Hindawi
Security and Communication Networks
Volume 2022, Article ID 7144516, 13 pages
https://doi.org/10.1155/2022/7144516

Research Article

Security Analysis on Blockchain-Powered Mobile APPs Connected with In-Vehicle Networks by Context-Based Reverse Engineering

Xingyu Wu,[1] **Ziyan Qiao**,[2] **Xingjuan Cai**,[1] **Qian Wang**,[1] **Zhiqiang Xie**,[2] **Rui Sun**,[2] **Dong Zi**,[2] **Wenjia Niu**,[2] **and Endong Tong**[2]

[1]*School of Computer Science and Technology, Taiyuan University of Science and Technology, Taiyuan 030024, China*
[2]*Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China*

Correspondence should be addressed to Endong Tong; edtong@bjtu.edu.cn

Received 24 June 2022; Accepted 29 August 2022; Published 23 September 2022

Academic Editor: Yujue Wang

The controller area network (CAN) bus for interconnection of electronic control units (ECUs) plays a highly important role in modern intelligent vehicles. To facilitate the CAN Bus accessing to vehicle control or diagnosis, a number of mobile APPs are designed and published by automobile manufacturers to support driving and vehicle-based social network, and some are realized through the in-vehicle infotainment (IVI) middleware. Blockchain technologies are also mature for automobiles to interact service information with the whole industry. Unfortunately, there is a serious threat of command leakage from these mobile APPs, and the reverse engineering (RE) can be exploited by hackers. Previous work has researched this threat by an automatic reverse engineering tool on both automotive android and IOS APPs. However, in such common tool, APP itself-related contexts, including the feature information of CAN Bus commands, vehicle application functions, and control diagnostic protocols, are overlooked, which might be utilized to promote the reverse engineering recall. In this paper, we propose a context-based reverse engineering approach to find deep hidden commands for further revealing security threats for blockchain-powered mobile automotive APPs. For the reverse engineering, we design a context model of four-order tensor to organize multidimensional contexts and establish a continuous updating mechanism. Based on the context model, we further develop two basic analysis algorithms, max-compute (A) and clustering (A), to perform the analysis of CAN Bus commands. Extensive experiments are conducted, and we evaluate it by two metrics, recovered ratio and correctness ratio. Experimental results and the case studied on the familiar APP Carly validate the effectiveness of our approach and reveal the threat of command leakage.

1. Introduction

Nowadays, with the development of information technology (e.g., IoT and AI), we have entered an era of intelligent vehicles. More automotive manufacturers focus on introducing more sensors with edge computing of computers such as electronic control units (ECUs) and advanced intelligent control systems into vehicles [1–3]. More specifically, those vehicle control functions ranging from steering, braking, acceleration, to lighting and infotainment are controlled by a variety of ECUs, which requires the support of an efficient infrastructure to ensure the momentary connection of in-vehicle networks. Actually, controller area network (CAN) [2] is the most widely deployed network which provides a shared internal network. For its implementation, CAN Bus is developed as a multi-master broadcast communication protocol and offers advantages such as self-diagnosing, error correction [2], and autonomous driving by utilizing predefined commands for specific automobiles. The recent study suggests that the typical luxury sedan generally integrates 50–70 ECUs with thousands of commands. Therefore, for future intelligent

automobiles, the CAN Bus and its commands essentially play a particularly important role to represent the technology advances of a modern vehicle and its manufacture.

Generally, there are two types of connection to the CAN Bus: onboard diagnostics (OBD-II) [4] connection and middleware connection. The OBD connection is a direct physical access to the CAN Bus via OBD port (typically under the dash), which can be plugged with a dongle. Among different types of dongles on the market, some only provide diagnosis or monitoring functions, while others offer remote control functionality [4] in addition to diagnosis capabilities. The middleware connection is implemented to connect the CAN Bus by introducing middleware technologies which resides in the vehicle's head unit, and it is the major equipment to drive the system of in-vehicle infotainment (IVI) [5]. However, both of the connections to CAN Bus are transparent for drivers or common users and they only need to operate via APP installed in mobile phone or IVI system. Specifically, dongles can be accessed by mobile phone APP via Bluetooth, Wi-Fi, or cellular network, and head unit can be accessed by IVI APP via direct network communication. Moreover, mobile phone APPs situate in IOS or Android environment, and IVI APPs situate in middleware with software module to extend mobile applications features to vehicles, typically such as MirrorLink [5], Android Auto [6], and CarPlay [7]. For instance, the Siri of CarPlay can be adapted to make calls and interact with other applications of CarPlay.

Mobile phone APPs and IVI APPs, middleware of head unit, and CAN Bus form a wide attack surface on the automobile by spoofing or injecting CAN Bus commands. As a result, hackers can stop the engine remotely and disable the brake. Many efforts summarized the vulnerabilities of remote control for vehicles through this attack surface, including mobile APP masquerading, privilege escalation, replay attack [8], DoS or DDoS attack, and man-in-the-middle attack. Some emerging work focuses on reverse engineering (RE) [9] of the CAN Bus commands, which is regarded as a significant building block for subsequent attacks on in-vehicle systems. In addition to observing the traffic inside the automotive for obtaining the CAN packets and replaying them back into the CAN to attack the vehicle, Wen et al. [4] developed a cost-effective (no real car needed) and automatic (no human intervention required) system CAN-HUNTER for reverse engineering of CAN Bus commands, which use just car companion mobile APPs for both Android and iOS platforms. Inspired by their efforts, we re-examine their implementation as shown in Figure 1, and we take the APP Carly used in their experiment as the instance. Through the reverse engineering, we get the source code including two CAN Bus command IDs, a label of Unified Diagnostic Services (UDS) protocol, a parameter with value 1A87, as well as a text description "left-front door" with semantics to show a corresponding function. However, it is difficult to further reveal the detailed command in the real CAN Bus only based on such source code. It requires massive contexts on other aspects, characteristics of CAN Bus message, third-party or automotive APP function, control/diagnose protocol, and vehicle details, except source code itself.

In addition, the implementation of blockchain technology [10, 11] on automobile industry is promising to manage information and interaction. Many researches try to exploit blockchain technology [12] to share information such as manufacturer data, real-time data from sensors, and environment variables. It can be also correspondingly used on automotive mobile APPs to support more innovations. Therefore, it is important to do the research blockchain-powered mobile APPs.

In this paper, we propose an approach of context-based reverse engineering, aiming to establish a general framework to perform security analysis on the phone and IVI APPs toward the threat of CAN Bus command leakage.

In summary, we make the following contributions:

(i) We design a context model of four-order tensor with a continuous updating mechanism through the interaction between the source code and the RE analysis module to organize multidimensional contexts for reverse engineer

(ii) We further develop two basic analysis algorithms—max-compute (A) and clustering (A), to explore the semantics of CAN Bus commands

(iii) We evaluate our approach by two metrics, recovered ratio (Recover@1, Recover@3, Recover@5) and correctness ratio on the three different dimensions of the model and receive impressive results

The remainder of this paper is organized as follows. Section 2 describes the background. Then, we propose the context-based reverse engineering of IVI APPs in Section 3. Experiments and detailed analysis are reported in Section 4. Section 5 discusses the related work. Finally, we conclude the paper in Section 6.

2. Background

2.1. Communication Infrastructure in Modern Automobile. Figure 2 presents the communication framework toward the automobile CAN Bus, and there are three ways of connection: (1) using mobile APP for accessing OBD-II dongle to connect CAN Bus; (2) using mobile APP to directly connect to the IVI APP for further CAN Bus access; (3) using mobile APP to connect the cloud server via cellular network and then further access the IVI APPs from the cloud for indirectly connecting to the CAN Bus. We can discover that the wireless communication protocol includes cellular network such as 2G~5G, Wi-Fi, and Bluetooth, while the communication between IVI APP and CAN Bus implemented directly by middleware. The middleware implements a middle interface between mobile applications and vehicle ECUs. More specifically, the middleware enables the drivers' interaction with IVI and even displays the compatible mobile applications completely on the IVI touch screen. In general, modern middleware is implemented in head unit and supported by most automobiles. Once the request is received by the middleware, the vehicle will take

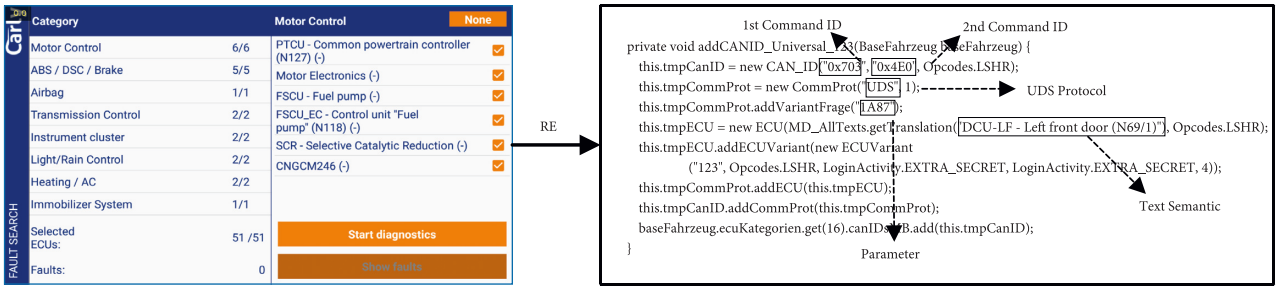

Figure 1: An example of mobile APP reverse engineering.

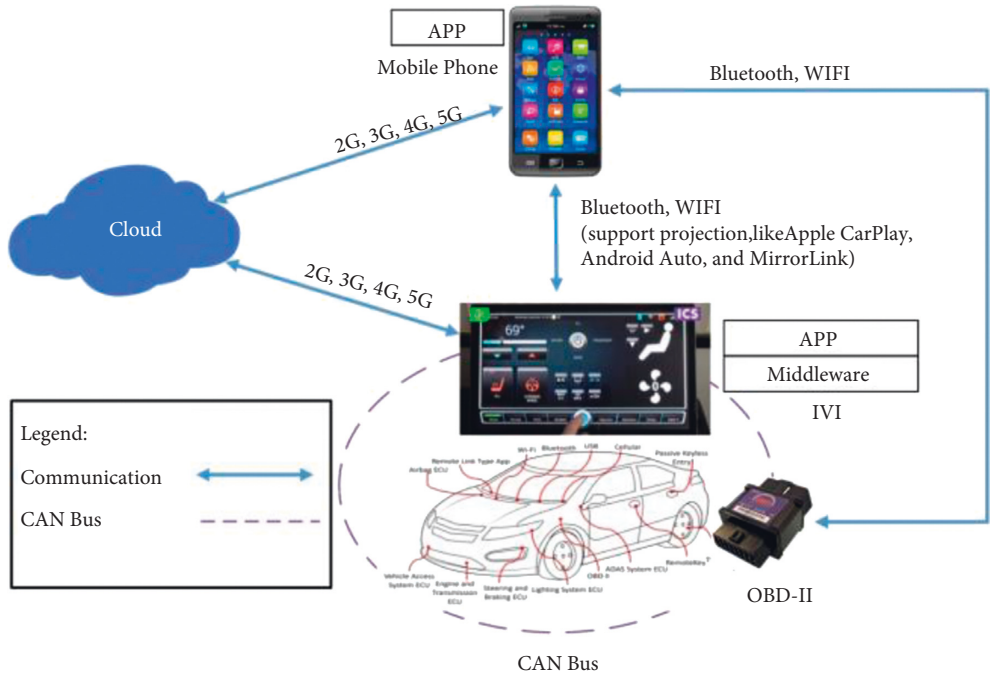

Figure 2: The automobile CAN Bus-oriented communication framework.

corresponding actions and send back a response or notification across the CAN Bus.

Note that the middleware utilizes the technologies of remote projection and procedural calls (RPC). Typically, APPs such as Apple CarPlay, Android Auto, and MirrorLink use projection to display an adapted user interface on the vehicle IVI screen.

2.2. CAN Bus Command. The CAN Bus, typically consisting of transmitting and receiving amplifier, is designed to connect different ECUs. The data which are sent back and forth for CAN communications are defined as message or frame. As shown in Figure 3, a CAN Bus message is composed with a specific data structure.

The onset of the frames is indicated by a SOF (start of frame) package (a dominant bit) with EOF (end of frame) package (7 recessive bits) at the frame end. CAN frame carries data containing at most 8 bytes, along with segments for characterizing the message identifier, the CRC (cyclic redundancy check) check, the RTR (remote transmission request), the IDE (identifier extension bit), the r0, and the DLC (data length code). The message identifier has either 11 or 28 bits and refers to the target ECU to forward. The CRC field consists of 16 bits, where one bit is delimiter and others are for checksum. ACK indicates that whether the data are received normally.

In this paper, differently, we separate the term "command" from "CAN message identifier (CAN ID)," because all ECUs will receive the command in network Bus, but only the specified ECU can execute the function as a response. Under the context of OBD and UDS protocol, some service ID (SID in UDS) or PID in OBD composing data in the frame can represent an explicit command for requesting a function of ECUs. Hence, the CAN Bus command exists within the 0–8 bytes in data segment of a CAN Bus message, and it is referred as a CAN Bus command or a command segment. The command syntactic which is hexadecimal is generally classified into two categories, (1) control command, e.g., unlocking the right-rear door or stopping the

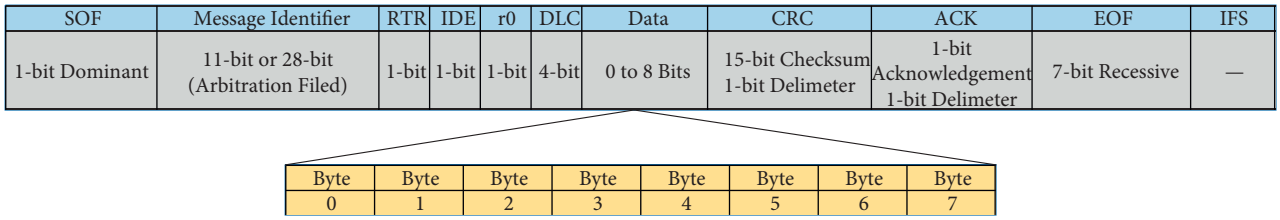

Figure 3: Structure of CAN frame for bus command understanding.

Table 1: Data captured in the CAN Bus of a real sedan.

SeqNum	System time	CAN channel	CAN ID	LEN	Command
0	21:00.9	ch1	$0\times00AF$	0×08	00 00 00 00 00 00 00 00
1	21:00.9	ch1	0×0169	0×06	7B B1 40 FE 24 00
2	21:00.9	ch1	$0\times01F2$	0×02	20 00

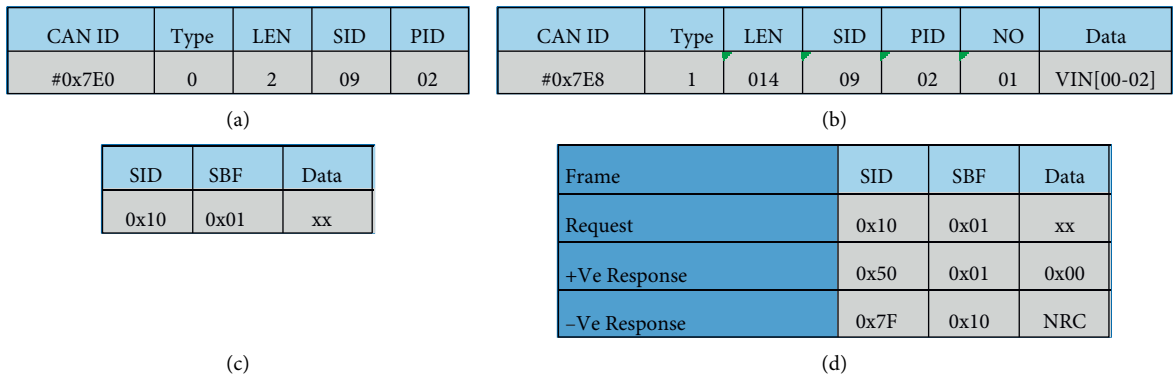

Figure 4: Request and response frames of OBD and UDS. (a) OBD-II request frame. (b) OBD-II response frame. (c) UDS request frame. (d) UDS response frame (positive and negative response).

engine and (2) diagnosis command for querying necessary status data.

Table 1 shows three records about the key commands of CAN message captured from the CAN Bus of a real sedan. The LEN field indicates the total number of bytes in the response. The three commands have 2, 6, 8 bytes, respectively, and take specific functions (e.g., open the left-front door) for specific ECUs according to corresponding CAN IDs.

2.3. OBD and UDS Frame Structure. The request and response frames of OBD and UDS are shown in Figure 4. In Figure 4(a), the LEN field specifies the following byte number in the request. SID is the service identifier, which is 0×09 in this case and represents the "Request Vehicle Data" service with a parameter ID (PID). The PID 0×02 corresponds to the vehicle identification number (VIN). The response for the request frame is shown in Figure 4(b). The SID and PID code fields should be the same as the request frame. The type is 0 if the response satisfies the request within a single frame, while the type 1 is used to indicate the "start frame" of a multi-frame packet if the response includes multiple frames. In an OBD response, the SID field equals 0×40 plus the SID from the request. NO is the number of data items (1 for the VIN in this case) for service 0×09. Data segment contains the first three bytes of the requested data.

In addition to legitimate OBD, many vehicles also support the newer Unified Diagnostic Services (UDS) standard, defined by ISO 14229–3, which builds on legislated OBD. The UDS protocol working on the CAN protocol can request the maximum 8 byte data and get the response from a message. In UDS protocol, there are also two available types of frames—diagnostic request frame and diagnostic response frame. The UDS protocol frame format is shown in Figures 4(c) and 4(d). The request frame has 3 fields including SID, sub-function ID (SFD), and data. There are two types of response, positive response and negative response. Whenever the tester requests to the server, it will send the response message by adding 0×40 to the respective SID for reference, if it is correct and the server has executed the request successfully. In positive response, the first byte should be request SID plus 0×40. If the client requests in an inappropriate frame format or the server is not able to execute the request due to the internal problem, then it will send a negative response to the client. The first byte of negative response should be $0\times7F$, and the second byte and the third byte should be SID and response code, respectively. If the ECU or server fails to send a request, it will send a response message with negative response code (NRC).

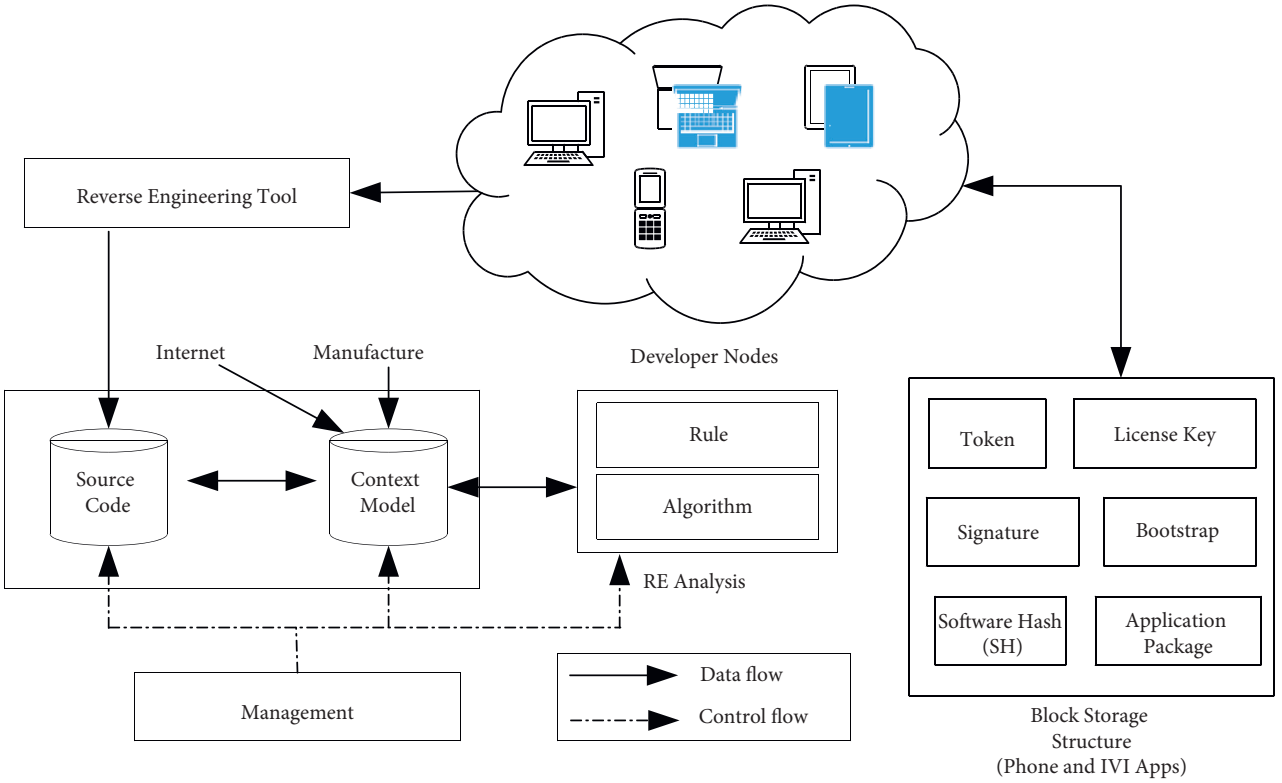

Figure 5: The workflow of context-based reverse engineering on blockchain-powered mobile APP.

3. Design

3.1. Model Architecture. The whole workflow of context-based reverse engineering on blockchain-powered vehicle APP is shown in Figure 5. This method is designed for reverse engineering on the phone and IVI APPs, suitable for IOS and Android platforms.

In our approach, the traditional APPs are combined with blockchain which has multiple software developer nodes [13]. We used the alliance blockchain, which is managed by all the alliance members, and all nodes in this alliance chain can share on-chain data. Those nodes are uploaded and downloaded with the block storage structure including some symbols and data of phone and IVI APPs. The acquirement of source data directly correlates with blockchain nodes. Firstly, we use typical RE tools including Apktool [13], dex2jar [14], and jadx [15] to obtain source codes. Next, we build a context model to organize multidimensional knowledge to support RE analysis. Note that, the context model has a continuous updating mechanism of the interaction with the source code and the RE analysis module, which contains two parts, rules and algorithms for analyzing. The management module is responsible for coordinating the source code, context model, and RE analysis and forming a complete loop to work continuously.

3.2. Constructing 4-Order Tensor-Based Context Model. Based on APP permissions, CAN IDs, semantic, and command, we design a 4-order tensor model to obtain and store semantics of command function described by text (see

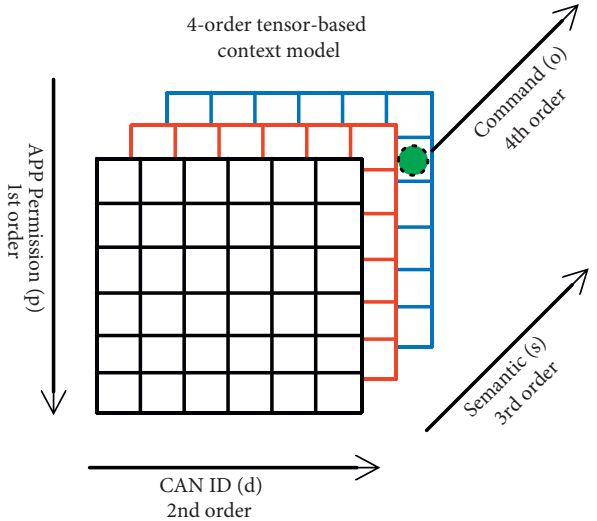

Figure 6: 4-order tensor-based context model for analyzing CAN Bus command.

Figure 6). We construct a 4-order tensor $A \in R^{X\times Y\times Z\times M}$ ($X = |p|$, $Y = |d|$, $Z = |s|$, $M = |o|$) in which $A_{x,y,z,m}$ refers to an element with the coordinate of (x, y, z, m) in the tensor A.

For the tensor A, the 1st-order p describes the APP permissions, such as access_wifi_STATE and write_external_STORAGE; The 2nd-order d contains the CAN ID; the 3rd-order s distinguishes the different semantics of commands, for example, "RWTS—rear-end door closing" refers to the semantic corresponding to the command

"0 × 4F7"; the 4th-order o stores 8-byte commands. Under the index of 4 orders, we can locate a text description for the semantic of command function. To facilitate the computation based on the tensor A, we further obtain the following four matrices in

$$A(p) =, A_{i,\mathbf{y},\mathbf{z},\mathbf{m}} = \left(A_{i,1,1,1}, \ldots, A_{i,1,1,|\mathbf{o}|}, \ldots, A_{i,1,|s||o|,}, \ldots, A_{i,|d|,|s|,|o|}\right), 1 \le \mathbf{i} \le |p|, \mathbf{i} \in N^{+}. \tag{1}$$

Here, the column vector size is $|p| \times 1$ and within the first-order p, we can choose i to determine the APP permission as an index to corresponding vector $A_{i,y,z,m}$.

We further get matrices $A(d)$, $A(s)$, and $A(o)$ with the same way in the following formulae:

$$A(\mathbf{d}) =, A_{\mathbf{x},i,\mathbf{z},\mathbf{m}} = \left(A_{1,i,1,1}, \ldots, A_{1,i,1,|\mathbf{o}|}, \ldots, A_{1,i,|\mathbf{s}|,|\mathbf{o}|}, \ldots, A_{|p|,i,|\mathbf{s}|,|\mathbf{o}|}\right), 1 \le \mathbf{i} \le |d|, \mathbf{i} \in N^{+}. \tag{2}$$

Here, the column vector size is $|d| \times 1$ and within the 2nd-order d, we can choose i to determine the CAN ID as an index to corresponding vector $A_{x,i,z,m}$.

$$A(\mathbf{s}) =, A_{\mathbf{x},\mathbf{y},i,\mathbf{m}} = \left(A_{1,1,i,1}, \ldots, A_{1,1,i,|o|}, \ldots, A_{1,|d|\,i,|o|}, \ldots, A_{|p|,|d|,i,|o|}\right), 1 \le \mathbf{i} \le, |s|\mathbf{i} \in N^{+}. \tag{3}$$

Here, the column vector size is $|s| \times 1$ and within the 3rd-order s, we can choose i to determine the semantics as an index to corresponding vector $A_{x,y,i,m}$.

$$A(\mathbf{o}) =, A_{\mathbf{x},\mathbf{y},i,\mathbf{m}} = \left(A_{1,1,1,i}, \ldots, A_{1,1,|s|,i}, \ldots, A_{1,|d|,|s|,i}, \ldots, A_{|p|,|d|,|\mathbf{s}|,i}\right), 1 \le \mathbf{i} \le |o|, \mathbf{i} \in N^{+}. \tag{4}$$

Here, the column vector size is $|o| \times 1$ and within the 4th-order o, we can choose i to determine the commands as an index to corresponding vector $A_{x,y,z,i}$.

3.3. Reverse Engineering Analysis. In this section, we propose two basic analysis algorithms; one is named max-compute (A), which makes a statistic on the number of revealed semantics in three orders. Another is named clustering (A), which performs clustering on texts corresponding to CAN Bus command, so as to analyze similar functions.

In Algorithm 1 line 2, for each element in APP permission p, it computes the number of revealed semantics *Amount*. Finally, we output the vector of *List_p*($A(p)[|p|]$, *List_d*($A(d)[|d|]$, *List_o*($A(o)[|o|]$.

For the given semantic set s, a text clustering method corresponding to CAN Bus command is realized to get the commands with the most similar function. As Algorithm 2 shows, this method begins with $|c|$ clusters. In line 5, we calculate distance between different clusters. In line 7, we merge two individual clusters with the shortest distance into a larger cluster. We then repeat the operations from line 4 to line 8 until the number of cluster reaches $|k|$.

The distance between two individual clusters is calculated by $Dal(c_i,c_j)$.

$$\%\ \mathrm{Dal}\left(c_i, c_j\right) = \frac{1}{\left|C_{\mathbf{i}}\right|\left|C_{\mathbf{j}}\right|} \sum_{s_i \in \mathbf{C_i}} \sum_{s_i \in \mathbf{C_j}} L\left(s_i, s_j\right), L\left(s_i, s_j\right) = \frac{\left|\left\{w_k | w_k \in s_i \cap w_k \in s_j\right\}\right|}{\log\left(\left|s_i\right|\right) + \log\left(\left|s_j\right|\right)}, \tag{5}$$

where c_i and c_j are clusters and $c_i \cap c_j = \varnothing$, $L(s_i, s_j)$ is the similarity between two semantics $s_i \in c_i$ and $s_j \in c_j$. w_k represents a word in a semantic, $s = w_1, w_2, \ldots, w_n$. Molecular part calculates the numbers of the same words that appear simultaneously in both semantics, and the denominator part calculates the logarithmic sum of the number of words in the semantics.

3.4. Computational Complexity Analysis. Computational complexity analysis of the two analysis algorithms is discussed as follows. We firstly analyze the computational complexity of Algorithm 1, which traverses all states in the three-dimensional space $p \times d \times s$ to calculate the number of revealed semantics in three orders. Thus, the computational complexity of Algorithm 1 is a level of $O(n)$, where n is the size of semantics.

For each cluster x, $x \in \{1, 2, \ldots, |k|\}$, as we need to perform clustering until the current number of clusters greater than the terminated cluster number $|k|$ and calculate the distance between any two commands. The computational complexity for computing the distance is $O(mn)$, in which the parameter m refers to the size of o_i and the parameter n refers to the size of o_j. Therefore, the computational complexity of Algorithm 2 is a level of $O(|k||k|mn)$.

4. Experiment

4.1. Setup. The experimental environment configuration is shown in Table 2, based on the Windows 11 of a Honor Hunter V700 laptop with Intel(R) Core(TM) i7-10750H CPU @ 2.60 GHz, 16G RAM. There are three reverse engineering tools used to analyze in our experiment. Apktool [13] is used to extract permission information from AndroidManifest.xml and original byte codes of the APPs. It is a tool for reverse engineering 3rd party, closed, and binary Android APPs which can decode resources to original form nearly and rebuild them after making some modifications. We use dex2jar [14] to work with files android.dex and java.class and convert the file classes.dex to classesdex2jar.jar which is the combination of original source class files. Jadx [15], a standalone graphical utility, is used to display Java source codes from Java object code ".class" files.

4.2. Evaluation Metric

4.2.1. Recovered Ratio. We describe the semantic integrity discovered by our method in terms of recovered ratio, which counts the number of semantics. We define the corresponding recovered ratio R_p, R_d, R_o for the three orders p, d, and o as follows, where $|\text{List\_}p|$, $|\text{List\_}d|$, and $|\text{List\_}o|$ represent the number of semantics using Algorithm 1 on the orders p, d, and o, respectively. The number of semantic recoveries with App permission as the tensor is represented by $|S_p|$, the number of semantic recoveries with CAN ID as the tensor is represented by $|S_d|$, and the number of semantic recoveries with command as the tensor is represented by $|S_o|$. R represents the semantic recovered ratio, $|S\_\text{recovered}|$ is the number of recovered semantics, and $|S\_\text{real}|$ is the real number of semantics. We further define $\mathscr{R}@3$ and $\mathscr{R}@5$ which take the best top 3 and 5 experimental results supposing that the results are arranged in order indexed by i.

$$\begin{aligned}\mathscr{R}_{\mathbf{p}} &= \frac{\text{Maxcompute}\left(A_{i,\mathbf{y},\mathbf{z},\mathbf{m}}\right).|\text{List\_}p|}{\left|S_p\right|},\\ \mathscr{R}_{\mathbf{d}} &= \frac{\text{Maxcompute}\left(A_{\mathbf{x},i,\mathbf{z},\mathbf{m}}\right).|\text{List\_}d|}{\left|S_d\right|},\\ \mathscr{R}_{\mathbf{o}} &= \frac{\text{Maxcompute}\left(A_{\mathbf{x},\mathbf{y},i,\mathbf{m}}\right).|\text{List\_}o|}{\left|S_o\right|},\end{aligned} \tag{6}$$

$$\begin{aligned}\mathscr{R}_{\mathbf{p}}@3 &= \frac{\sum_{i=1}^{3}|\text{List\_}p|}{3\left|S_p\right|},\\ \mathscr{R}_{\mathbf{d}}@3 &= \frac{\sum_{i=1}^{3}|\text{List\_}d|}{3\left|S_d\right|},\\ \mathscr{R}_{\mathbf{o}}@3 &= \frac{\sum_{i=1}^{3}|\text{List\_}o|}{3\left|S_o\right|},\end{aligned} \tag{7}$$

$$\begin{aligned}\mathscr{R}_{\mathbf{p}}@5 &= \frac{\sum_{i=1}^{5}|\text{List\_}p|}{3\left|S_p\right|},\\ \mathscr{R}_{\mathbf{d}}@5 &= \frac{\sum_{i=1}^{5}|\text{List\_}d|}{3\left|S_d\right|},\\ \mathscr{R}_{\mathbf{o}}@5 &= \frac{\sum_{i=1}^{5}|\text{List\_}o|}{3\left|S_o\right|},\end{aligned} \tag{8}$$

$$\mathscr{R} = \frac{\left|S_{\text{recovered}}\right|}{\left|S_{\text{real}}\right|}. \tag{9}$$

4.2.2. Correctness Ratio. To measure the effectiveness of our method, we define the correctness ratio $\mathscr{C}$ that calculates the number of correct semantics in the third-order s, where $c\left(A_{x,y,i,m}\right) = 1$ when $A_{x,y,i,m}$ is a true semantic; otherwise, $c\left(A_{\mathbf{x},\mathbf{y},i,\mathbf{m}}\right) = 0$. $\mathscr{C}_p$, $\mathscr{C}_d$, $\mathscr{C}_o$, and $\overline{\mathscr{C}}$, respectively, shows the correctness ratio on the order p, d, o and the average correctness ratio.

$$\mathscr{C}_p = \frac{\sum_i c_p\left(A_{x,y,i,m}\right)}{\left|S_{\text{recovered}}\right|},\ \mathscr{C}_d = \frac{\sum_i c_d\left(A_{x,y,i,m}\right)}{\left|S_{\text{recovered}}\right|},\ \mathscr{C}_o = \frac{\sum_i c_o\left(A_{x,y,i,m}\right)}{\left|S_{\text{recovered}}\right|}, \tag{10}$$

$$\overline{\mathscr{C}} = \frac{\sum_i c_{\mathbf{p}}\left(A_{\mathbf{x},\mathbf{y},i,\mathbf{m}}\right) + \sum_j c_{\mathbf{d}}\left(A_{\mathbf{x},\mathbf{y},j,\mathbf{m}}\right) + \sum_k c_{\mathbf{o}}\left(A_{\mathbf{x},\mathbf{y},k,\mathbf{m}}\right)}{3\left|S_{\text{recovered}}\right|}. \tag{11}$$

4.3. Case Study on Carly.

Our entire workflow on Carly APP is shown in Figure 7, and detailed processes are described as follows.

4.3.1. Decompilation. We firstly disassemble the Android application code and convert it into intermediate languages, such as Jimple format or Smali format. Different types of code transformation are implemented according to the specific analysis tools. Based on these decompiled codes, we perform a further analysis.

4.3.2. Construction of Application Code Graph. Based on static analysis technology, we analyze the calling relationship among functions in Android application code and build the function call graph (FCG) in which each point in FCG represents a function and each edge represents the calling relationship among functions. The calling relationships are described once the FCG is established. Further, for each function in the Android application code to construct control flow graph (CFG), each point in CFG represents a continuous code block and each edge represents a possible branch execution path relationship, such as those branches caused by control instructions if and switch.

4.3.3. Execution Path Search. In both FCG and CFG, the search algorithm is utilized to find all possible execution paths targeting the critical function calling behavior. Firstly, according to selected key functions, it selects all the calling sequences of those functions by searching the paths on the FCG; then, search all the control flow conditions related to the key function according to the CFG of the function in each calling sequence and obtain the corresponding paths which also needs to make sure their acyclicity.

4.3.4. Execution Control. Modify the control flow conditions in each selected execution path to ensure the correct execution process of application following the selected path. Modify the entry function of the Android software, and turn it directly to the selected execution path in above step. Then, modify all the execution flow conditions on the path to ensure that the key function will eventually be called. This process can be done with code insertion technology, and each path will produce a new application.

Input: 4-order tensor A
Output: ($List\_p(A)$ [1], ..., $List\_p(A)[|p|]$), ($List\_d(A)$ [1], . . ., $List\_d(A)[|d|]$), ($List\_o(A)$ [1], . . ., $List\_o(A)[|o|]$)

(1) for each $p \in p$:
(2) //the number of revealed semantics in p order
$Amount$ = SemanticNum($A(p)$);
(3) $List\_p(A)[p]$.add($Amount$);
(4) **end for**
(5) **return** $List\_p(A)[|p|]$
(6) for each $d \in d$:
(7) //the number of revealed semantics in d order
$Amount$ = SemanticNum($A(d)$);
(8) $List\_d(A)[d]$.add($Amount$);
(9) **end for**
(10) **return** $List\_d(A)[|d|]$
(11) for each $o \in o$:
(12) //the number of revealed semantics in o order
$Amount$ = SemanticNum($A(o)$);
(13) $List\_o(A)[o]$.add($Amount$);
(14) **end for**
(15) **return** $List\_o(A)[|o|]$

Algorithm 1: Max-compute (A).

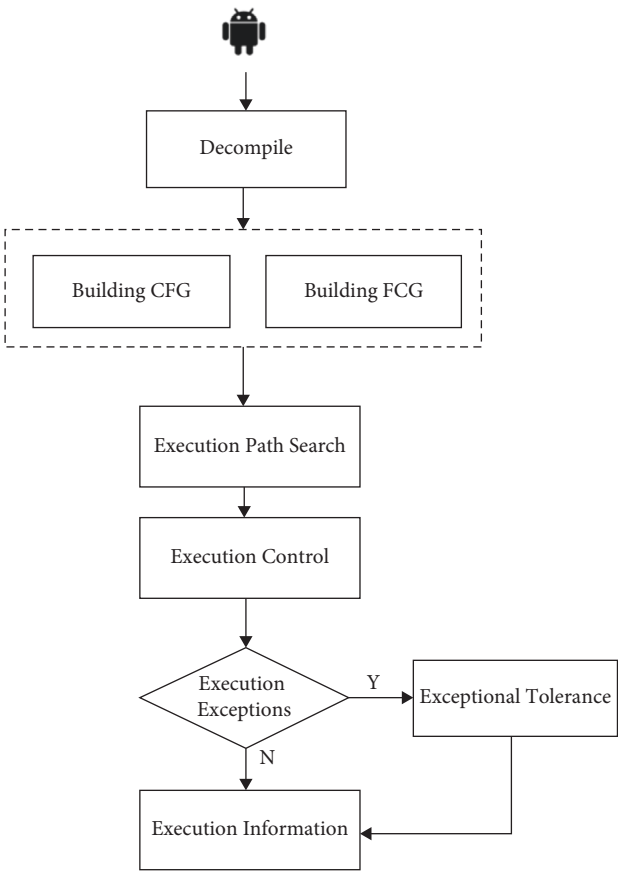

Figure 7: The workflow of reverse analysis on Carly APP.

4.3.5. Dynamic Execution of Exceptional Tolerant. There are many execution exceptions during running when the execution process of application is controlled and some execution conditions are modified. The application software will stop executing immediately if those exceptions are not processed. To guarantee that the code for selected path can be executed, the exception-handling logic of the Android execution environment is modified so that execution proceeds from the next instruction even though the error also continues. For Java code, modify the Android source code to change the exception-handling process in the Dalvik virtual machine execution environment and tolerate the exception to execute from the next instruction. For C/C++ code, similar methods can be used to tolerate exceptions or errors based on virtualization techniques.

4.3.6. Execution Parameter Collection. During the application execution process, the required parameters are collected for further analysis. According to different collected execution parameters, the data are output when the application software is executed, such as the ID identifier data for a CAN message.

The numbers of APP permission, CAN ID, and command and part of contents in Carly are shown in Table 3 in which there are huge numbers of CAN ID and command. In Table 4, we list correspondence between semantic and vehicle models. The first column indicates some semantics, and the second line shows 13 models of Mercedes-Benz. Different models may have different sets of semantics. For example, all of car models have the semantic AB—airbag, model C Stufenheck/Kombi-203 has the semantic WSS—weight sensing system, while G Steilheck/Cabrio—463 does not.

4.3.7. Evaluation on Recovered and Correctness Ratio. The results of recovered ratio are shown in Table 5, and we respectively calculate the value from the three dimensions by equations (7)–(10). There is the highest recovery rate on the dimension of APP permission, followed by CAN ID and

Input: semantic $s = \{s_1, s_2, \ldots, s_{|s|}\}$, cluster number $k = \{k_1, k_2, \ldots, k_{|k|}\}$
Output: $\{c_1, c_2, \ldots, c_{|k|}\}$
(1) **repeat**
(2) $x \leftarrow 1$
(3) //initial setting
$c_x = \{c_{x;1}, \ldots, c_{x;|o|}\} = \{\{s_1\}, \ldots, \{s_{|s|}\}\}$
(4) **repeat**
(5) //initial cluster merging
$(c_\alpha, c_\beta) = \text{argmin}_{c_i \in c_x, c_j \in c_y} \text{Dal}(c_i, c_j)$
$= \text{argmin}_{c_i \in c_x, c_j \in c_y} (1/|C_i||C_j|) \sum_{s_{i \in C_i}} \sum_{s_{i \in C_j}} L(s_i, s_j)$
(6) $New\_c = c_\alpha \cup c_\beta$
(7) $c_x = (c_x \backslash \{c_\alpha, c_\beta\}) \cup \{New\_c\}$
(8) **until** $|c_x| \geq k_x$
(9) $x \leftarrow x + 1$
(10) **until** $x > |k|$
(11) **return** $\{c_1, c_2, \ldots, c_{|k|}\}$

Algorithm 2: Clustering (A).

Table 2: Experimental environment configuration.

Experimental environment	Environmental configuration
Operating system	Windows 11
CPU	Inter(R) Core(TM) i7-10750H CPU @ 2.60 GHz
RAM	16G
Software	Apktool 2.6.1, Dex2jar 2.1, Jadx 1.4.1

command which has a worse result comparably. On the dimension of APP permission, the recovered number of semantics is determined by the other two dimensions CAN ID and command from our context model. Firstly, because both of them have large cardinal numbers 2460×20010, small absence of semantics has little effect on the overall recovery rate. In addition, as shown in CAN frame of Figure 3, CAN ID and command are almost able to determine one semantic except for some special semantics. Therefore, the Recover@1 is up to 99.8%. The huge cardinal numbers also lead to the less decay of Recover@3 and Recover@5 in APP permission. Similarly, when we calculate the rate from the dimension of CAN ID or command, the value is determined by APP permission and command or APP permission and CAN ID. However, on the one hand, one permission may be mapped to multiple semantics that makes it difficult to judge which is the correct semantic the permission mapped. On the other hand, only CAN ID or command cannot uniquely ascertain one semantic. That leads to the worse results in CAN ID and command. The size of APP permission × command is 14×20010, while APP permission × CAN ID is 14×2460. Therefore, the result in command has greater impact which is 66.7% because the smaller cardinal number and its decline trend from Recover@1 to Recover@5 are also the greatest. Overall, we conclude that we can recover the most semantics from the dimension of APP permission or even CAN ID and we obtain a good result of the average recovered ratio which is 84.24%. We also evaluate the correctness ratio by equations (7) and (8). The result validates the effectiveness of our context model that all of the ratios are close to 100%.

Table 3: Part of dimensional tensors in context model.

Tensor type	Content	Number
APP permission	WRITE_EXTERNAL_STORAGE BLUETOOTH_ADMIN BLUETOOTH INTERNET ACCESS_NETWORK_STATE WAKE_LOCK ACCESS_COARSE_LOCATION READ_EXTERNAL_STORAGE ACCESS_WIFI_STATE ...	14
CAN ID	0 × 7E0, 0 × 7E8 0 × 6A3, 0 × 4D4 0 × 632, 0 × 486 0 × 63 B, 0 × 5BB 0 × 62 A, 0 × 485 ...	2460
Command	7B B1 40 FE 24 00 00 00 80 53 20 00 00 00 00 00 0E 00 00 00 08 63 00 00 8A 8 F 10 06 FD 68 8B 01 8B 10 00 03 FF 00 00 48 53 31 38 37 37 30 36 12 05 07 10 11 6E 00 00 ...	20010

4.4. Analysis of Cluster Semantics. The clustering result of semantic is shown in Table 6, where we set the threshold $|k| = 176$. We, respectively, calculate the normalized maximal distance, minimal distance, and mean distance in the

Table 4: Part of distribution among semantics on different vehicle models.

Semantic	Vehicle model												
	A class—176	B class—245	B class—246	C Stufenheck/ Kombi—203	CLS coupe—219	GLK Steilheck—X204	G Steilheck/ Cabrio—463	S limousine—221	AMG/GT roadster—199	SL roadster—231	M Steilheck—164	E coupe/ Cabrio—207	CLA coupe—117
AB-airbag	√	√	√	√	√	√	√	√	√	√	√	√	√
CGW—central gateway	√	√	√	√	√	√	√	√	√	√	√	√	√
TV—TV tuner	√		√		√	√		√					√
WSS—weight sensing system				√	√	√		√	√		√	√	
HS—seat heater					√						√		
SGR—radar sensor control unit						√		√	√	√		√	
FLAL—fire extinguishing system (N100)							√						
TPM—tire pressure monitor (N88)				√	√	√		√	√				
SBC—sensotronic brake control									√				
FSW—windshield wiper (M6/1)	√	√	√	√	√	√	√	√	√	√	√	√	√
KG—keyless go	√		√					√	√	√	√		√
360 camera—60 camera				√	√	√			√			√	
KAB—control unit “camera cover” (N42)													√

three dimensions, in which distance is referred to the average value of $Dal(c_i, c_j)$ between pairwise semantics across all the semantics using equations (5) and (6). For the complete data of semantic in the last line, the max distance represents the distance between the two farthest data in our context model which is up to 1, while the min distance is 0.014 and the mean distance is 0.603. The maximal mean distance 0.893 appears in the dimension of APP permission that indicates it is widely distributed with its data in this dimension. On the dimension of command, its mean distance is smaller that means it has a concentrated distribution. Therefore, this clustering result shows that the data of semantic are decentral or diverse on the dimension of APP permission that concludes most of the data while those are concentrative on the dimension of command that only concludes part of data.

5. Related Work

5.1. Attack Vectors of CAN Security Analysis. As an essential part of modern automobile, CAN is responsible for coordinating the various and sophisticated ECUs of the vehicle to control sub-systems like steering, braking, doors, and windows properly. It is also exploited to develop many applications, such as remote control, vehicle diagnosis, security monitoring, vehicle hacking, and autonomous driving, which has improved driving safety, comfort, and functionality. However, it also causes new attack surfaces to the modern automobile. In recent years, there has been an increasing amount of studies on the CAN Bus attacks. The mobile phone APPs and IVI APPs, middleware of head unit, and CAN Bus form a lot of attack vectors to the automobile by spoofing or injecting CAN Bus commands. There are six categories, mobile APP masquerading, privilege escalation, replay attack, compromising attack, DoS or DDoS attack, and man–in-the-middle attack.

Mobile APP masquerading: The attackers could clone or repackage the legitimate mobile applications straightforwardly by reverse engineering of Android APPs or iPhone APPs [16]. They release the masqueraded APPs with malicious logic on Google Play or Apple App Store, which is not easy to detect [17–21] and obtain vehicle status, manipulate vehicle functionalities, and impact vehicle safety. Privilege escalation: Some middleware APIs which need to be opened to developers are quite security-sensitive. Therefore, it is necessary to audit such critical APIs before invocation. Otherwise, malicious application could invoke these APIs covertly. A malicious application X can successfully invoke the critical APIs by leveraging a flawed design of an open application Y, which has the privilege to use security-sensitive APIs. The study of Han et al. [18] found that Apple private Object C function calls could be compromised by using dynamic loading of functions during runtime. Replay attack: Attackers intercept the communication in one session and retransmit the messages in another session to launch the relay attack. For example, the permissions can be intercepted and reused in malicious application by attackers. The middleware API requests from the legitimate application can be intercepted and resent to the middleware for receiving services. Compromising attack: Attackers exploit the vulnerabilities on both Android system [22] and IOS [18] system to manipulate the mobile applications running on the mobile systems in various ways, such as intercepting the middleware API, replaying, and revising the communication between the application and middleware. Meanwhile, the head unit system could be compromised due to it runs in a more privileged mode and it is vulnerable for remote exploitation. It has been demonstrated that it is feasible to hack into head unit system directly or remotely [2, 9]. DoS or DDoS attack: The attackers could launch the DoS against mobile application by leveraging the compromised mobile system or head unit system to discard the APIs or reject response for the APIs. Also, the middleware APIs could be invoked by the attackers who can query vehicle status and flood messages for vehicle CAN Bus to interfere the vehicle control. In addition, the attackers can launch DDoS attack against vehicle manufacturer facility by controlling a large amount of zombie machines [23] and then make the middleware functionality fail. Man-in-the-middle attack: There are multiple parties in the modern automobile, including vehicle manufacturer facility, mobile application, and vehicle head unit. Attackers could launch the attack in the interaction between any two parties to intercept the encrypted credential of users or obtain the permissions of invoking APIs and reuse such permissions in their own applications. Considering all of the studies reviewed here, it is undoubted that APP reverse engineering is one of the most important techniques to launch a remote attack.

5.2. APP Reverse Engineering. Several previous studies [2, 3, 24–26] have shown that reverse engineering of CAN Bus commands can remotely attack a vehicle. Through a number of possible attack surfaces [5, 9, 27] like Bluetooth, Internet, and APPs, attackers could obtain the CAN packets and replay them back into the CAN to attack the vehicle. Koscher et al. [2] demonstrated that an attacker could observe and reverse-engineer CAN packets, and later inject new packets to induce various attacks. In another major study, Miller and Valasek [24] reported some of attacks via the CAN Bus. An attacker could acquire codes running on ECUs via an attack over Bluetooth, telematics, tire sensor, and physical access to re-send packets and thus to affect the regular work of the automobile. The study by Miller and Valasek [27] identified techniques for reverse engineering CAN Bus commands and demonstrated that attackers could manipulate CAN-enabled components of an automobile. Miller and Valasek [26] in their study showed a remote attack that could form physical control of some aspects of the vehicle by reverse engineering the mechanics tools, ECU firmware, etc. Recently, Li et al. [3] proposed a cost-effective (no real car needed) and automatic (no human intervention required) system, CAN-HUNTER, to realize the reverse engineering of CAN Bus commands. This system only utilizes the companion mobile APPs without using realistic automobiles, and it could perform well in both Android and IOS platforms.

Table 5: Evaluation results of recovered ratio on three dimensions.

Tensor	Recovered ratio		
	Recover@1 (%)	Recover@3 (%)	Recover@5 (%)
APP permission	99.8	99.6	99.5
CAN ID	87.6	87.1	86.9
Command	66.7	65.8	65.2
Total		84.24	

Table 6: Clustering result of semantic using Algorithm 2 ($|k| = 176$).

Dimension	Max distance	Min distance	Mean distance
APP permission	0.992	0.019	0.893
CAN ID	0.986	0.021	0.775
Command	0.994	0.023	0.648
Overall	1	0.014	0.603

However, there are some problems in above researches. For instance, Li et al. [3] had a lack of considering the context. For overcoming it, this work designs a context model of four-order tensor to organize multidimensional contexts trying to avoid the security threat from APP itself-related context.

6. Conclusions

In order to uncover the security threat that overlooking vehicle mobile APP itself-related context, this paper proposes a context-based reverse engineering approach to locate deep hidden commands. We construct a context model with 4-order tensor, including APP permission, CAN ID, semantic, and command, to organize multidimensional contexts, and we also build a continuous updating mechanism. Two algorithms max-compute (A) and clustering (A) are proposed to support RE analysis. We perform the experiment based on the Carly APP with several RE tools and evaluate our approach by two indexes, recovered ratio (Recover@1, Recover@3, Recover@5) and correctness ratio to assess the degree of recovered semantics in different dimensions. The high average recovered ratio and correctness ratio show the effectiveness of our method in the two evaluation metrics. Therefore, there are certain security risks that can be utilized to reverse engineering recall and vehicle mobile APP should pay attention to itself-related context such as the feature information of CAN Bus commands, vehicle application functions, and control diagnostic protocols.

This work is expected to conduct a series of studies on the safety of blockchain automotive mobile APP, including but not limited to (1) research on instruction-semantic mining techniques, (2) matching CAN Bus instruction and corresponding context semantics, and (3) constructing reinforcement learning model toward reverse process.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant nos. 61972025, 61802389, 61672092, U1811264, and 61966009 and the National Key R&D Program of China under Grant nos. 2020YFB1005604 and 2020YFB2103802.

References

[1] P. Mundhenk, *Security for Automotive Electrical/electronic (E/E) architectures [M]*, Cuvillier Verlag, Göttingen Germany, 2017.

[2] K. Koscher, A. Czeskis, F. Roesner, P. Shwetak, and K. Tadayoshi, "Experimental Security Analysis of a Modern Automobile," in *Proceedings of the 2010IEEE symposium on security and privacy*, pp. 447-462, Oakland CA USA, May 2010.

[3] L. Li, J. Liu, L. Cheng et al., "CreditCoin: a privacy-preserving blockchain-based incentive announcement network for communications of smart vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 7, pp. 2204–2220, 2018.

[4] H. Wen, Q. Zhao, and Q. A. Chen, "Automated Cross-Platform Reverse Engineering of CAN Bus Commands from mobile Apps," in *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS'20)*, San Diego California USA, February 2020.

[5] S. Mazloom, M. Rezaeirad, and A. Hunter, "A Security Analysis of an {In-Vehicle} Infotainment and App Platform," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin TX, June 2016.

[6] A. K. Mandal, F. Panarotto, A. Cortesi, P. Ferrara, and F. Spoto, "Static analysis of Android Auto infotainment and on board diagnostics II apps," *Software: Practice and Experience*, vol. 49, no. 7, pp. spe.2698–1161, 2019.

[7] D. L. Strayer, J. M. Cooper, M. M. McCarty et al., "Visual and cognitive demands of carplay, android auto, and five native infotainment systems," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 61, no. 8, pp. 1371–1386.

[8] M. Bozdal, M. Samie, and I. Jennions, "A survey on can Bus protocol: attacks, challenges, and potential solutions," in *Proceedings of the 2018 International Conference on Computing, Electronics & Communications Engineering (ICCECE)*, pp. 201-205, Southend UK, August 2018.

[9] S. Checkoway, D. McCoy, B. Kantor, and B. Anderson, "Comprehensive experimental analyses of automotive attack surfaces," in *Proceedings of the 20th USENIX Security Symposium (USENIX Security 11)*, USA, August 2011.

[10] R. M. Parizi and H. Sajad, "A Blockchain-Based Framework for Detecting Malicious mobile Applications in App Stores," in *Proceedings of the IEEE Canadian Conference of Electrical*

and Computer Engineering (ICCECE), pp. 1–4, Edmonton AB Canada, May 2019.

[11] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2021.

[12] M. M. Moharrer and S. D. Nogoorani, "A Decentralized App Store Using the Blockchain Technology," in *Proceedings of the 17th International ISC Conference On Information Security And Cryptology (ISCISC)*, pp. 14-21, Tehran Iran, September 2020.

[13] "Apktool", "A tool for reverse engineering Android apk files," https://ibotpeaches.github.io/Apktool.

[14] "Dex2jar", "Tools to work with android .dex and java," https://sourceforge.net/projects/dex2jar.

[15] "Jadx", "Dex to Java decompiler," https://github.com/skylot/jadx.

[16] W. Enck, "A study of android application security," *USENIX security symposium*, vol. 2, no. 2, 2011.

[17] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in *Proceedings of the the 2014 ACM International Conference on Measurement And Modeling of Computer Systems*, pp. 221-233, New York NY USA, 2014.

[18] L. Han, A. L. Kashyap, T. Finin, and J. Mayfield, "UMBC_ EBIQUITY-CORE: semantic textual similarity systems," in *Proceedings of the Second Joint Conference on Lexical and Computational Semantics (∗ SEM)*, vol. 1, pp. 44-52, August 2013.

[19] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the android ecosystem," *IEEE Transactions on Mobile Computing*, vol. 19, no. 5, pp. 1184–1199, 2020.

[20] W. Wang, Y. Li, X. Wang, J. Liu, and X Zhang, "Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Generation Computer Systems*, vol. 78, pp. 987–994, 2018.

[21] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 11, pp. 1869–1882, 2014.

[22] Y. T. Lee, W. Enck, and H. Chen, "{PolyScope}:{Multi-Policy} Access Control Analysis to Compute Authorized Attack Operations in Android Systems," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pp. 2579-2596, Auguest 2021.

[23] W. Wang, Y. Shang, Y. He, Y. Li, and J. Liu, "BotMark: automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors," *Information Sciences*, vol. 511, pp. 284–296, 2020.

[24] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *DefCon*, vol. 21, no. 260-264, pp. 15–31, 2013.

[25] J. Staggs, *How to Hack Your Mini cooper: Reverse Engineering Can Messages on Passenger Automobiles*, Institute for Information Security, Japan, 2013.

[26] C. Miller and C. Valasek, *Remote Exploitation of an Unaltered Passenger Vehicle*, Black Hat USA, 2015.

[27] C. Miller and C. Valasek, *A Survey of Remote Automotive Attack Surfaces*, Black Hat USA, 2014.