# Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols

Le Yu, Yangyang Liu, Pengfei Jing, Xiapu Luo, Lei Xue, and Kaifa Zhao, *The Hong Kong Polytechnic University;* Yajin Zhou, *Zhejiang University;* Ting Wang, *The Pennsylvania State University;* Guofei Gu, *Texas A&M University;* Sen Nie and Shi Wu, *Tencent Keen Security Lab*

https://www.usenix.org/conference/usenixsecurity22/presentation/yu-le

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

# Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols

Le Yu[1], Yangyang Liu[1], Pengfei Jing[1], Xiapu Luo[1*], Lei Xue[1]
Kaifa Zhao[1], Yajin Zhou[2], Ting Wang[3], Guofei Gu[4], Sen Nie[5], Shi Wu[5]
[1]*The Hong Kong Polytechnic University* [2] *Zhejiang University*
[3]*The Pennsylvania State University* [4]*Texas A&M University* [5]*Tencent Keen Security Lab*

## Abstract

In-vehicle protocols are very important to the security assessment and protection of modern vehicles since they are used in communicating with, accessing, and even manipulating ECUs (Electronic Control Units) that control various vehicle components. Unfortunately, the majority of in-vehicle protocols are proprietary without publicly available documents. Although recent studies proposed methods to reverse engineer the CAN protocol used in the communication among ECUs, they cannot be applied to vehicle diagnostics protocols, which have been widely exploited by attackers to launch remote attacks. In this paper, we propose a novel framework for automatically reverse engineering the diagnostic protocols of vehicles by leveraging professional diagnostic tools. Specifically, we design and develop a new cyber-physical system that uses a set of algorithms to control a programmable robotics arm with the aid of cameras to automatically trigger and capture the messages of diagnostics protocols as well as reverse engineer their formats, semantic meanings, and proprietary formulas required for processing the response messages. We perform a large-scale experiment to evaluate our prototype using 18 real vehicles. It successfully reverse engineers 570 messages (446 for reading sensor values and 124 for controlling components). The experimental results show that our framework achieves high precision in reverse engineering proprietary formulas and obtains much more messages than the prior approach based on app analysis.

## 1 Introduction

In-vehicle protocols are very important to the security assessment and protection of modern vehicles since they are used in the communication between ECUs (e.g., CAN protocol) as well as the accessing and manipulating ECUs (e.g., diagnostic protocols). For example, the communication between ECUs usually follows the ISO 11898 standard [18]. The diagnostic protocols were designed for reading sensor values or controlling ECUs through the OBD port. For example, Keyword Protocol 2000 (*KWP 2000*) [1,5,6] and Unified Diagnostic Services (*UDS*) [8] are most widely used diagnostic protocols. Although these standards define the low-level message formats, the syntactic information and semantic meaning of messages, as well as formulas for encoding the return values, are usually defined by vehicle manufacturers and thus are proprietary without publicly available documents.

Recent studies for reverse engineering in-vehicle protocols focus on analyzing the CAN messages transmitted between ECUs [48,71]. Unfortunately, they cannot be applied to the diagnostics protocols because they neither consider the transmission layer protocol used to transmit high-level messages that may span multiple CAN messages [20] nor recover the proprietary formats and formulas. It is worth noting that diagnostics protocols have been widely exploited to launch various attacks on vehicles. For example, Miller et al. [65] send diagnostic messages through OBD port to kill engine or control fuel gauge of the Ford and Toyota [65]. Other examples include the attacks using diagnostic messages to control Hyundai Sonata and Toyota Camry [9,10], BMW [14], Jeep [66], Volkswagen [50].

CANHunter [83] inspects the apps of vehicle telematics dongles to extract the messages sent by the apps. Although it may collect some messages of diagnostics protocols, we find in §4.6 that the majority of the collected messages belong to the well-documented OBD-II protocols [34] instead of proprietary diagnostics protocols. The reason may be that the telematics apps studied in [83] are mainly designed for normal drivers, and thus they just read ordinary information (e.g., speed, engine RPM) through the OBD-II protocols [34]. Moreover, CANHunter [83] focuses on driving the apps to send request messages *without* reverse engineering the request messages and the formulas required for processing response messages. Therefore, even if a few telematics apps in [83] send and receive messages of proprietary diagnostics protocols, CANHunter does not reverse engineer them.

In this paper, to fill the gap, we propose a novel frame-

[*] The corresponding author.

work named DP-Reverser to reverse engineer vehicle diagnostic protocols automatically. We focus on KWP 2000 [12] and UDS [26] because they have been widely used in most vehicles. For example, UDS, which is derived from KWP 2000, has been used in all new ECUs from tier-1 suppliers and is incorporated into other popular standards such as AUTOSAR [74, 84]. Although the standards define the low-level message formats, the high-level information customized and used by vehicle manufacturers is proprietary. For example, the value and functionality of some fields in the request messages are customized by the manufacturers. Moreover, proprietary formulas are required to transform the data stored in the response messages to meaningful values.

In particular, we observe that the ECUs of almost all modern vehicles can be accessed and even manipulated by certain professional diagnostic tools. They can be either general diagnostic tools that can handle multiple kinds of vehicles e.g., [31] or special ones designed for a specific kind of vehicles e.g., [72]. Moreover, it is not difficult to purchase or access such tools since they are widely used by automobile 4S shops and repair shops. By exploiting this observation, our framework first drives professional diagnostic tools to generate messages of diagnostic protocols and captures them. Then, it uses a set of algorithms to reverse engineer the value and functionality (i.e., semantic information) of target fields in request messages as well as the proprietary formulas required to process response messages.

The design and development of our framework need to address several challenging issues. First, the implementation of diagnostic tools is diverse, which can be roughly divided into three categories, namely professional handheld diagnostic equipments (e.g., [31, 36]), professional diagnostic software running on a host (e.g., [37, 72]), and OBD based telematics apps. Thus, the framework should be general for all existing diagnostic tools, extensible for incorporating new tools, and automated for reverse engineering the diagnostic protocols of a large number of vehicles. Moreover, professional diagnostic tools usually adopt various security techniques to prevent analysis and reverse engineering. For example, the hardware and software of professional handheld diagnostic equipments are usually carefully hardened and well protected so that other programs or apps cannot be installed into them and their firmware cannot be extracted. Second, vehicle manufacturers define proprietary formulas to encode the actual values of ECUs into the data in response messages. Thus, the framework should be able to automatically reverse engineer these proprietary formulas to recover the raw data into meaningful values.

To solve the first challenge, we design and develop a new cyber-physical system to automatically drive diagnostic tools and collect the corresponding request and response messages of diagnostic protocols. Specifically, it uses two cameras to capture UI screenshots of the diagnostic tools and uses *OCR* (Optical Character Recognition) algorithms [23] to extract the sensor values and semantic information from the screenshots. Moreover, it analyzes the UI screenshots to decide the optimal movement strategy to instruct a programmable robotic arm (i.e., robotic clicker) to click the diagnostic tools. In the meantime, it monitors the OBD port to capture all CAN frames and associates them with the screenshots of the diagnostic tools. To tackle the second challenge, we first extract the sensor values from the UI video and associate them with the values of sensor fields that are extracted from the recovered payload of diagnostic messages. Then, we design an improved genetic programming algorithm to infer the proprietary formulas based on the extracted data.

We develop a prototype of DP-Reverser and conduct a large-scale experiment to evaluate it using 18 real vehicles, which come from 14 major manufacturers in six countries. DP-Reverser successfully reverse engineered 570 messages (446 for reading sensor values and 124 for controlling components). Its precision of inferring the proprietary formulas of UDS and KWP 2000 reaches 98.3%, which is much higher than alternative algorithms, including the linear regression (43.8%) and polynomial curve fitting (32.1%). By using 7 kinds of ECU signal values of the OBD-II protocol as the ground truth, we find that the DP-Reverser achieves 100% precision. Moreover, in order to compare the number of proprietary formulas that can be extracted from professional handheld diagnostic equipments and OBD based telematics apps, we developed a new tool to analyze 160 telematics apps. The result shows that only 3 apps include UDS/KWP 2000 formulas. Moreover, we conduct experiments with two real vehicles to compare the information that can be reversed engineered from professional diagnostic tools and that from telematics apps. The results show that much more important information about the diagnostic protocols can be obtained from professional diagnostic tools.

In summary, our major contributions include:

- To the best of our knowledge, we propose the *first* CPS-based framework, entitled DP-Reverser, for automatically reverse engineering vehicle diagnostic protocols from professional diagnostic tools.
- We develop a prototype of DP-Reverser with a focus on two widely used diagnostic protocols (i.e., KWP 2000 and UDS) after tackling several technical challenges, such as coordinating the robotic arm, cameras, and diagnostic tools to efficiently collect and process the messages of diagnostic protocols, reverse engineering the proprietary formulas.
- We conduct a large-scale experiment to evaluate DP-Reverser with 18 vehicles, 4 professional diagnostic tools/software, and 160 telematics apps. We also develop a new tool to analyze telematics apps for comparison. The experimental results show that it can accurately reverse engineer the detailed information of the request and response messages of diagnostic protocols. The source code of DP-Reverser and the new tool for analyzing telematics apps will be released at: https://github.com/yulele/DP-Reverser.

# 2 Threat Model and Background

## 2.1 Threat Model

Fig. 1 shows a model of a modern vehicle [78]. The ECUs of the vehicle are connected through bus systems. They can handle the data collected from sensors or camera [53]. They communicate with each other by using CAN frames [19] and are connected to the gateway. The OBD port was designed for On-board diagnostics (*OBD*), which refers to the vehicle's self-diagnostic and reporting capability [34]. The professional diagnostic tools [31,36,37,72] connected to the OBD port can send diagnostic messages to ECUs and receive response messages to read their values or even manipulate them. Aiming at automatically reverse engineering the request and response messages of diagnostics protocols, we only assume the availability of a target vehicle and a diagnostic tool that works for the vehicle.

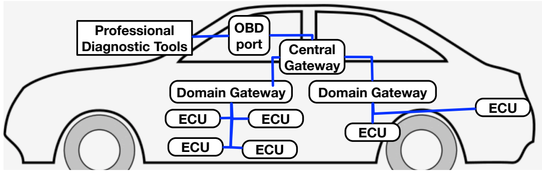


Figure 1: Communication system in the vehicle.

Our system and results can be used by both attackers and defenders. On the one hand, to attack a vehicle, the attacker can first rent a vehicle of the same type and then use our system to reverse engineer the diagnostics protocols supported by the vehicle. After obtaining the detailed information of such messages, the attacker can inject such messages to the target vehicle through vulnerable OBD-II dongles or T-Box or other communication channels [10,65] in order to control the vehicle or cause severe safety consequences. To demonstrate it, we conduct an experiment with 4 real vehicles, including BMW i3, Lexus NX300, Toyota Corolla, and Kia. We find that all these diagnostic messages can trigger certain actions even when the vehicle is running and thus cause safety issues. For example, by sending the diagnostic message `40 05 30 11 00 ... 00`[1] to Toyota Corolla, we can successfully unlock all doors when the vehicle is running. Tab. 13 in Appendix lists other reverse-engineered diagnostic messages used in this experiment. On the other hand, third-party security solution providers can benefit from our research because they need the information of diagnostic messages to filter out injected malicious diagnostic messages [54,79,81].

## 2.2 CAN message.

ECUs connected to the CAN bus use CAN frames to communicate with each other. According to the CAN 2.0 specification, each CAN frame contains one CAN ID and a data field [45]. The CAN ID determines which ECU will process the frame. A lower value of the CAN ID field indicates a higher priority of the corresponding frame. The data field contains up to 8-byte data. To send a long message with more than 8-byte data, the sender relies on the transport/network layer protocols (e.g., ISO 15765-2 [11], VW TP 2.0 [29]) to split the long message into multiple CAN frames.

Table 1: OSI model of KWP 2000, UDS, and OBD-II

| | | | |
|---|---|---|---|
| Application<br>Session | KWP 2000:<br>ISO 14230-3 [5] or 15765-3 [6] | UDS:<br>ISO 14229-2 [28] | OBD-II:<br>ISO-15031 [34] |
| Transport<br>Network | ISO 15765-2 [11]<br>VW TP 2.0 [29] | ISO 15765-2 [11] | ISO 15765-2 [11] |
| Data Link<br>Physical | K-Line: ISO 14230-1(2) [3,4], CAN: ISO 11898 [18] | | |

## 2.3 Diagnostic Protocols

OBD-II, KWP 2000, and UDS are the most popular diagnostic protocols [13, 22]. Tab. 1 shows their OSI model. We do *not* reverse engineer the OBD-II protocol because it is well-documented. The SAE J1979 standard [7] defines the list of services (i.e., modes), the parameter ids (i.e., PIDs) list of each service, the formulas used to parse response messages, and the diagnostic trouble codes [33,35].

### 2.3.1 KWP 2000

Keyword Protocol 2000 is an on-board diagnosis protocol compatible with both K-Line and CAN in-vehicle networking systems [17]. KWP 2000 is widely used in vehicles manufactured and sold in Europe. For K-Line based KWP 2000, the physical, data link, and application layers are compatible with ISO 14230 standard [2,3,5]. For CAN based KWP 2000, the transport and application layer protocols are compatible with ISO 15765-2 [11] and 15765-3 [6]. Volkswagen uses its own transport layer protocol VW TP 2.0 [29].

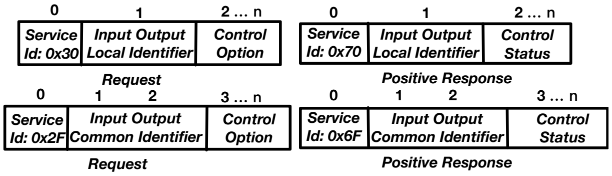


Figure 2: Request and positive response messages of the *input output control by local identifier* service and *input output control by common identifier* service of KWP 2000

Using KWP 2000, diagnostic tools can control an output (actuator) of an electronic system of the vehicle through two

[1] Part of the messages is hidden to avoid being abused. This practice is also applied to other diagnostic messages in this paper.

services: *input output control by local identifier* service (id 0x30) [1, 5] and *input output control by common identifier* service (id 0x2F) [1, 5], and read ECU signal values denoted as **ESV** through the *read data by local identifier* service (id 0x21) [1]. When using KWP 2000 to control an output (actuator) of an electronic system, the diagnostic tool will send a request message to the ECU [1]. If the request message is successfully executed, the ECU will send a positive response message to the diagnostic tool.

Fig. 2 shows the formats of the request message and positive response message. For the *input output control by local identifier* service, the request message contains the service id (0x30), the input output control local identifier (one byte), and control option. The control option field shall include all information required by the ECU's output signal or actuator. We use ECU Control Record (**ECR**) to represent the control option. The positive response message contains the service id, input output local identifier, and control status. For the *input output control by common identifier* service, the format of request and response messages are similar.

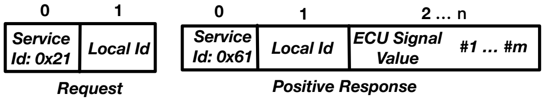


Figure 3: Request and positive response messages of the *read data by local identifier* service of KWP 2000

When using KWP 2000 to read data from the vehicle, the formats of the request and positive response messages are displayed in Fig. 3. The request message contains one local identifier. The positive response message contains 1 to $m$ ECU signal values(i.e., **ESV**s). One **ESV** in the response message describes one kind of data in ECU (e.g., engine RPM, steering angle). It has three bytes. The first byte determines the formula used to calculate the actual ECU signal value. We use $F_{type}$ to represent it. The remaining two bytes are the values used in the formula. We use $X_0$ and $X_1$ to represent them, respectively. By combining these three bytes, the diagnostic device/software calculates the real **ESV**.

*Example.* To turn on/off the light, the diagnostic tool sends the request messages "`30 15 00 40 00`" and "`30 15 00 00 00`" to the Main Body Control ECU, respectively [65]. To obtain the engine RPM, the diagnostic tool sends the request message "`21 07`" to the engine. Then, it receives a response message containing the **ESV** "`01 F1 10`". The formula type is `0x01`. The corresponding formula is $X_0 * X_1/5$. As the value of $X_0$ is `0xF1` (i.e., 241) and the value of $X_1$ is `0x10` (i.e., 16), the actual **ESV** is 771.2 /min (i.e., 242*16/5).

**Target of reverse engineering KWP 2000.** We aim at three kinds of information that is not defined in the standard of KWP 2000, including (1) the value of local id and its semantic information; (2) the semantic information of **ECR**; (3) the corresponding formula used to transform **ESV** in the response message to actual **ESV**.

### 2.3.2 UDS

Unified Diagnostic Services is a diagnostic protocol that combines the K-Line based and CAN based KWP 2000 [26, 27]. Compared with KWP 2000, UDS supports more bus systems, such as CAN, CAN-FD, LIN [12]. The ISO 14229 standard lists 26 kinds of UDS services and the message formats of each service [27]. Using UDS, diagnostic tools can control actuator of an electronic system through the *IO control* service (id 0x2F) or read **ESV** by using the *Read Data By Identifier* service (id 0x22) [26]. The standard defines the formats of the request and response messages of these two services.

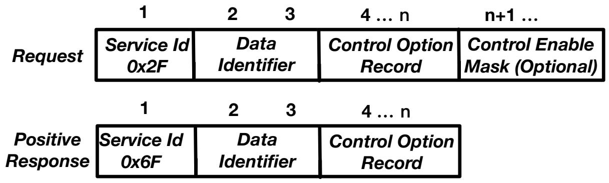


Figure 4: Request and positive response messages of UDS *IO control* service.

To control the actuator of an electronic system, the diagnostic tool first sends a request message to the ECU. After receiving the request message, the ECU will rely with a response message. Fig. 4 shows the formats of the request and positive response messages. The request message contains four parts: 1 byte service id (i.e., 0x2F), 2 bytes data identifier (DID), control option record (1 or more bytes), and Control Enable Mask Record (optional). The response message starts with 0x6F (positive response) or 0x7F (negative response). The data identifier (DID) specifies the actuator to be controlled. The control option record includes all information required by the actuator. We use ECU Control Record (**ECR**) to represent the control option record.

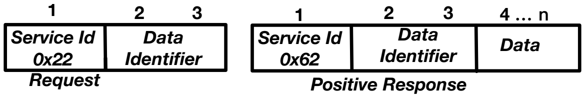


Figure 5: Request and the positive response messages of UDS *Read Data By Identifier* service.

For the *Read Data By Identifier* service, the formats of request and response messages are shown in Fig. 5. The request message contains one or more DIDs. The length of each DID is two-byte. The response message contains one or more **ESV**s. When parsing the response message, the diagnostic device employs proprietary formulas to transform the **ESV** in the message to actual **ESV**.

*Example* When testing the fog lights on the left hand, to light up the lights for 5 seconds, the diagnostic tool sends the request message "`2F 09 50 03 05 01 00 00`". The DID is

0x0950 and the **ECR** is “03 05 01 00 00”. To read the speed of the vehicle (DID 0xF40D), the diagnostic tool sends the request message ‘22 F4 0D” to the engine. Then, it receives the response message “62 F4 0D 21”. The **ESV** in the response message is 0x21 (i.e., 33). According to the formula $X * 1.0$, the actual speed is 33 km/h (i.e., $33 * 1.0 = 33.0$).

**Target of reverse engineering UDS.** We aim at three kinds of information that is not defined in the standard but used for controlling the vehicle components or reading **ESV**, including (1) the value of DID and the corresponding semantic information that specifies the component to be controlled or the type of **ESV** to be read. (2) the semantic information of **ECR**. (3) the formulas used when transforming **ESV** in response message to actual **ESV**.

## 2.4 Vehicle diagnostic tools

According to the platforms running the diagnostic software, the vehicle diagnostic tools available on the market can be roughly divided into three categories (More details in §9.1).

(1) **Professional handheld diagnostic equipment.** It contains both the hardware and software used to perform the diagnostic services. Users can connect the equipment to a vehicle and read the results shown on the equipment’s screen [36, 80].

(2) **Professional diagnostic software.** It usually runs on a laptop. After connecting the laptop to a vehicle, users can read the result shown on the software’s UI [72].

(3) **OBD based telematics app.** They usually run on a mobile phone that connects to a diagnostic device plugged into the OBD port via wireless channel (e.g., bluetooth, WIFI).

We use *professional diagnostic tools* to represent (1) professional handheld diagnostic equipment and (2) professional diagnostic software in the following part of this paper.

# 3 System Design

Fig. 6 (a) shows the architecture of DP-Reverser. The diagnostic tool is connected to the vehicle. The data collection module described in §3.1 collects the diagnostic messages and captures the video of the UI of the diagnostic tool, which will be used as the input of reverse engineering.

As a cyber-physical system [57], the data collection module includes three layers. At the sensor/actuator layer, it has one robotic clicker (i.e., actuator) to click the diagnostic tool’s screen. It also has two cameras: *camera a* captures the screenshots of the diagnostic tool to guide the robotic clicker and *camera b* records the video of the UI of the diagnostic tool for reverse engineering. At the communication layer, *camera a* sends the screenshots to a laptop via a USB connection. The laptop also uses a USB connection to send the latest control scripts to the actuator. The application layer is a laptop that analyzes the screenshots taken by *camera a* and generates the control scripts to control the robotic clicker. When the data collection module is working, we sniff the CAN frames exchanged between the diagnostic tool and the vehicle.

The diagnostic frames analysis module described in §3.2 receives the CAN frames generated by the diagnostic tool. Since one CAN frame may contain partial payload of the diagnostic protocols, this module first assembles the payload and then extracts the fields from request and response messages. The screenshot analysis module described in §3.3 takes in the video of UI and then extracts the texts shown on UI.

Finally, the request message analysis module described in §3.4 associates the local identifier fields and DID fields customized by vehicle manufacturers with the text shown on UI to determine their functionality (i.e., semantic information). The response message analysis module described in §3.5 uses the **ESV** stored in response messages and the actual values shown on UI as input. It recovers the formulas used to parse response messages.

## 3.1 Data Collection

Since diagnostic tools are usually well protected and we cannot install any other programs into them to control or extract their firmware, we leverage a robotic clicker to interact with them so that the CAN messages and UI video can be recorded automatically. Note that our approach is independent of diagnostic tools and can be applied to various diagnostic tools in order to automatically reverse engineer the diagnostic protocols of a large number of vehicles.

Fig. 6 (b) shows the procedure of the data collection module. By using *camera a* to capture the screenshot of the UI, the *UI analyzer* identifies the ECUs or **ESV**s that will be *tested*. Here, *test* means the diagnostic tool reads **ESV** or controls the component over a period of time. The (X,Y) coordinates of these ECUs/**ESV**s are sent to the planner, which will estimate the total moving distance of different sequences of clicking. The *planner* will select the sequence with the shortest path and send it to the *script generator* that transforms the clicking sequence to a script and inserts waiting times between clicking. The *script executor and logger* run the script and record the detail of each clicking. When the diagnostic tool shows a new GUI, *camera a* will take a new picture and send it to the UI analyzer to continue the data collection.

The robotic clicker is a hardware that controls a stylus pen to click the screen according to the specified (X, Y) coordinates. Since the stylus pen can only move straight along the coordinate axis with fixed speed, we determine the optimal clicking sequence with the least moving distance to reduce the time of data collection.

**UI Analyzer.** It inspects the texts shown on UI and outputs the (X, Y) coordinates to click. To achieve this goal, we leverage computer vision techniques (CV) [47] to first locate the areas with text from the figure and then filter out irrelevant areas according to the text information in them.

More precisely, given a picture of the UI, identifying the

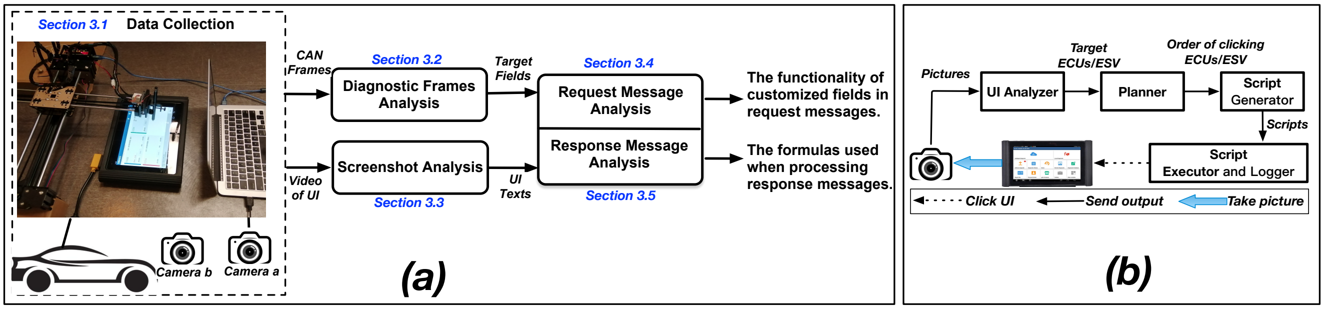


Figure 6: System Overview (a) and Data Collection Steps (b)

areas with text is a “Text Detection” task in CV. We use the open-source text detector “EAST” [39] to accomplish it because EAST can process 13 pictures per second and achieves high text detection accuracy. After identifying the areas with text, the UI analyzer checks the text content of each area to filter out the areas that are not our target (e.g., “clear trouble codes”). This is an “OCR” (i.e., “Optical Character Recognition”) task. We use *Tesseract*, one of the most popular open-source OCR engines [56, 67, 75, 77] to accomplish it, because Tesseract has a new neural net (LSTM) based OCR engine that can recognize more than 100 languages and its word error rate is only 6.4% for English and 6.29% for Chinese [76]. If the text recognized by OCR contains the keywords (e.g., “Read Data Stream”), the robotic clicker will click this area.

If some buttons do not contain text, our system recognizes them by exploiting the similarity of UI widgets. In detail, for each screenshot of the UI, we first identify all widgets in it. Then, we identify the widgets with/without text from the screenshot seperately since we cannot perform static or dynamic analysis on the diagnostic tools to extract the widgets from UI. The widgets with text are identified by using Tesseract [24]. The widgets without text are identified by using Canny edge detection [43, 69]. To identify the buttons, we calculate the similarity between each extracted widget and the pictures of pre-defined buttons by referring [86]. If this similarity is higher than a pre-defined threshold, we guide the robotic clicker to click it.

**Planner.** After receiving a set of ECUs/**ESV**s to click, the planner tries to determine the optimal clicking sequence with the least moving distance. We formalize it as a travelling salesman problem [38, 70]. That is, given a set of **ESV**s on UI and the distance between each pair of **ESV**s, the planner looks for the shortest route that visits each **ESV** exactly once and returns to the origin **ESV**. Since it is an NP-hard problem in combinatorial optimization, we approach it by using heuristic approach [73]. In particular, after getting the picture of the UI, we first calculate the distance between a pair of **ESV**s and then run the nearest neighbor algorithm [55] to determine the sequence of clicking. The nearest neighbor algorithm is a heuristic algorithm that can obtain good solutions without guaranteeing that the optimal solution will be found. However, it is much faster than brute-force approach [59]. We compared the time cost of using the nearest neighbour algorithm to select 14 **ESV**s on UI with that of randomly selecting 14 **ESV**s on UI. The result shows that, compared with random selection, the nearest neighbor algorithm saves 7.3% time of moving (i.e., (80.45-74.6)/80.45).

**Script Generator.** After getting the sequence of clicking ECUs/**ESV**s, we generate a script to control the robotic clicker. Each target will be mapped to a statement for clicking their (X, Y) coordinates. After each clicking statement, the generator inserts a statement that waits for a fixed period of time to ensure that the diagnostic tool has enough time to react to the clicking. If the diagnostic tool starts reading **ESV** or controlling the component, the waiting time will be relatively long to get enough data for reverse engineering (e.g., 30 seconds). All the clicking statements and waiting statements are combined together to generate the script.

**Script Executor and Logger.** When the robotic clicker is executing the script, it logs the timestamp of each UI clicking so that we can split the captured CAN frames and recorded video into multiple parts. When the diagnostic tools are communicating with the vehicle, we sniff the CAN frames exchanged between the diagnostic tool and the vehicle. To record the texts shown on UI, *camera b* is controlled by the “Camera Timestamp Free” app [25] running on a smartphone to record video of the UI. The recorded video contains timestamps used to associate the UI texts with the corresponding CAN frames because both the CAN frames and the video of UI are needed for reverse engineering the diagnostic messages.

## 3.2 Diagnostic Frames Analysis

After getting CAN frames through sniffing, this module assembles the raw payload of diagnostic messages and extracts the target field of request and response messages. We first filter out useless frames because they do not contain the payload of diagnostic messages in *Step 1: Screening Frames*. Then, we assemble the raw payload of diagnostic messages because the transport/network layer protocols transmit long diagnostic messages through multiple frames in *Step 2: Assembling Payload*. Finally, we extract the fields from the assembled payload

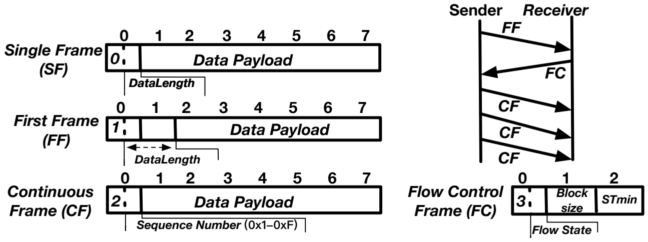


Figure 7: ISO 15765-2: Structures of single frame (SF), first frame (FF), continuous frame (CF), flow control frame (FC), and flow control mechanism.

because those fields are defined by the vehicle manufacturers in *Step 3: Fields Extraction*.

**Step 1: Screening Frames.** As shown in Fig. 7, for *ISO 15765-2* [20], four kinds of frames will be captured: single frame(SF), first frame(FF), continuous frame(CF), flow control frame(FC) [20]. We remove the flow control frames (the first four bits opcode is '`0x3`") because they are used to notify the sender the receiver's properties (e.g., buffer size). They do not carry the payload of diagnostic messages. The other three kinds of frames are kept. For *VW TP 2.0* [29], we remove three kinds of frames (i.e., broadcast, channel setup, and channel parameters) because they do not contain payload [29]. Only the data transmission frames will be kept because they contain the payload of the diagnostic messages.

**Step 2: Assembling Payload.** Since the network/transport layer protocols split the long diagnostic message into multiple short frames, we combine these short frames to assemble the raw payload. For *ISO 15765-2*, different types of frames are processed separately. If the frame is a single frame, the payload stored in it is complete and we extract the payload directly. If the frame is the first frame, the payload stored in it is incomplete. Thus, we need to first capture the following continuous frames and then assemble the complete payload by combining the payload of the first frame and the payloads of continuous frames until the total length reaches the data length field of the first frame. For *VW TP 2.0*, the data transmission frames do not contain the data length fields. We check their opcodes to determine if the current frame is the last frame or not. If true (i.e., the current packet is the last packet), we extract the concatenated payload as the raw payload. Otherwise, we concatenate its payload with the payload of the next frame.

Wen transmitting long diagnostic messages, we observe that some vehicles like BMW and Mini Copper do not directly adopt the *ISO 15765-2* protocol. Instead, the first byte of each CAN frame stores the ID of the target ECU. The remaining bytes are the payload of the diagnostic message. To correctly recover the payload of diagnostic messages, we ignore the first byte and put the remaining bytes together since the ID is not part of the payload.

**Step 3: Fields Extraction.** After obtaining the payload of a diagnostic message, we split the payload into multiple fields according to the protocol formats described in §2.3. We extract the local id, DID, **ESV** and **ECR** contained in diagnostic messages because they will be used in discovering their semantic meaning and formulas. For each **ECR**, we also extract the IO control parameter and control state from it to discover the semantic meaning of the control state field.

To extract fields of the diagnostic messages involved in reading **ESV**, since the response message of UDS can contain multiple **ESV**s and the length of each **ESV** is not fixed, we cannot extract these **ESV**s by only checking the response message. To solve this issue, we extract the last request message sent to the ECU. By using the DIDs contained in the request message as a reference, we can identify the **ESV**s in the corresponding response message, because the list of DIDs in the request message also appear in the corresponding response message with the same order and the field value after each DID is just the corresponding **ESV**.

## 3.3 Screenshot Analysis

Since the texts from the video of the UI contains the semantic information of the request messages and actual **ESV**s calculated by using formulas, we extract them and the timestamps from the video by using the OCR engine *Tesseract* [24] which can automatically convert typed or handwritten content into machine readable, editable format [23]. We also design a post-processing step to remove incorrect ECU signal values because the precision of the OCR engine may not be 100.0% and some incorrect values will be identified.

**UI Text Extraction.** We first use MPlayer [21] to transform the video into a series of images since the OCR engine can only process one single image at one time. Then, we apply *Tesseract* [24] to these images for extracting the text.

**Incorrect ESV Value Filtering.** Since sometimes the OCR engine might miss some decimal points (e.g., "25.00" is incorrectly identified as "2500"), we propose a two-stage filtration to remove incorrect **ESV**s. In the first stage, we set a normal value range for each type of **ESV**. For each **ESV** extracted by OCR-engine, we check if it is in the pre-defined range or not [15]. If true, we include it in the data set. Otherwise, we exclude it. In the second stage, we employ the anomaly detection (i.e., outlier detection) algorithm [51] to remove the values that are significantly different from the majority of the identified **ESV**s. The reason is that during a short period of time, the measured **ESV**s cannot change greatly. In other words, the "outlier" has a high possibility to be incorrect ones generated by the OCR engine.

## 3.4 Request Message Analysis

If the values of fields in request messages and their semantic information (i.e., functionality) are not defined in the standards, we recover their semantic information by using the texts shown on UI. When reading the **ESV** through UDS or KWP 2000, the request message includes the DID (as shown in Fig. 5) or local identifier (as shown in Fig. 3), which are customized by the manufacturers. When controlling vehicle components with UDS, the request message also includes DID of the target component(as shown in Fig. 4). To obtain the semantic meaning of the DID or local identifier in the request message, we extract the types of **ESV**s displayed on UI or the type of controlled component shown on UI.

## 3.5 Response Message Analysis

When parsing the response message of UDS and KWP 2000, if it contains **ESV**, the diagnostic tool uses proprietary formulas to transform the **ESV** in response to **ESV** displayed on UI. To infer these formulas, since both the **ESV** in response messages and the **ESV** displayed on UI keep changing, we first need to correlate them. Then, we design an improved genetic programming algorithm to infer the formulas. We do not employ the linear regression to infer the formula [71] because the linear regression can only infer linear formulas (e.g., $Y = a*X+b$) [62] and it cannot infer the nonlinear formulas (e.g., $Y = X_0 * X_0$) [32].

**Step 1: Constructing the mapping between ESV in diagnostic messages and ESV displayed on UI.** We create a data set that contains a series of data pairs (i.e., $(X,Y)$) according to the timestamps in the video and the captured CAN frames. Each data pair contains one **ESV** extracted from diagnostic messages (i.e., $X$) and the **ESV** extracted from UI (i.e., $Y$).

For each **ESV** extracted from the diagnostic message (i.e., $X$), we get the timestamp of receiving the message (i.e., $time_{traffic}$). Then, we $time_{traffic}$ to search the nearest timestamp displayed on UI ($time_{ui}$). By using $time_{ui}$, we can identify the corresponding actual **ESV** displayed on UI (i.e., $Y$). We combine $X$ and $Y$ to get a value pair $(X,Y)$ and add it to the data set. Note that, each **ESV** $X$ is an integer value for UDS and each **ESV** contains two integer values for KWP 2000 (i.e., $X_0$ and $X_1$ mentioned in §2.3.1).

The alignment of diagnostic messages and screenshots of UI is important for correctly inferring the formulas. Before starting the data collection, we adopt two methods to ensure the alignment of diagnostic messages and screenshots of UI. One is to use NTP protocol [64]. The other is to use the well-documented OBD-II protocol [34]. We described the details of the alignment in §9.4.

**Step 2: Inferring the formula through genetic programming.** Based on the data set, we search the space of mathematical expressions to find the model that best fits the data set. That is, based on a series of independent variables (i.e., **ESV** in diagnostic messages, $X$) and their dependent variable targets (i.e., **ESV** displayed on UI, $Y$), we want to find out the formula (i.e., $f$) that allows $f(X) = Y$. In the machine learning field, this task is called "Symbolic Regression" [16]. We apply genetic programming (**GP**) to solve this problem [30] because GP can handle both arithmetic operators (i.e., addition, subtraction, division and multiplication) and nonlinear functions (e.g., square root, log, sine, tangent).

GP uses syntax trees to represent the formulas (e.g., $a*a+b$), where the functions are interior nodes (e.g., $*$, $+$) and the variables/constants are the leaves of the trees (e.g., $a$, $b$). GP will change the randomly generated formulas in multiple generations and select the one that has the highest fitness score (i.e., most close to $f(X) = Y$). Starting from a few randomly generated formulas, GP selects the fittest individuals and leverage evolution (i.e., Crossover and Mutation) to get the next generation of formulas. The evolution process repeats until one stopping criteria is satisfied. Two kinds of stopping criteria are usually used: ($i$) the number of evolution has reached the maximum number of generations. ($ii$) the fitness of one of the latest formula has reached the threshold. The fitness (similar to "error" or "loss" in machine learning) of one formula is calculated using functions such as "mean absolute error" or "mean squared error".

Table 2: Pre-processing of the data set and post-processing of the variables contained in the formulas inferred by GP.

| | **ESV** in response messages (X) | | **ESV** displayed on UI (Y) | |
|---|---|---|---|---|
| Range | Pre-process | Formula<br>Post-processing | Pre-process | Formula<br>Post-processing |
| $> 10^4$ | $X' = X/10^4$ | Replace($X'$, $X/10^4$) | $Y' = Y/10^4$ | Replace($Y'$, $Y/10^4$) |
| $10^3$-$10^4$ | $X' = X/10^3$ | Replace($X'$, $X/10^3$) | $Y' = Y/10^3$ | Replace($Y'$, $Y/10^3$) |
| $10^2$-$10^3$ | $X' = X/100$ | Replace($X'$, $X/100$) | $Y' = Y/100$ | Replace($Y'$, $Y/100$) |
| 10-$10^2$ | $X' = X/10$ | Replace($X'$, $X/10$) | $Y' = Y/10$ | Replace($Y'$, $Y/10$) |
| 0.1-1.0 | - | - | $Y' = Y*10$ | Replace($Y'$, $Y*10$) |
| $10^{-2}$-$10^{-1}$ | - | - | $Y' = Y*100$ | Replace($Y'$, $Y*100$) |
| $10^{-3}$-$10^{-2}$ | - | - | $Y' = Y*10^3$ | Replace($Y'$, $Y*10^3$) |
| $< 10^{-3}$ | - | - | $Y' = Y*10^4$ | Replace($Y'$, $Y*10^4$) |

**Step 3: Pre-processing of the data set and post-processing of the formula.** When using GP to infer formulas, we find that the correctness of the output is affected by the ranges of **ESV** displayed on UI (i.e., variable targets, $Y$ in Tab. 2). For example, if most values of $Y$ are extremely small (e.g.,$< 0.0001$), GP will directly set a constant value as the formula (e.g., $Y = 0.0001$). In this case, the fitness score (e.g., "mean absolute error") is smaller than the pre-defined threshold and the evolution process stops. Moreover, if most $Y$ values are extremely large (e.g., $> 1000$), GP will generate complex formulas to decrease the distance between $f(X)$ and $Y$.

Our manual testing shows that when most absolute values of $X$ and $Y$ are in the range 1.0 to 10.0, the output of GP achieves the highest accuracy. Thus, before using the GP to process $(X,Y)$ pairs in the data set, we first check the absolute values of $Y$ to determine if they should be reduced or enlarged: If more than half of the absolute values of $Y$ are larger than 10 (smaller than 1), we think these $Y$ values should be reduced (enlarged). In this case, we replace each $Y$ value in the data set with a new value $Y'$, which is equal to $Y*a$, where $a$ is the reduced or enlarged factor. We combine the new $(X, Y')$ pairs to construct a data set and infer the formula $f(X) = Y'$. After getting the formula $f(X) = Y'$, we replace the variable "$Y'$" with "$Y*a$" to get the formula "$f(X) = Y*a$", which describes the relation between $X$ and $Y$. For example, as shown in Tab. 2, if most absolute values of $Y$ are in range $10^3$ to $10^4$, we divide each value with $10^3$ (i.e., $Y'$). If the inferred formula is $Y' = X$, we replace variable "$Y'$" with "$Y/10^3$" (i.e., the final formula is $Y/10^3 = X$).

For the $X$ values in the data set, we process them with the same method to ensure that most new $X'$ values are in the range 1.0 to 10.0. Because all of $X$ values are integer (i.e., larger than 1.0), we only check if they should be reduced or not ("**ESV** in response messages (X)" in Tab. 2).

## 4 Experiment

We conduct a large-scale experiment to evaluate DP-Reverser and answer the following research questions:
**Q1:** Can the OCR engine correctly extract text from the UI of diagnostic tools? (§4.1)
**Q2:** For the frames of reading **ESV**s through OBD-II protocol, can DP-Reverser correctly recover their formulas? (§4.2)
**Q3:** For the frames of reading **ESV**s through UDS and KWP 2000 protocols, can DP-Reverser correctly recover the formulas and semantic information? (§4.3)
**Q4:** How is the performance of alternative algorithms to infer the formulas of UDS and KWP 2000? (§4.4)
**Q5:** For the CAN frames of controlling the vehicle components, can DP-Reverser correctly recover the semantic information of request messages? (§4.5)
**Q6:** How many telematics apps contain the formulas of diagnostic protocols? (§4.6)

Table 3: Vehicles and diagnostic tools used in experiments

| Car | Vehicle Model | Protocol | Diagnostic Tools |
|---|---|---|---|
| **Car A** | Skoda Octavia | UDS | LAUNCH X431 |
| **Car B** | Volkswagen Magotan | KWP 2000 | VCDS |
| **Car C** | Volkswagen Lavida | KWP 2000 | LAUNCH X431 |
| **Car D** | Lexus NX300 | UDS | Techstream |
| **Car E** | Mini Copper R56 | UDS | AUTEL 919 |
| **Car F** | Mini Copper R59 | UDS | AUTEL 919 |
| **Car G** | BMW i3 | UDS | AUTEL 919 |
| **Car H** | RongWei MARVEL X | UDS | AUTEL 919 |
| **Car I** | Changan Eado | UDS | AUTEL 919 |
| **Car J** | BMW 532Li | UDS | AUTEL 919 |
| **Car K** | Volkswagen Passat | KWP 2000 | AUTEL 919 |
| **Car L** | Toyota Corolla | UDS | AUTEL 919 |
| **Car M** | Peugeot 308 | UDS | AUTEL 919 |
| **Car N** | Kia k2 (UC) | UDS | AUTEL 919 |
| **Car O** | Ford Kuga | UDS | AUTEL 919 |
| **Car P** | Honda Accord | UDS | AUTEL 919 |
| **Car Q** | Nissan Teana | UDS | AUTEL 919 |
| **Car R** | Audi A4L | UDS | AUTEL 919 |

To answer these questions, we use 18 vehicles listed in Tab. 3 to conduct experiments. We use four diagnostic tools to interact with these vehicles: (1) Two professional handheld diagnostic equipment (i.e., LAUNCH X431 [80] and AUTEL 919 [36]). (2) Two professional diagnostic software (i.e., VCDS [72] and Techstream [37]).

### 4.1 Precision of the OCR engine

**Approach:** We use a camera to record the video of the screens of two diagnostic tools (i.e., AUTEL 919 and LAUNCH X431) and then check if the OCR engine can correctly extract the text from the pictures transformed from the video.
**Result:** The result is shown in Tab. 4. We find that the precision of the OCR engine reaches 97.6% for AUTEL 919 and 85.0% for LAUNCH X431. The reason is that the screen of AUTEL 919 is larger with better resolution than that of LAUNCH X431.

Table 4: Performance of OCR engine

| Diagnostic Tool | #Total Pics | #Correct Pics | Precision |
|---|---|---|---|
| AUTEL 919 | 500 | 488 | 97.6% |
| LAUNCH X431 | 500 | 425 | 85.0% |

**Answer of Q1:** The precision of the OCR engine reaches 97.6% for AUTEL 919 and 85.0% for LAUNCH X431.

### 4.2 Result of OBD-II Frames

**Approach:** To measure the effectiveness of DP-Reverser, we test the **ESV** in the OBD-II protocol because the standard has defined the formula of each **ESV**, which serves as the ground truth. We use one vehicle simulator, which supports OBD-II protocol, to generate the diagnostic messages, and employ an OBD telematics app "ChevroSys Scan Free" to read **ESV**s from the vehicle simulator. After recording the video of the UI and capturing the diagnostic messages sent or received by the app, DP-Reverser reverse engineers the formulas. Finally, we verify the correctness of the obtained formulas with the OBD-II standard (i.e., ground truth).
**Result:** For the 7 types of **ESV**s of OBD-II protocol listed in Tab. 5, DP-Reverser correctly recover all formulas.

For the first three types of **ESV**s, the recovered formulas are the same as that of ground truth. For the fourth **ESV** Engine Speed (RPM), the formula of ground truth contains two variables (i.e., $X_0$ and $X_1$). Although the output of our system only contains one variable (i.e., $X_0$), our result is correct because the real values of $X_1$ are all 128 in real diagnostic messages. Thus, the formula of ground truth can be simplified to $Y = 64X_0 + 32$.

For last three types of **ESV**s, the ground truth includes two formulas for each **ESV** because the **ESV** has two possible units. However, DP-Reverser identifies one formula for each **ESV**, because the app only uses one formula to parse the response messages. Moreover, for Engine Coolant Temperature, the formula of ground truth $Y = 1.8X - 40$ and the output of our system $Y = 1.7X - 22$ look different. However, since the values of $X$ in traffic are in the range 0xA0-0xC0 (i.e.,

Table 5: Result of reverse engineering the formulas of OBD-II protocol: the request messages, the formulas in ground truth, and the formulas inferred by DP-Reverser.

| **ESV** | Request<br>Message | Formula<br>Ground Truth | Formula (GP)<br>System Output |
|---|---|---|---|
| Absolute Throttle<br>Position | `01 11` | $Y = \frac{X}{2.55}$ | $\frac{Y}{10} = \frac{(X/100)}{0.255}$ |
| Calculated<br>Engine Load | `01 04` | $Y = \frac{X}{2.55}$ | $\frac{Y}{10} = \frac{X/100}{0.255}$ |
| Fuel Tank<br>Level Input | `01 2F` | $Y = 0.392 * X$ | $\frac{Y}{100} = 0.389 * \frac{X}{100}$ |
| Engine Speed<br>(RPM) | `01 0C` | $Y = \frac{256*X_0+X_1}{4}$ | $Y = 64X_0 + 32$ |
| Vehicle Speed<br>(Km/h or Mile/h) | `01 0D` | $Y = X$ or<br>$Y = 0.621 * X$ | $\frac{Y}{100} = 0.619 * \frac{X}{100}$ |
| Engine Coolant<br>Temperature($^oC$ or $^oF$) | `01 05` | $Y = X - 40$ or<br>$Y = 1.8 * X - 40$ | $Y = 1.7 * X - 22$ |
| Intake Manifold Absolute<br>Pressure(KPa or inHg) | `01 0B` | $Y = X$ or<br>$Y = X/3.39$ | $\frac{Y}{10} = \frac{X}{100}/0.335$ |

160-192), the output of the formula in the ground truth (i.e., $Y = 1.8X - 40$) is in the range 248-305. The output of the recovered formula (i.e., $Y = 1.7X - 22$) is in the range 250-304.4. Thus, the output of these two formulas are almost the same, and we regard the output of GP as a correct one.

**Answer of Q2:** The result shows that, for the CAN frames of reading 7 types of **ESV**s with OBD-II protocol, the precision of reverse engineering is 100%.

## 4.3 Result of UDS and KWP 2000 Frames

**Approach:** For the **ESV**s of UDS, we manually analyze the frames and the corresponding actual **ESV**s shown on UI to extract the data identifiers (i.e., DIDs) and infer the formulas. These DIDs and formulas are used as the ground truth. For the **ESV**s of KWP 2000, we use a document containing the formulas KWP 2000 (provided by an experienced vehicle researcher) as the ground truth. Since some **ESV**s do not have formulas (e.g., the door status is open or closed), we only extract the DID or local identifier used in the request message (i.e., no formula can be extracted). To verify the correctness of inferred formulas, we conduct an additional experiment by using the **ESV**s displayed on the dashboard of real vehicles as the ground truth.

**Result:** In total, we select 446 **ESV**s from these 18 vehicles. For these **ESV**s, 290 of them contain formulas used when parsing the response messages. Other 156 **ESV**s do not have formulas (e.g., the door is open or closed). Tab. 6 lists the precision of reverse engineering formulas with GP .

Overall, the precision of GP reaches 98.3%. Note that if the coefficient in one inferred formula is very close to that of the ground truth, we regard the inferred formula as a correct one. The slight difference between the coefficient of inferred formulas and the coefficient of ground truth is generated due to two reasons: (*i*) There is a time interval between the time receiving the response message and the time displaying the **ESV** on the device screen. For some $(X,Y)$ pairs in the dataset, $X$ comes from the latest received message and $Y$ has not been updated. (*ii*) The stopping criterion of GP defines that if the fitness of one latest formula has reached the threshold, the GP will stop evolution.

Table 6: Result of **ESV** analysis: Number of **ESV**s with formulas (i.e., column "**#ESV** (formula)"), number of **ESV**s that GP can infer formulas correctly (i.e., column "#Correct **ESV**"), precision of inferring formulas with GP (i.e., column "Precision"), and the number of **ESV**s without formulas (i.e., column "**#ESV** (Enum)").

| Car | **#ESV**<br>(formula) | #Correct **ESV** | Precision | **#ESV**<br>(Enum) |
|---|---|---|---|---|
| **Car A** | 28 | 28 | 100.0% | 0 |
| **Car B** | 8 | 7 | 87.5% | 0 |
| **Car C** | 5 | 5 | 100.0% | 0 |
| **Car D** | 12 | 12 | 100.0% | 5 |
| **Car E** | 5 | 5 | 100.0% | 4 |
| **Car F** | 8 | 8 | 100.0% | 5 |
| **Car G** | 5 | 4 | 80.0% | 22 |
| **Car H** | 5 | 5 | 100.0% | 13 |
| **Car I** | 11 | 9 | 81.8% | 0 |
| **Car J** | 20 | 20 | 100.0% | 20 |
| **Car K** | 41 | 41 | 100.0% | 0 |
| **Car L** | 29 | 28 | 96.6 % | 20 |
| **Car M** | 4 | 4 | 100.0% | 14 |
| **Car N** | 26 | 26 | 100.0% | 19 |
| **Car O** | 18 | 18 | 100.0% | 9 |
| **Car P** | 7 | 7 | 100.0% | 6 |
| **Car Q** | 18 | 18 | 100.0% | 17 |
| **Car R** | 40 | 40 | 100.0% | 2 |
| **Total** | 290 | 285 | 98.3% | 156 |

Although some inferred KWP 2000 formulas are quite different from that of ground truth, manual verification reveals that they are equal when implemented in systems. We use two **ESV**s to explain the reason.

▷ *Vehicle Speed*: The formula of ground truth contains two variables (i.e., $X_0$, $X_1$). But the formula inferred by GP only contains one (i.e., $X_1$). The reason is that, in frames, the values of $X_0$ are all `0x64` (i.e., 100). The formula of ground truth will be transformed to $Y = X_1$ when $X_0$ is constant value 100.

▷ *Torque Assistance*: In frames, $X_1$ has two possible values (i.e., `0x7F` and `0x81`). The formula in ground truth will change to $Y = X_0 * (-0.001)$ and $Y = X_0 * 0.001$, respectively. Moreover, the formula generated by GP will be $Y = X_0 * (-1.03/1000)$ and $Y = X_0 * (0.97/1000)$. Thus, the formulas in ground truth and GP are still the same.

*Cause of inconsistency.* We find that the inferred formula of lateral acceleration contains only one variable (i.e., $X_1$) whereas the formula of ground truth contains two variables (i.e., $X_0$ and $X_1$). This inconsistency is caused by the data in the collected frames. More precisely, in these frames, all the values of variable $X_0$ is `0x00`. Thus, the genetic programming regards the values of $X_0$ as constant and the inferred formula only uses another variable $X_1$.

*Result validation with real vehicles.* Since the real vehicles display some **ESV**s on their dashboards, we can use them as the ground truth of reverse engineering. In detail, we use the diagnostic tools to read **ESV**s from the vehicles and then combine the diagnostic messages and the inferred formulas to obtain the possible **ESV**s shown on dashboards. We compare the possible **ESV**s with the **ESV**s displayed on the dashboards (recorded by camera) to check the correctness of the inferred formulas. We use four real vehicles in this validation experiment. The result in Tab. 7 shows that the inferred formulas are correct for all these vehicles.

Table 7: Result of using real vehicles to evaluate the correctness of reverse engineering the formulas of **ESV**s.

| Vehicle | **ESV**<br>on vehicle dashboards | Formula (GP)<br>System Output | Same<br>or not |
|---|---|---|---|
| **Car F** | Engine speed | $Y = X$ | √ |
| **Car K** | Engine speed | $Y = X_0 * X_1/5$ | √ |
| **Car L** | Coolant Temperature | $Y = 0.5X$ | √ |
| **Car R** | Engine speed | $Y = 64.1X_0 + 0.241X_1$ | √ |

*Time cost.* We measure the average time needed by genetic programming to infer the formulas. The result listed in Tab. 8 shows that it costs about 201.40 and 192.19 seconds to infer UDS and KWP 2000 formulas, respectively. One reason may be that we set the maximum number of generations to 30 and each generation contains 1000 formulas to calculate their fitness score ('mean absolute error'). To shorten the time, we will decrease the maximum number of generations and the number of formulas in each generation in future work.

Table 8: Average time cost of inferring formulas (seconds).

| Protocol | Genetic<br>Programming | Linear<br>Regression | Polynomial<br>Curve Fitting |
|---|---|---|---|
| UDS | 201.40 | 0.0009 | 0.0004 |
| KWP 2000 | 192.19 | 0.0017 | 0.0006 |

**Answer of Q3:** The result shows that, for the frames of reading 290 types of **ESV**s with UDS or KWP 2000, the precision of reverse engineering is 98.3%.

## 4.4 Comparion with Alternative Algorithms for Formula Inferring

**Approach:** Previous research LibreCAN [71] enhances READ [60] to identify the fields contained in the CAN messages transmitted between ECUs. It shows that linear regression can be used to find the relation between these fields and OBD sensors. Since READ [60] does not consider the transmission layer protocol, it cannot correctly identify the payload of a long diagnostic message before analyzing it. We examine this issue and evaluate the precision of using linear regression and polynomial curve fitting [63] to infer formulas for comparison.

**Result:** We describe the result from two aspects: (1) Necessity of payload recovering; (2) Precision of linear regression and polynomial curve fitting.

*(1) Necessity of payload recovering.* A long diagnostic message will be transmitted through multiple CAN messages. Recovering the raw payload of a diagnostic message is the first step before analyzing it. To measure the percentage of CAN messages that should leverage the transport/network layer protocol to assemble the payload, we examine the number (percentage) of each type of messages contained in UDS and KWP 2000 traffic.

For UDS, we analyze the CAN messages of **Car A** (Skoda Octavia). The captured traffic contains 31,963 messages. Half of them (i.e., 17,601/31,963=55.1%) are single frames. The multi-frame messages (i.e., first frames and continuous frames) account for 32.0% (i.e., 10,213/31,963=32.0%). Other frames are flow control frames.

For KWP 2000, we analyze the CAN messages of **Car B** (Volkswagen Magotan) and **C** (Volkswagen Lavida) since they implemented KWP 2000. The collected traffic contains 4,556 messages in total. We find that 3,425 (i.e., 75.2%) of them needs to wait for the next frames. The remaining 1,131 messages (i.e., 24.8%) are the last frames of multi-frame messages (i.e., do not need to wait for next frames).

The result shows that, for UDS, 32.0% CAN messages must be processed by payload recovering. For KWP 2000, 75.2% CAN messages must be processed by payload recovering. Otherwise, we cannot extract fields from them.

Table 9: Number/Percentage of single frames and multi-frames in UDS and KWP 2000 traffic.

| Protocol | # Single<br>Frames | # Multi<br>Frames | # Total<br>Frames |
|---|---|---|---|
| UDS | 17,601 (55.1%) | 10,213 (32.0%) | 31,963 |
| KWP 2000 | 1,131 (24.8%) | 3,425 (75.2%) | 4,556 |

*(2) Precision of linear regression and polynomial curve fitting.* The performance of linear regression and polynomial curve fitting is shown in Tab. 10. Overall, the precision of linear regression reaches 43.8% and that of polynomial curve fitting achieves 32.1%. In contrast, the GP algorithm employed by our system has a much better precision (i.e., 98.3%).

*Cause of inconsistency.* Manual investigation shows that GP outperforms linear regression and polynomial curve fitting due to the following reasons:

(*i*) Some outliers are generated if the OCR engine fails to correctly extract sensor values from the screenshots. For example, in one screenshot, the actual **ESV** is "3.7", but the

Table 10: Precision of inferring formulas of UDS and KWP 2000 with linear regression (i.e., column "#Correct **ESV** (Linear Reg)") and polynomial curve fitting (i.e., column "#Correct **ESV** (Polynomial)").

| Car | **#ESV** (formula) | #Correct **ESV** (Linear Reg) | # Correct **ESV** (Polynomial) |
|---|---|---|---|
| **Car A** | 28 | 14 | 20 |
| **Car B** | 8 | 2 | 1 |
| **Car C** | 5 | 1 | 2 |
| **Car D** | 12 | 10 | 8 |
| **Car E** | 5 | 3 | 2 |
| **Car F** | 8 | 4 | 3 |
| **Car G** | 5 | 2 | 2 |
| **Car H** | 5 | 5 | 3 |
| **Car I** | 11 | 9 | 6 |
| **Car J** | 20 | 11 | 8 |
| **Car K** | 41 | 2 | 0 |
| **Car L** | 29 | 25 | 12 |
| **Car M** | 4 | 4 | 2 |
| **Car N** | 26 | 14 | 11 |
| **Car O** | 18 | 11 | 6 |
| **Car P** | 7 | 3 | 3 |
| **Car Q** | 18 | 7 | 4 |
| **Car R** | 40 | 34 | 28 |
| **Total** | 290 | 127 | 93 |

OCR engine outputs "8.0". Another screenshot contains actual **ESV** "11.4", but the OCR engine only extracts "4" from it. Due to the incorrect sensor values, the factor related to the variable of the inferred formula is affected, which is much smaller than that of ground truth. We observe that GP is more robust than linear regression. Other studies also show that GP is robust to outliers/noise [40,44,52].

(*ii*) Linear regression can only infer linear formulas and polynomial curve fitting can only find polynomial formulas. For KWP 2000 protocol, if the formula has two variables (i.e., $X_0$ and $X_1$), the linear formulas can be represented as $Y = \beta_0 X_0 + \beta_1 X_1 + X2$. However, if the formulas of ground truth contain non-linear elements, linear regression cannot identify them. An example is "Engine Speed (RPM)" (i.e., $Y = X_0 * X_1/5$). Linear regression can only obtain the formula $Y = 0.45X_0 + 17.85X_1 + 498.47$. Polynomial curve fitting can find one formula $Y = 0.032X_0 * X_1 - 10X_0 + 64.52X_1 + 0.04X_0^2 - 0.46X_1^2 - 215$. This formula contains $X_0 * X_1$, but its coefficient is much smaller than that in the ground truth (i.e., 0.2). GP can handle these non-linear formulas since it uses syntax trees to represent the formulas. Both linear and non-linear formulas can be inferred.

*Time cost.* Although the linear regression and polynomial curve fitting cannot achieve high precision when inferring formulas, their time cost is much smaller than that of GP (i.e., Tab. 8, less than one second for each **ESV**), because they do not need to iterate a large number of times.

**Answer of Q4:** The result shows that, 32.0% CAN frames for UDS and 75.2% CAN frames for KWP 2000 must be processed by payload recovering. Otherwise, the fields in them cannot be extracted. Moreover, when inferring formulas, the linear regression algorithm only achieves 43.8% precision and the polynomial curve fitting only achieves 32.1% precision. Their precision is much lower than that of DP-Reverser.

## 4.5 Result of Reverse Engineering ECR

**Approach:** We test controlling components of vehicles by using the IO control service or input output control via localidentifier service. We analyze the captured frames to infer the procedure of sending different types of request messages.
**Result:** In total, we extracted 124 **ECR**s from 10 vehicles. Five of them use IO control service of UDS (service ID is "2F") and other five of them use input output control via local identifier service of KWP 2000 (service ID is "30"). We list the number of identified **ECR**s of each vehicle in Tab. 11.

Table 11: Number of **ECR**s extracted from vehicles.

| Car | **#ECR** | Service ID | Car | **#ECR** | Service ID |
|---|---|---|---|---|---|
| **Car A** | 11 | 2F | **Car D** | 5 | 30 |
| **Car E** | 3 | 30 | **Car F** | 5 | 30 |
| **Car H** | 6 | 2F | **Car I** | 10 | 2F |
| **Car J** | 27 | 30 | **Car N** | 21 | 2F |
| **Car O** | 4 | 2F | **Car Q** | 32 | 30 |

Although these components have different DIDs or local IDs, their control procedures have similar patterns. To control one component (e.g., window or light), the controller should send three request messages. If the component sends one positive response message for each request message, the component can be controlled successfully.
*First Request:* The controller sends the "Freeze current state" message. The format is "`2F {DID: 2 bytes} 02`". The last byte 0x02 is the IO control parameter, which means freeze current state (i.e., prepare to control). For example, the first step of controlling the fog light is sending the request message "`2F 09 50 02`". When controlling other components of the vehicle, we just need to change the values of DIDs.
*Second Request:* The controller sends the "Short term adjustment" message. The format is "`2F {DID: 2 bytes} 03 {control state: n bytes}`". The byte 0x03 is the first byte of **ECR** (i.e., IO control parameter), which means short term adjustment (i.e., start controlling). Followed by the control state. For example, to light up the fog lights on the left hand for 5 seconds, the request message is "`2F {DID: 09 50} 03 {control state: 05 01 00 00}`". To control the fog lights on the right hand for 3 second, the request message is "`2F {DID: 09 50} 03 {control state: 03 00 00 00}`". Only 2 bytes in the control state are modified (one byte for the time duration of control and one byte for left/right side).
*Third Request:* The controller sends the "Return control to ECU" message. The format is "`2F {DID: 2 bytes} 00`". The

last byte 0x00 is the IO control parameter, which means return control to ECU (i.e., the control is finished). For example, after controlling the fog light, sending the request message "`2F 09 50 00`" returns the control to the ECU.

**Answer to Q5:** For the 124 **ECR**s of ten vehicles, we correctly extracted all **ECR**s. We also find that three messages are required to control each vehicle component by using the IO control service.

## 4.6 Formulas Extracted from Apps

**Approach:** When processing the response messages, if one response message contains **ESV**, the telematics apps may also use a formula to transform it to actual **ESV**. To extract these formulas, we analyze 160 telematics apps: 38 apps are downloaded from Google Play by searching the keywords "vehicle diagnostic" and "vehicle OBD". We also include all 122 apps from the data set of CANhunter [83].

For each app, we identify the data buffer that stores the response message by calling framework APIs (e.g., *InputStream.read()*). Then, we perform forward taint analysis on the data buffer to identify the statements that process the response message. If the statement includes mathematical operators (e.g., $+$, $*$), we extract the formula from the statement. We also extract the condition of using the formula by utilizing the control dependency relation to identify the dependent branch statement and then analyzing the constant values contained in the branch statement. (More details in §9.2).

**Result:** Tab. 12 lists the number of formulas discovered from apps. We only find three apps containing the formulas used to process the response messages of UDS and KWP 2000. In "Carly for VAG", we obtain 90 UDS formulas and 137 KWP 2000 formulas. For example, if the DID is 0xF43C, the corresponding formula is $Y = 0.1X - 40$. In "Carly for Mercedes", we extract 1624 UDS formulas and 468 KWP 2000 formulas. In "Carly for Toyota", we get 7 KWP 2000 formulas. For example, if the request message is "`21 1A`", the formula $Y = X$ is used to obtain the vehicle model speed.

Our tool discovers 12 apps containing formulas of OBD-II protocol. Note that the standard of OBD-II [34] has defined these formulas (e.g., engine coolant temperature $Y = X - 40$). Our system can also reverse engineer them. Manual inspection of the result reveals that the formulas in 13 apps cannot be extracted due to multiple reasons. For example, the request message is sent by subclass and the response message is parsed by the parent class, or the app only checks partial bytes of response messages to determine the used formula.

We do not discover any KWP 2000 or UDS formulas from other apps. One reason is that although some apps send request messages of KWP 2000 or UDS, they only use them to read/clear DTC or read freeze frame (i.e., they do not use KWP 2000 or UDS to read **ESV**). Moreover, since most vehicles support OBD-II protocol, some apps only send request messages of OBD-II protocol to read **ESV**.

Table 12: Telematics apps containing formulas.

| APP Name | Formula Type | # Formula |
|---|---|---|
| Carly for VAG | UDS<br>KWP 2000 | 90<br>137 |
| Carly for Mercedes | UDS<br>KWP 2000 | 1624<br>468 |
| Carly for Toyota | KWP 2000 | 7 |
| inCarDoc | OBD-II | 82 |
| Car Computer - Olivia Drive | OBD-II | 74 |
| CarSys Scan | OBD-II | 64 |
| Easy OBD | OBD-II | 55 |
| inCarDoc Pro | OBD-II | 49 |
| OBD Boy(OBD2-ELM327) | OBD-II | 45 |
| FordSys Scan Free | OBD-II | 42 |
| ChevroSys Scan Free | OBD-II | 40 |
| ToyoSys Scan Free | OBD-II | 40 |
| Obd Mary | OBD-II | 34 |
| OBD2 Boost | OBD-II | 34 |
| Obd Harry Scan | OBD-II | 28 |
| Obd Arny | OBD-II | 27 |
| MOSX | OBD-II | 24 |
| Dr Prius Dr Hybrid | OBD-II | 22 |
| Dacar Pro OBD2 | OBD-II | 21 |
| OBD2 Scanner Fault Codes Desc | OBD-II | 16 |
| Dacar Pro OBD2 | OBD-II | 14 |
| Engie Easy Car Repair | OBD-II | 8 |
| PHEV Watchdog | OBD-II | 8 |
| Torque Lite(OBD2&Car) | OBD-II | 5 |
| Kiwi OBD | OBD-II | 3 |
| OBDclick | OBD-II | 2 |
| Dr Prius Dr Hybrid | OBD-II | 1 |
| Fuel Economy for Torque Pro | OBD-II | 1 |

We also investigate the number of ECUs and **ESV**s that can be obtained from professional diagnostic tools and telematics apps by using two vehicles (i.e., VW Passat, Toyota Corolla). For VW Passat, the diagnostic tool (i.e., AUTEL 919) discovers 18 ECUs, but only 10 of them can be discovered by the app (i.e., "Carly for VAG"). Other ECUs (e.g., Airbag) are missed by the app. For Toyota Corolla, the diagnostic tool (i.e., AUTEL 919) discovers 31 ECUs, but only 14 of them can be discovered by the app(i.e., "Carly for Toyota"). For VW Passat, the diagnostic tool can read 203 **ESV**s (i.e., 173 KWP 2000 **ESV**s and 30 UDS **ESV**s). For Toyota Corolla, the diagnostic tool can read 242 **ESV**s through UDS. However, none of these **ESV**s can be read by the apps. For example, the tool sends UDS request message "`03 22 10 17`" to Toyota Corolla to get quantity of fuel injection cylinder 1. But this request message cannot be discovered in any apps.

**Answer of Q6:** For the 160 telematics apps, we only find three of them contain UDS or KWP 2000 formulas. Some other apps only contain formulas of the OBD-II protocol. A more detailed comparison with two vehicles shows that, compared with telematics apps, the professional diagnostic tools can discover more ECUs and read more **ESV**s.

## 5 Related Work

To determine the registers of supervisory control and data acquisition systems used to store constant, counter, or sensor values, Erez et al. designed a decision tree [46]. However, since it is based on fixed window size, it cannot identify the fields whose length is not equal to the window. To identify three kinds of fields contained in CAN messages, Markovitz et al. dynamically changed the window size and checked the number of values contained in the window [61].

To analyze the format of CAN messages, after putting messages of the same CAN ID in the same group, Marchetti et al. proposed READ [60], which first calculates the bit flip rate, bit flip magnitude and then uses them to determine physical (i.e., signal), counter, and CRC fields. Although READ achieves high precision when processing an online dataset, it can neither identify constant values and multiple value related fields nor discover the type of signals stored in each field (e.g., engine RPM). Pes et al. proposed LibreCAN [71] to reverse engineer the format of CAN messages. For the CAN messages collected from the vehicle, LibreCAN first enhances READ [60] to identify fields storing ECU signal values. For each field, LibreCAN uses cross-correlation to identify its related powertrain-related signals. To identify CAN messages related to body related events, LibreCAN filters out useless messages with three stages. However, LibreCAN can neither handle long messages transmitted via transport/network layer protocols nor discover the nonlinear relationships between fields and signals. CANHunter [83] performs dynamic force execution to extract all messages sent by diagnostic apps. However, it neither examines the diagnostic protocols nor analyzes the processing of response messages.

Dynamic analysis has been used to recover the format of network messages. Polyglot [42] records all instructions of processing received messages and performs taint analysis to infer the format. Polyglot integrates algorithms to identify fields. To obtain outgoing message formats, Dispatcher [41] performs backward taint analysis to build up the dependency chain. To extract the structure of the given buffer, Dispatcher checks the sources of current and previous buffer locations are same or not. If false, there exists a boundary. To replay the application communication, Replayer [68] builds a symbolic formula of how the original host processed the communication. Then, it uses an off-the-shelf decision procedure to derive an input tailored to a different host from the symbolic formula. To construct protocol field tree, AutoFormat [58] infers the message format based on the observation that bytes processed under the same execution context belong to the same message field. To recover the format of encrypted data (e.g., TLS encryption records), ReFormat [82] identifies the instructions of decryption algorithms and then uses AutoFormat [58] to recover the message format. These approaches could reverse engineer the diagnostic protocols if we can control the professional diagnostic equipments with root privilege.

## 6 Discussion

**Limitations**. DP-Reverser suffers from the following limitations: (1) DP-Reverser requires access to both the OBD port to collect the diagnostic messages and the diagnostic tools to collect the sensor values. Otherwise, it does not work. (2) Currently, DP-Reverser supports 14 kinds of functions (e.g., addition, subtraction, multiplication, division, square root, log, absolute value, negative, maximum) in the genetic programming library [30], without considering the non-trivial formulas (e.g., LFSRs, seed-key encryptions). (3) The performance of DP-Reverser relies on the performance of the OCR engine used to extract the sensor values from the GUI. (4) The payloads of diagnostic messages should be recovered based on the corresponding standards (e.g., ISO 15765-2 [11]), which is the prerequisite domain knowledge to apply DP-Reverser. (5) When analyzing the formulas contained in telematics apps, the forward taint analysis cannot handle complex apps.

**Future works**. We will apply DP-Reverser to non-trivial formulas to solve other practical problems in reverse engineering. Moreover, we will enhance the performance of OCR engines to make DP-Reverser more robust. We will improve our tool to analyze complex telematics apps.

## 7 Conclusion

We propose DP-Reverser, the first automated framework to reverse engineer the vehicle diagnostic protocols from professional diagnostic tools. Besides the semantic information of request messages in the diagnostic protocols, DP-Reverser recovers the proprietary formulas used to process the response messages. We implement DP-Reverser and conduct experiments with 18 vehicles. The results show that it can accurately reverse engineer the detailed information in the request/response messages of professional diagnostic tools.

## 8 Acknowledgement

We sincerely thank Prof. Van-Thuan Pham for shepherding our paper and the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong ITF Project (No. ITS/197/17FP), Hong Kong RGC Project (No. PolyU15223918), NSFC (No. 62002306), HKPolyU Start-up Fund (ZVU7), CCF-Tencent Open Research Fund (ZDCK), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform ZJUNGICS2021017, K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), and National Science Foundation under Grant No. 1951729, 1953813, 1953893, and 1700544. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

## References

[1] ISO 14230 Keyword Protocol 2000 Part 3: Implementation. http://read.pudn.com/downloads118/ebook/500929/14230-3.pdf, 1996.

[2] Keyword Protocol 2000: Data Link Layer Recommended Practice. http://www.internetsomething.com/kwp/kwp2000_recommended_guidlines.pdf, 1997.

[3] ISO 14230-1. https://www.sis.se/api/document/preview/612052/, 1999.

[4] ISO 14230-2. https://www.sis.se/api/document/preview/612053/, 1999.

[5] ISO 14230-3. https://www.sis.se/api/document/preview/895162/, 1999.

[6] Diagnostics on Controller Area Networks (CAN) - Part 3: Implementation of unified diagnostic services (UDS on CAN). http://read.pudn.com/downloads506/doc/2103567/ISO_15765-3.pdf, 2004.

[7] SAE J1979. https://law.resource.org/pub/us/cfr/ibr/005/sae.j1979.2002.pdf, 2006.

[8] Unified diagnostic services (UDS) — Part 3: Unified diagnostic services on CAN implementation (UDSonCAN). https://www.iso.org/standard/55284.html, 2012.

[9] Car hacking demonstration video. https://www.youtube.com/watch?v=qt1xrRL9ULc, 2014.

[10] Zubie: This Car Safety Tool 'Could Have Given Hackers Control Of Your Vehicle'. https://bit.ly/3adTPmq, 2014.

[11] Diagnostic communication over Controller Area Network (DoCAN) — Part 2: Transport protocol and network layer services. https://www.iso.org/standard/66574.html, 2016.

[12] KWP 2000 and UDS Protocols for Vehicle Diagnostics: An Analysis and Comparison. http://bit.ly/2OlVNL6, 2018.

[13] Automotive most used protocols - KWP2000 and UDS. http://www.devcoons.com/automotive-protocols-kwp2000-uds/, 2018.

[14] Experimental Security Assessment of BMW Cars: A Summary Report. https://bit.ly/2BSP848, 2018.

[15] OBD2 PIDs for Programmers (Technical). http://bit.ly/36RJUzH, 2018.

[16] Symbolic Regression and Genetic Programming. https://jankrepl.github.io/symbolic-regression/, 2018.

[17] What Is Kwp2000. https://gizaedu.weebly.com/blog/what-is-kwp2000, 2018.

[18] CAN - ISO 11898 BUS-system for automotive diagnostic and flash applications. https://automotive.softing.com/en/standards/bus-systems/can-iso-11898.html, 2019.

[19] Controller Area Network (CAN) Overview. https://bit.ly/3BfL1sl, 2019.

[20] ISO 15765-2. http://canbushack.com/iso-15765-2/, 2019.

[21] MPlayer. http://www.mplayerhq.hu/, 2019.

[22] OBD2 EXPLAINED - A SIMPLE INTRO (2019). http://bit.ly/2FKQagH, 2019.

[23] OCR Engines. http://www.cvisiontech.com/library/ocr/image-ocr/ocr-engines.html, 2019.

[24] Tesseract Open Source OCR Engine. https://github.com/tesseract-ocr/tesseract, 2019.

[25] Timestamp Camera Free. http://bit.ly/2FNY1dz, 2019.

[26] UDS ISO 14229: Standardized CAN-based protocol for diagnostics. https://automotive.softing.com/en/standards/protocols/uds-iso-14229.html, 2019.

[27] UDS Protocol. https://www.piembsystech.com/protocol/diagnostic-protocol/uds-protocol/, 2019.

[28] Unified diagnostic services (UDS) - Part 2: Session layer services. https://www.iso.org/standard/77322.html, 2019.

[29] VW Transport Protocol 2.0 (TP 2.0) for CAN bus. https://jazdw.net/tp20, 2019.

[30] Welcome to gplearn's documentation! https://gplearn.readthedocs.io/en/stable/, 2019.

[31] Launch X431 V+ Full System Diagnostic Tool Bi-Directional Scan Tool. http://www.x431tool.com/wholesale/launch-x431-v-plus.html, 2020.

[32] Nonlinear Functions. https://bit.ly/3ck25QI, 2020.

[33] OBD-II (Check Engine Light) Trouble Codes. https://www.obd-codes.com/trouble_codes/, 2020.

[34] OBD ISO 15031. https://automotive.softing.com/en/standards/protocols/obd-iso-15031.html, 2020.

[35] OBD2 PIDs for Programmers (Technical). https://shorturl.at/dmtIJ, 2020.

[36] AUTEL 919. https://item.jd.com/70636576685.html, 2021.

[37] Toyota TIS Techstream. http://shorturl.at/zADOQ, 2021.

[38] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[39] argman. EAST: An Efficient and Accurate Scene Text Detector. https://github.com/argman/EAST, 2020.

[40] G. Barlow and C. Oh. Robustness analysis of genetic programming controllers for unmanned aerial vehicles. In *Proc. EuroGP*, 2006.

[41] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. CCS*,

2009.

[42] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proc. CCS*, 2007.

[43] J. Canny. A computational approach to edge detection. *TPAMI*, 1986.

[44] P. Day and A. Nandi. Robust text-independent speaker verification using genetic programming. *IEEE Tran. TASLP*, 2006.

[45] CSS Electronics. CAN BUS EXPLAINED - A SIMPLE INTRO. https://bit.ly/3mm5up2, 2019.

[46] N. Erez and A. Wool. Control variable classification, modeling and anomaly detection in modbus/tcp scada systems. *IJCIP*, 2015.

[47] D. Forsyth and J. Ponce. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.

[48] D. Frassinelli, S. Park, and S. Nürnberger. <<i know where you parked last summer>> automated reverse engineering and privacy analysis of modern cars. In *Proc. S&P*, 2020.

[49] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. SP*, 2016.

[50] F. Garcia and D. Oswald. A New Wireless Hack Can Unlock 100 Million Volkswagens. https://bit.ly/3iAUvXL, 2016.

[51] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 2004.

[52] T. Hu, W. Banzhaf, and J. Moore. Robustness and evolvability of recombination in linear genetic programming. In *Proc. EuroGP*, 2013.

[53] P. Jing, Q. Tang, Y. Du, L. Xue, X. Luo, T. Wang, S. Nie, and S. Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *Proc. USENIX Security*, 2021.

[54] M. Kang and J. Kang. Intrusion detection system using deep neural network for in-vehicle network security. *PloS one*, 2016.

[55] G. Kizilateş and F. Nuriyeva. On the nearest neighbor algorithms for the traveling salesman problem. In *Proc. CCSEIT*. 2013.

[56] M. Koistinen, K. Kettunen, and J. Kervinen. How to improve optical character recognition of historical finnish newspapers using open source tesseract ocr engine. *Proc. of LTC*, 2017.

[57] Kwanwoo L. Cyber-Physical Systems (CPS) vs. IoT. https://kwanwooleecom.wordpress.com/2018/03/03/cyber-physical-systems-cps-vs-iot/, 2018.

[58] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proc. NDSS*, 2008.

[59] Suzanne Ma. Understanding the Travelling Salesman Problem (TSP). https://blog.routific.com/travelling-salesman-problem, 2020.

[60] M. Marchetti and D. Stabili. Read: Reverse engineering of automotive data frames. *Trans. TIFS*, 2018.

[61] M. Markovitz and A. Wool. Field classification, modeling and anomaly detection in unknown can bus networks. *Vehicular Communications*, 2017.

[62] MathsIsFun. Linear Equations. https://www.mathsisfun.com/algebra/linear-equations.html, 2017.

[63] Mathworks. Polynomial Curve Fitting. https://www.mathworks.com/help/matlab/math/polynomial-curve-fitting.html, 2020.

[64] Mathworks. NTP: The Network Time Protocol. http://www.ntp.org/, 2021.

[65] C. Miller and C. Valasek. Adventures in automotive networks and control units. *Def Con*, 2013.

[66] C. Miller and C. Valasek. Black Hat USA 2015: The full story of how that Jeep was hacked. https://bit.ly/3oGp16j, 2015.

[67] M. Nayak and A. Nayak. Odia characters recognition by training tesseract ocr engine. *IJCA*, 2014.

[68] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proc. CCS*, 2006.

[69] T. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *Proc. ASE*, 2015.

[70] I. Oliver, D. Smith, and J. Holland. Study of permutation crossover operators on the traveling salesman problem. In *International Conference on Genetic Algorithms and their applications*, 1987.

[71] M. Pesé, T. Stacer, C. Campos, E. Newberry, D. Chen, and K. Shin. Librecan: Automated can message translator. In *Proc. CCS*, 2019.

[72] ROSS-Tech. VCDS: Diagnostic Software for VW-Audi Group Cars. https://www.ross-tech.com/vag-com/, 2020.

[73] D. Simchi-Levi and O. Berman. Heuristics and bounds for the travelling salesman location problem on the plane. *Operations research letters*, 1987.

[74] C. Smith. *The Car Hacker's Handbook: A Guide for the Penetration Tester*. No Starch Press, 2016.

[75] R. Smith. An overview of the tesseract ocr engine. In *Proc. ICDAR*, 2007.

[76] R. Smith. History of the tesseract ocr engine: what worked and what didn't. In *Document Recognition and Retrieval*, 2013.

[77] R. Smith, D. Antonova, and D. Lee. Adapting the tesseract open source ocr engine for multilingual ocr. In *MOCR*, 2009.

[78] F. Sommer, J. Dürrwang, M. Wolf, H. Juraschek, R. Ranert, and R. Kriesten. Automotive network protocol de-

tection for supporting penetration testing. In *Proc. SECURWARE*, 2019.

[79] H. Song, H. Kim, and H. Kim. Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network. In *Proc. ICOIN*, 2016.

[80] Launch Tech. X-431 Pad III. https://launchtechusa.com/new-product-x431-pad3/, 2020.

[81] Q. Wang, Z. Lu, and G. Qu. An entropy analysis based intrusion detection system for controller area network in vehicles. In *Proc. SOCC*, 2018.

[82] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proc. ESORICS*, 2009.

[83] H. Wen, Q. Zhao, Q. Chen, and Z. Lin. Automated cross-platform reverse engineering of can bus commands from mobile apps. In *Proc. NDSS*, 2020.

[84] B. Williams. *Intelligent Transport Systems Standards*. Artech House Publishers, 2008.

[85] L. Yu, J. Chen, H. Zhou, X. Luo, and K. Liu. Localizing function errors in mobile apps with user reviews. In *Proc. DSN*, 2018.

[86] S. Yu, C. Fang, Y. Yun, and Y. Feng. Layout and image recognition driving cross-platform automated mobile testing. In *Proc. ICSE*, 2021.

[87] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *Proc. ICSE*, 2021.

# 9 Appendix

## 9.1 Three categories of diagnostic tools

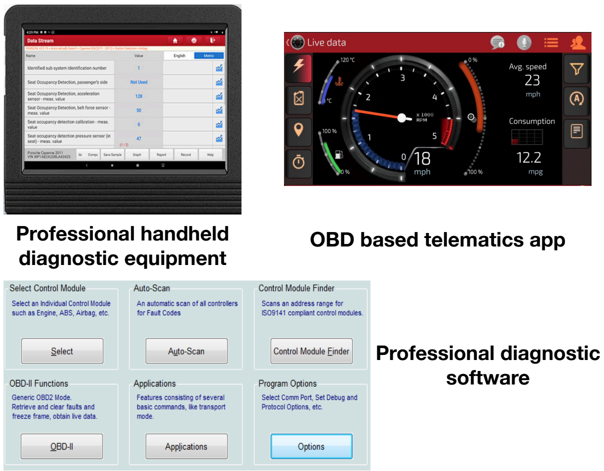

Professional handheld diagnostic equipment
OBD based telematics app
Live data
Avg. speed
23
mph
Consumption
12.2
mpg
18
mph
Professional diagnostic software
Select Control Module
Select an Individual Control Module such as Engine, ABS, Airbag, etc.
Select
Auto-Scan
An automatic scan of all controllers for Fault Codes
Auto-Scan
Control Module Finder
Scans an address range for ISO9141 compliant control modules.
Control Module Finder
OBD-II Functions
Generic OBD2 Mode. Retrieve and clear faults and freeze frame, obtain live data.
OBD-II
Applications
Features consisting of several basic commands, like transport mode.
Applications
Program Options
Select Comm Port, Set Debug and Protocol Options, etc.
Options

Figure 8: Three categories of diagnostic tools.

Fig. 8 shows three categories of diagnostic tools. (1) The *professional handheld diagnostic equipment* can cover 90% of vehicle models on the market. For each vehicle model, the equipment can: a) Scan all ECUs to perform diagnoses; b) Control ECUs to perform active test (e.g., turn fuel pump on/off); c) Perform ECU coding (i.e., reprogram adaptive data). (2) The *professional diagnostic software* can read data flow from ECUs, scan trouble codes, or use OBD-II protocol to read emission related data. (3) The *OBD based telematics apps* can read emission related data. If the apps or the third-party libraries in these apps have the vulnerabilities [87] or bugs [85] that can let attackers gain the root permission, attackers can leverage these vulnerabilities to control the communications between the apps and ECUs.

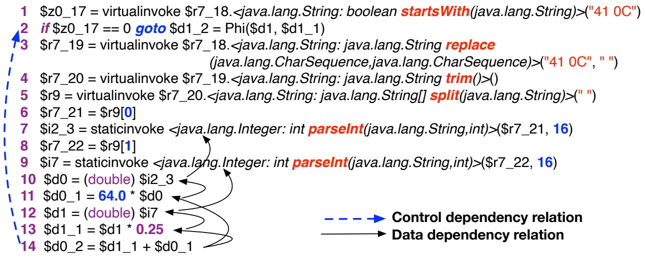
## 9.2 Response Analysis of Telematics Apps

When processing the response messages, if one response message contains **ESV**, a formula will be used to transform it to actual **ESV**. To extract these formulas, as shown in Alg. 1, we first perform forward taint analysis to identify all the statements that use the content of response messages. Then, we extract the processing statements containing mathematical operators (e.g., $+,-,*,/$). We also identify the condition under which the formula will be used.

**Algorithm 1:** Formula extraction of app

```
Input: Stmts: Set of statements contained in the app, APIs: Set of
       framework APIs that read the response messages.
Output: Formulas: List of formulas extracted from the app,
        Conditions: List of conditions of each formula.
1  Function FormulaExtraction(Stmts,APIs):
2      Formulas = [];
3      Conditions = [];
4      foreach stmt ∈ Stmts do
5          if stmt invokes api && api ∈ APIs then
6              ProcStmts = forwardTaintAnalysis(stmt);
7              foreach procstmt ∈ ProcStmts do
8                  if procstmt includes math operations then
9                      DDStmts=getDataDepStmts(procstmt);
10                     formula=extractFormula(DDStmts);
11                     Formulas.add(formula);
12                     CDStmts=getControlDepStmts(procstmt);
13                     condition=generateCondition(CDStmts);
14                     Conditions.add(condition);
15                 end
16             end
17         end
18     end
19     return Formulas, Conditions;
```

For each Android app, to identify the statements processing the response message, we add a taint tag to the buffer storing response message and perform forward taint analysis. In detail, for the statement that can receive response messages (e.g., `InputStream.read(byte[])`), we identify the variable of the data buffer and add a taint tag to it because it stores the content of the response message (Alg.1 line 4-5). Then, we perform forward taint analysis from these tainted variables because the it can identify all statements that use the tainted



```
1   $z0_17 = virtualinvoke $r7_18.<java.lang.String: boolean startsWith(java.lang.String)>("41 0C")
2   if $z0_17 == 0 goto $d1_2 = Phi($d1, $d1_1)
3   $r7_19 = virtualinvoke $r7_18.<java.lang.String: java.lang.String replace
                                    (java.lang.CharSequence,java.lang.CharSequence)>("41 0C", " ")
4   $r7_20 = virtualinvoke $r7_19.<java.lang.String: java.lang.String trim()>()
5   $r9 = virtualinvoke $r7_20.<java.lang.String: java.lang.String[] split(java.lang.String)>(" ")
6   $r7_21 = $r9[0]
7   $i2_3 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String,int)>($r7_21, 16)
8   $r7_22 = $r9[1]
9   $i7 = staticinvoke <java.lang.Integer: int parseInt(java.lang.String,int)>($r7_22, 16)
10  $d0 = (double) $i2_3
11  $d0_1 = 64.0 * $d0
12  $d1 = (double) $i7
13  $d1_1 = $d1 * 0.25
14  $d0_2 = $d1_1 + $d0_1
```

Figure 9: Example of processing response message.

variables (Alg.1 line 6). For example, in Fig. 9, since the variable $7_18 stores the hex string of a response message, we extract line 1-14 by performing static taint analysis.

For each statement identified through forward taint analysis, we check if it contains mathematical operators (e.g., $+,-,*,/$) or not (Alg.1 line 7-8). If true, we leverage the data dependency relations to extract the statements calculating the final result (Alg.1 line 9). We extract the formula from these statements and add it to the output (Alg.1 line 10-11). For example, in Fig. 9, line 11,13,14 are identified since they contain mathematical operators. We focus on line 14 since it is executed after lines 11 and 13. By analyzing the data dependency relations of line 14, we discover that lines 13,12,9 and lines 11,10,7 are used to calculate the final result. Thus, the corresponding formula is "$v1 * 0.25 + 64 * v2$". The $v1$ and $v2$ are the int values extracted from the response message. Note that the data dependency relation analysis stops at lines 7 and 9 since they extract int values from the response message.

To discover the condition under which the formula will be used, for each processing statement with mathematical operators, we leverage the control dependency relation to identify the related branch statement (`if`, `switch`) (Alg.1 line 12). Control dependency means if the condition of the branch statement is (not) satisfied, the processing statement will be executed [49]. For each variable used in the branch statement, we check the statement that defines it to discover which field value of the response message is checked (Alg.1 line 13). We record the condition in the output. For example, in Fig. 9, the formula in line 14 is calculated when the condition in line 2 is checked. We can infer that the formula is calculated when the prefix of the response message is "`41 0C`" since the variable $z0_17 in the branch statement is generated by calling `String.startsWith("41 0C")`.

## 9.3 Attack Real Vehicles With the Reverse Engineering Result

**Approach** To show the usage of the reverse engineering result of diagnostic tools, we conduct experiment with four real vehicles (i.e., BMW i3, Lexus NX300, Toyota Corolla, and Kia). Based on the reverse engineering result, we send the diagnostic messages to read data, control ECUs, or reset ECUs of the vehicles. Then, we check if the attack success or not.

**Result:** We test the diagnostic messages shown in Tab. 13. All of them succeed when the vehicles are running. For example, by sending the diagnostic message `40 05 30 11 00 ... 00` to Toyota Corolla, we can successfully unlock all doors when the vehicle is running.

Table 13: Using reverse engineered diagnostic messages to attack BMW i3, Lexus NX300, Toyota Corolla, and Kia. Note that part of each message is hidden to avoid being abused.

| Diagnostic Message(BMW) | Functions |
|---|---|
| `29 03 22 DB ... E5` | Read brake pressure |
| `12 03 22 DE ... 9C` | Read accelerator position |
| `43 05 31 01 ... 03` | Control high beam (FLEL) |
| `43 05 31 01 ... 01` | Control low beam (FLEL) |
| `60 05 31 01 ... 13` | Control turn light (KOMBI) |
| `01 02 ... 01` | Reset collision safety module |
| `60 02 ... 01` | Reset combination instrument |
| Diagnostic Message(Lexus) | Functions |
| `03 22 ... 7B` | Read engine speed (Engine) |
| `03 22 ... 59` | Read throttle position (Engine) |
| `04 30 01 ... 10` | Control displayed speed (KOMBI) |
| `04 30 02 ... 08` | Control engine speed (KOMBI) |
| Diagnostic Message(Toyota) | Functions |
| `40 05 30 11 00 ... 00` | Unlock all doors |
| `40 05 30 1C 00 ... 00` | Turn on the wiper |
| `40 05 30 11 00 ... 00` | Unlock the trunk |
| Diagnostic Message(Kia) | Functions |
| `04 2F B0 ... 03` | Unlock central lock |
| `04 2F B0 ... 03` | Turn on all light on dashboard |

## 9.4 Alignment of Diagnostic Messages and Screenshots of GUI

We adopt two methods to ensure the alignment of diagnostic messages and screenshots of GUI.

(1) For the app recording the video of GUI and the software capturing the diagnostic messages, we modify their setting so that they will use timestamps with the precision of millisecond. Moreover, before recording the video and diagnostic messages, we put the smartphone and Windows PC in the same local area network. Then, they use the Network Time Protocol(NTP) [64] to synchronize their system time.

(2) Before collecting the diagnostic messages of UDS or KWP 2000, we first use the diagnostic tools to read **ESV**s by using OBD protocol. We save the request and response messages of OBD-II protocol and the video of GUI of diagnostic tools to perform the alignment. Since the standard of OBD-II protocol is well-defined, we have the format of each request message and formula used when parsing each response message. Thus, we can infer the real **ESV**s of OBD-II protocol based on the captured diagnostic messages. Given one response message, we calculate the real **ESV** and then search the real **ESV** on the screenshots of GUI. If the real **ESV** is found on one screenshot and the timestamps of the diagnostic message and screenshot are very close, we align the diagnostic message with the screenshot. We will use the time offset between the diagnostic message and the screenshot in the alignment of other diagnostic messages and screenshots.