1)$. Therefore, since the number of *Unused*, *Constant*, and *Multi-value* fields for CAN ID 0 × 80, 0xA0, 0 × 316, and 0 × 440 in *VehicleA* is 2, 5, 4, and 7, respectively, the number of fuzzing input values is 8, 20, 16, and 28, respectively. Likewise, since the number of *Unused*, *Constant*, and *Multi-value* fields for CAN ID 0 × 112, 0 × 316, 0 × 366, 0 × 368, 0 × 374, and 0 × 556 in *VehicleB* is 4, 2, 2, 8, 8, and 4, respectively, the number of fuzzing input values is 16, 8, 8, 32, 32, and 16, respectively.
  - Number of sensor violations: Sensor violation fuzzing generates two input values per *Sensor* field. One is greater than the maximum value of the *Sensor* field, while the other is less than the minimum value. By using this fact, the number of fuzzing input values related to sensor violations can be calculated in the following way: if the number of *Sensor* fields is n_2 in the data field of the specific CAN ID, the corresponding number of fuzzing input values is $(2 \times n_2)$. Therefore, since the number of *Sensor* fields for CAN ID 0 × 80, 0xA0, 0 × 316, and 0 × 440 in *VehicleA* is 6, 4, 5, and 3, respectively, the number of fuzzing input values is 12, 8, 10, and 6, respectively. Likewise, since the number of *Sensor* fields for CAN ID 0 × 112, 0 × 316, 0 × 366, 0 × 368, 0 × 374, and 0 × 556 in *VehicleB* is 5, 6, 5, 3, 1, and 5, respectively, the number of fuzzing input values is 10, 12, 10, 6, 2, and 10, respectively.
- Number of field violations: After dividing the data field (64 bits) by 8 bits, each field is injected with units of 2^k, such that k increases by 1 from 0 to 7. In short, all eight fields have the same value according to k. Therefore, the number of fuzzing input values for engine-related CAN IDs in both *VehicleA* and *VehicleB* is 8.

As shown in Table 6, the total number of fuzzing input values of our method for *VehicleA* is 140 and 226 for *VehicleB*, which are the summations of the number of fuzzing input values for engine-related CAN IDs. We confirmed that the total number of fuzzing input values in our method is less than 2^8, whereas the number in existing brute force CAN fuzzing methods is greater than 2^{66}. As a result, we confirmed that our method takes less fuzzing time than existing brute force CAN fuzzing methods.

V. LIMITATIONS AND DISCUSSION

The efficiency of our proposed Structure-aware CAN Fuzzing system was validated on a 2014 Kia Soul and a 2018 Genesis EQ900. However, this study still faces some limitations.

First, the proposed Structure-aware CAN Fuzzing system was validated on vehicles produced by the same manufacturer. Therefore, in order to extensively validate our method, we must evaluate our method using vehicles from other makers in future research. Additionally, if the manufacturer offers vehicles equipped with an intrusion detection system (IDS), new fuzzing methods should be studied because it might be difficult to inject fuzzing input values into the vehicle. Second, the proposed Structure-aware CAN Fuzzing system does not contain feedback-directed fuzzing that is generally included in software fuzzing. When generating fuzzing input values, there are cases where the vehicles do not respond even if the checksum values of the fuzzing input values are correctly inferred. We think that the root of these issues may

lie in the contexts of the CAN-bus. For example, a correct counter value may need to be set if a CAN message has a counter field. Third, we use only the IMU sensor to obtain side information. However, it is possible to obtain additional side information from the vehicle's black box video data, sound sensor, and so on. By adding such additional sensors, we can monitor vehicular misbehavior more specifically. Last, we did not consider unusual situations (e.g., a collision event or a debug mode). However, our method generates unusual fuzzing input values by modifying the unused field and constant field values, which are known to contain a fixed value or some multi-values during normal driving conditions. Even if modifying the values included in these fields could solve unusual situations to some extent, more tests must be performed to cover unusual situations.

In future work, we will improve our Structure-aware CAN Fuzzing method by adopting feedback-directed fuzzing input generation, which enables us to control the vehicle more critically by adjusting the field, such as the counter field of the control message. In addition, a variety of sensors other than the IMU sensor will be adopted to monitor vehicular misbehavior, and more fuzzing tests will be conducted on vehicles made by various manufacturers.

VI. CONCLUSION

With the increase in the number of connected and autonomous vehicles (CAVs), cyber attacks on such vehicles are increasing. To deal with vehicular cyber attacks, ECU reverse engineering and CAN fuzzing have been studied to analyze ECU vulnerabilities. However, ECU reverse engineering needs physical access to a target ECU, and existing CAN fuzzing techniques require non-negligible analysis time to randomly transmit fuzzing input values to the CAN-bus. Additionally, monitoring responses corresponding to random fuzzing input values is notoriously difficult. To overcome the limitations of existing analysis methods, we proposed our Structure-aware CAN Fuzzing system, which considers the structure of CAN messages. Additionally, to our knowledge, this is the first method of its kind developed for the Structure-aware CAN Fuzzing method. In comparison with brute force CAN fuzzing, the proposed Structure-aware CAN Fuzzing system can minimize fuzzing time and monitor the response results of CAN fuzzing. Our method was validated on real vehicles, a 2014 Kia Soul and a 2018 Genesis EQ900. In future work, we will apply our method to various real vehicles and validate efficiency.

REFERENCES

[1] *Road Vehicles—Controller Area Network*, Standard 11898-1: 2003, International Organization for Standardization, Geneva, Switzerland, 2003.
[2] 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 *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 447–462.
[3] 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 *Proc. USENIX Secur. Symp.*, San Francisco, CA, USA, vol. 4, 2011, p. 2021.
[4] C. Valasek and C. Miller, ''Adventures in automotive networks and control units,'' *Def Con*, vol. 21, nos. 260–264, pp. 15–31, 2013.
[5] 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 Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 993–1006, Sep. 2014.
[6] C. Miller and C. Valasek, ''Remote exploitation of an unaltered passenger vehicle,'' *Black Hat USA*, vol. 2015, no. S91, Aug. 2015.
[7] S. Nie, L. Liu, and Y. Du, ''Free-fall: Hacking Tesla from wireless to CAN bus,'' *Briefing, Black Hat USA*, vol. 25, pp. 1–16, Jul. 2017.
[8] S. Nie, L. Liu, Y. Du, and W. Zhang, ''Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars,'' in *Proc. Black Hat USA*, vol. 1. Las Vegas, NV, USA, Aug. 2018, pp. 1–19.
[9] R. Bosch, ''Can specification version 2.0,'' *Rober Bousch GmbH, Postfach*, vol. 300240, p. 72, Sep. 1991.
[10] T. U. Kang, H. M. Song, S. Jeong, and H. K. Kim, ''Automated reverse engineering and attack for CAN using OBD-II,'' in *Proc. IEEE 88th Veh. Technol. Conf. (VTC-Fall)*, Aug. 2018, pp. 1–7.
[11] H. M. Song and H. K. Kim, ''Discovering CAN specification using on-board diagnostics,'' *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 38, no. 3, pp. 93–103, Jun. 2020.
[12] M. Markovitz and A. Wool, ''Field classification, modeling and anomaly detection in unknown CAN bus networks,'' *Veh. Commun.*, vol. 9, pp. 43–52, Jul. 2017.
[13] M. Marchetti and D. Stabili, ''Read: Reverse engineering of automotive data frames,'' *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 4, pp. 1083–1097, 2018.
[14] M. Verma, R. Bridges, and S. Hollifield, ''ACTT: Automotive CAN tokenization and translation,'' in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2018, pp. 278–283.
[15] M. D. Pesé, T. Stacer, C. A. Campos, E. Newberry, D. Chen, and K. G. Shin, ''LibreCAN: Automated CAN message translator,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 2283–2300.
[16] D. Frassinelli, S. Park, and S. Nurnberger, ''I know where you parked last summer: Automated reverse engineering and privacy analysis of modern cars,'' in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1401–1415.
[17] C. Young, J. Svoboda, and J. Zambreno, ''Towards reverse engineering controller area network messages using machine learning,'' in *Proc. IEEE 6th World Forum Internet Things (WF-IoT)*, Jun. 2020, pp. 1–6.
[18] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim, ''Fuzzing can packets into automobiles,'' in *Proc. IEEE 29th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2015, pp. 817–821.
[19] D. S. Fowler, J. Bryans, S. A. Shaikh, and P. Wooderson, ''Fuzz testing for automotive cyber-security,'' in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2018, pp. 239–246.
[20] 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 *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2019, pp. 1–8.
[21] A.-I. Radu and F. D. Garcia, ''Grey-box analysis and fuzzing of automotive electronic components via control-flow graph extraction,'' in *Proc. Com-put. Sci. Cars Symp.*, Dec. 2020, pp. 1–11.
[22] T. Werquin, R. Hubrechtsen, A. Thangarajan, F. Piessens, and J. T. Muehlberg, ''Automated fuzzing of automotive control units,'' 2021, *arXiv:2102.12345*.
[23] M. A. Mokhadder, S. Bayan, and U. Mohammad, ''An intelligent approach to reverse engineer CAN messages in automotive systems,'' in *Proc. IEEE Int. Conf. Electro Inf. Technol. (EIT)*, May 2021, pp. 1–7.
[24] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, ''Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,'' *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Oct. 2012.

HYUNGHOON KIM received the B.S. degree in computer engineering from Hallym University, Chuncheon, South Korea, in 2019, and the M.S. degree from the Graduate School of Software, Soongsil University, Seoul, in 2021, where he is currently pursuing the Ph.D. degree. His research interests include applied cryptography, security and privacy for ad-hoc networks, and IoT/CPS security.

YEONSEON JEONG received the B.S. degree in computer engineering from Hallym University, Chuncheon, South Korea, in 2021. She is currently pursuing the M.S. degree in information security with Korea University, Seoul. Her research interests include applied cryptography, security and privacy for ad-hoc networks, and IoT/CPS security.

WONSUK CHOI received the B.S. degree in mathematics from the University of Seoul, Seoul, South Korea, in 2008, and the M.S. and Ph.D. degrees in information security from Korea University, Seoul, in 2013 and 2018, respectively. He was a Postdoctoral Researcher at the Graduate School of Information Security, Korea University. He joined as an Assistant Professor with the Division of IT Convergence Engineering, Hansung University, Seoul, in 2020. His research interests include security for body area networks, usable security, applied cryptography, and smart car security.

DOON HOON LEE (Member, IEEE) received the B.S. degree from the Department of Economics, Korea University, Seoul, South Korea, in 1985, and the M.S. and Ph.D. degrees in computer science from the University of Oklahoma, Norman, OK, USA, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University. He is currently a Professor and the Director of the Graduate School of Information Security, Korea University. His research interests include cryptographic protocol, applied cryptography, functional encryption, software protection, mobile security, vehicle security, and ubiquitous sensor network (USN) security.

HYO JIN JO (Member, IEEE) received the B.S. degree in industrial engineering and the Ph.D. degree in information security from Korea University, Seoul, South Korea, in 2009 and 2016, respectively. He was a Postdoctoral Researcher at the Department of Computer and Information Systems, University of Pennsylvania, Philadelphia, PA, USA, from 2016 to 2018. Currently, he is an Associate Professor with the School of Software, Soongsil University, Seoul. His research interests include applied cryptography and vehicle security.