Automated Cross-Platform Reverse Engineering of CAN Bus Commands From Mobile Apps

Haohuang Wen
The Ohio State University
wen.423@osu.edu

Qingchuan Zhao
The Ohio State University
zhao.2708@osu.edu

Qi Alfred Chen
University of California, Irvine
alfchen@uci.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—In modern automobiles, CAN bus commands are necessary for a wide range of applications such as diagnosis, security monitoring, and recently autonomous driving. However, only a small portion of CAN bus commands is standardized, and a vast majority of them is developed privately by car manufacturers. Today, the most effective way of revealing the proprietary CAN bus commands is to reverse engineer with real cars, which unfortunately is time-consuming and costly. In this paper, we propose a cost-effective (no real car needed) and automatic (no human intervention required) system, CANHUNTER, for reverse engineering of CAN bus commands using just car companion mobile apps. To achieve high effectiveness, we design an efficient technique to uncover the syntactics of CAN bus commands with backward slicing and dynamic forced execution, and a novel algorithm to uncover the semantics of CAN bus commands by leveraging code-level semantic clues. We have implemented a prototype of CANHUNTER for both Android and iOS platforms, and tested it with all free car companion apps (236 in total) from both Google Play and Apple App Store. Among these apps, CANHUNTER discovered 182,619 unique CAN bus commands with 86.1% of them revealed with semantics, covering 360 car models from 21 car manufactures. We have also evaluated their correctness (both syntactics and semantics) using public resources, cross-platform and cross-app validation, and also real-car testing, with which over 70% of all the uncovered commands are validated. We observe no inconsistency in cross-platform and cross-app validation. While there are 3 semantic inconsistency among 241 manually validated CAN bus commands from public resources and real-car testing, we find that these three cases are actually caused by mistakes from app developers.**

I. INTRODUCTION

In modern automotive systems, nearly all vehicle control functions ranging from steering, acceleration, braking, to lighting are controlled by a variety of Electronic Control Units (ECUs) communicating over in-vehicle networks. In such networks, Controller Area Network (CAN) is the most widely deployed communication protocol today [38]. Consequently, the knowledge of the syntactics (*i.e.*, the structure, the format, and the concrete value) and semantics (*i.e.*, the meaning and functionality) of its command messages, or *CAN bus commands*, is crucial to many applications such as automation, diagnosis, and security. For instance, parsing and constructing CAN bus commands are the most basic building blocks for in-vehicle fault diagnosis [56] [52], vehicle security testing [38] [27] [43] [32], security monitoring [47] [28], and recently programmable vehicle control (*e.g.*, autonomous driving) [15].

Although CAN bus commands are highly valuable, only a small portion of them is standardized and the vast majority of them is developed privately by car manufacturers. As a result, there are usually completely different CAN bus commands across different car models [20]. Over the years, a significant amount of effort has been made to reverse engineer CAN bus commands for different car models. Currently, the state-of-the-art is to rely on dynamic testing with a real car by analyzing the repeated correlation patterns between specific CAN messages and vehicle behavior. In particular, there are two ways to do so: (i) one is to manually trigger physical actions in the car, *e.g.*, stepping on the throttle, and then observes changes on the CAN bus [13] [4]; (ii) the other is to generate and send arbitrary CAN messages to the CAN bus, and then observe the triggered physical actions on the vehicle [38] [39]. Unfortunately, these approaches not only require a huge amount of hardware resources, *e.g.*, real cars and testing equipment such as jack stands [38] and CLX000 CAN analyzers [4], but also are time-consuming and error-prone due to the significant manual efforts involved.

However, recently we have witnessed a rapid growth of the Internet of Things (IoT) in many application domains including automobiles for various reasons such as convenience and automation. Today, car manufacturers and 3rd-party developers have produced a great number of mobile apps to connect with in-vehicle systems such as in-vehicle infotainment (IVI) and offer convenient app-based vehicle control or diagnosis. Interestingly, to achieve compatibility with existing in-vehicle systems, these apps fundamentally still rely on the CAN bus commands to engage with the vehicles. That is, these apps must contain the logic to generate CAN bus commands either directly or indirectly (through translation or relay in a dongle or cloud server) for each supported vehicle. As a result, such a newly-formed automotive mobile app ecosystem can create a new direction for reverse engineering of CAN bus commands.

To demonstrate this new approach, in this paper we present a cost-effective (no real car needed) and automatic (no human intervention required) system, CANHUNTER, to reverse engineer the CAN bus commands (targeting both syntactics and semantics) from car companion apps. To achieve high effectiveness, we have to address several technical challenges. First, to recover the syntactics, we need to locate and analyze the CAN bus command generation logic in car companion

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24231
www.ndss-symposium.org

apps. However, since there is neither standard guidance nor unified programming interfaces for CAN bus command generation, how to systematically locate this logic is non-trivial. Moreover, even with such logic located, it is unclear how to trigger the generation process without the required execution contexts such as real cars. To overcome these challenges, we use a technique known as backward slicing that starts from the standardized low-level network programming interfaces to locate the generation paths. Meanwhile, based on the insight that these apps typically embed CAN bus commands without requiring sophisticated external input, we adapt dynamic forced execution techniques (*e.g.*, [60] [36] [34]) to forcefully execute the CAN commands generation paths without real cars.

In addition to revealing the syntactics of CAN bus commands, CANHUNTER also focuses on recovering the semantics to facilitate the practical usage of the commands. Prior efforts on protocol semantics recovery rely on either the keywords in the network traces (*e.g.*, [57]) or the parameters and execution contexts of well-known APIs (*e.g.*, [41] [25] [24]), and they mostly focus on desktop applications or malware and cannot be directly applied in our targeted domain. Interestingly, we notice that CAN bus commands are often triggered by users when using the mobile apps, and the UI-rich apps must provide clues in their user interfaces (*e.g.*, using an unlock door button to trigger unlock door CAN bus commands). Therefore, by leveraging the semantics clues in the app, we design an app code based semantics recovery algorithm to reveal the semantics of the CAN bus commands.

We have implemented a prototype of CANHUNTER and applied it to test all of the free car companion apps we were aware of from both the official iOS and Android app markets in April 2019. In total, we have 236 apps: 114 for iOS and 122 for Android. Among them, CANHUNTER successfully uncovers 182,619 unique CAN bus commands, with 157,296 of them (86.1%) uncovered with semantics from 107 (45.3%) apps: 104 third-party dongle apps and 3 official manufacturer apps. These results cover over 360 popular car models from 21 car manufactures. For the rest 129 (54.7%) apps, CANHUNTER identified indirect vehicle control and monitoring related commands from 87 IVI apps and 22 dongle apps (note that these 109 apps do not directly generate CAN bus commands and instead they rely on cloud servers or dongles to finally generate the real CAN bus commands), and failed in 20 dongle apps due to obfuscation.

To validate the correctness of these reverse-engineered CAN bus commands, we have exhaustively tried all possible methods we could access and afford, including checking with public resources (*e.g.*, those from car hacking forums), cross-platform and cross-app validations, and also real-car testing. With these methods, we were able to successfully validate over 70% of the uncovered commands. Specifically, in cross-platform and cross-app validation, we observe *no inconsistency* in command syntactics and semantics, which implies high effectiveness of CANHUNTER. From public resources and real-car testing, we discover only 3 (1.2% among the 241 manually validated CAN bus commands) false positives in semantics recovery. However, these 3 false positives are actually all caused by mistakes from app developers instead of CANHUNTER. The rationale of why CANHUNTER succeeds, and also the countermeasures against our approach if necessary are also discussed in the paper.

Contributions. The main contributions of this paper are:

- **Novel approach.** We propose a novel, cost-effective, and automatic cross-platform reverse engineering approach for CAN bus commands through analyzing *only* the companion mobile apps, without using real cars.
- **Effective techniques.** We design a suite of new and effective techniques that uncover CAN Bus command syntactics with backward slicing and dynamic forced execution, and infer the command semantics with a novel app code based semantics recovery algorithm.
- **Implementation and evaluation.** We implemented CANHUNTER and evaluated it on 236 car companion apps, and discovered 182,619 unique CAN bus commands in which 86.1% of them are recovered with semantics. We also validate the correctness of over 70% of the recovered CAN bus commands of their syntactics and semantics using public resources, cross-platform and cross-app validations, and real-car testing.

Roadmap. The rest of this paper is organized as follows. We provide the necessary background related to CAN bus commands, automotive mobile apps, and the applications of CAN bus commands in §II. Next, we present a running example to illustrate the challenges and insights when reverse engineering the CAN bus commands from mobile apps in §III. Then, we describe the detailed design of CANHUNTER in §IV and implementation in §V. In §VI, we present the detailed evaluation results of the reverse-engineered CAN bus commands, followed by the discussion on the root causes and countermeasures in §VII. We review related works in §VIII, and finally conclude in §IX.

II. BACKGROUND

A. CAN and CAN Bus Command

For a modern automobile, there are approximately hundreds of Electronic Control Units (ECUs) responsible for controlling various sub-systems such as steering, acceleration, braking, doors, and windows. To coordinate these sophisticated components, the CAN bus is designed for connecting the ECUs and ensuring that the whole system works properly. Messages that can be understood by all the components inside the network are sent back and forth for communications, which are called CAN bus messages. These messages are formed with a specific standardized structure [33], which is presented in Figure 1. Note that the identifier and the command data in the data field altogether determine the function of the message, which are commonly used to identify a CAN bus message. In this paper, we use the term CAN bus command to represent both the syntactics (*i.e.*, identifier and format of a command data) and the semantics (*i.e.*, the human understandable meaning) of a CAN bus message.

The syntactics of a CAN bus command is usually presented as hexadecimal values, including its identifier and data. The identifier can be either 11 or 29 bit referring to the specific ECU that emits the command [33]. The command data contains various length of bytes ranging from 0 to 8, each

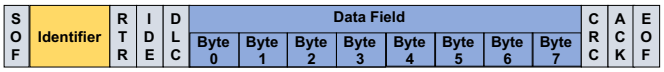

Fig. 1: The structure of a CAN bus message.

of which stores the parameter of the command. The semantics for CAN bus commands can be generally classified into two categories: (1) *control*, which operates physical components of a vehicle such as unlocking doors and starting the engine, and (2) *diagnosis*, which queries the data of a vehicle such as speed and temperature. For each semantic meaning, there is a one-to-one mapping to a CAN bus command. For instance, for Toyota Prius, the CAN bus command identifier "`0x750`" refers to the main body and "`0x7C4`" is for the air conditioning.

The specific syntactics and semantics of most of the CAN bus commands are not standardized, and thus they are defined privately by car manufacturers to represent various functions for different car models. This makes CAN bus commands highly diversified, and even different car models of the same brand (*e.g.*, Audi A3 and A4) can often have completely different set of commands. For example, the command identifier "`0x3B7`" represents engine for BMW E65 but window for BMW E84, according to the validated commands from a car hacking forum [20].

B. Automotive Mobile App Ecosystem

Today, there are many automotive related mobile apps available to consumers. We have investigated car companion apps on popular mobile app markets such as Google Play Store and Apple App Store, and found that they all can be categorized into the following two types according to how they connect to an automobile.

In-Vehicle Infotainment (IVI) apps. This kind of app is developed and authorized by the car makers (*e.g.*, Audi, and Toyota) and is only compatible with specific types of cars. Typically, they offer remote control capabilities such as unlocking doors and starting engines. To interact with the cars, these apps need to connect to the IVI system via either Bluetooth or cellular network. The IVI system itself, as shown in Figure 2a, is widely deployed in most of modern vehicles. The outer interface of an IVI system offers touchscreen for drivers to interact with the cars to play music, navigate and so on, while the other side of it connects to the inner network of the vehicle. To perform remote control functions, IVI apps typically send requests to a remote server on the Internet, *e.g.*, a cloud server, and rely on the server to issue corresponding commands to the vehicle.

OBD-II dongle apps. In addition to car manufactures, some 3rd-party service providers including many auto-insurance companies also develop car companion apps to interact with vehicles through the On Board Diagnostic (OBD-II) port. In general, these OBD-II dongle apps (short as dongle apps) connect to a dongle plugged into the vehicle, and these OBD-II dongles work like wireless sensors connecting to the OBD-II dongle apps via Bluetooth, Wi-Fi, or cellular network and at the same time, directly communicate with the inner CAN

(a) An IVI System

(b) An OBD-II Dongle

Fig. 2: An IVI system and an OBD-II dongle.

of the vehicle [59]. A typical OBD-II dongle is shown in Figure 2b. There are different types of dongles on the market. Some of them are compatible with different automobiles (*e.g.*, dongles provided by auto-insurance companies), and only provide diagnosis or monitoring functions based on a set of standardized OBD diagnostic protocols. Other ones are designed for specific types of automobiles, offering not only diagnosis capabilities but also remote control functionality. Similar to OBD-II dongles, dongle apps also have different types, with some designed for many different dongles such as DashLink, EOBD-Facile, and AutoDoctor, and others only for specific dongles such as Carly, Carista, and FIXD.

C. Applications of CAN Bus Commands

CAN bus commands are essential to many applications including but not limited to:

- **Remote control.** CAN bus commands can be used by mobile apps to offer remote vehicle control, which provides great convenience for users to remotely operate their vehicles such as locking doors and closing windows. For example, we find various car companion apps such as Carista, BimmerCode, and OSCC [17] that interact with vehicles by sending remote control CAN bus commands.
- **Vehicle diagnosis.** CAN bus commands are also used for vehicle diagnosis and inspection such as monitoring the status of vehicles including reading parameters like fuel, pressure, and speed. For instance, we have noticed that various apps and dongles are developed for this purpose, including those mentioned in §II-B.
- **Security monitoring.** CAN bus commands are also widely used in developing CAN bus firewalls (*e.g.*, [35] [48] [47] [44] [28]), to prevent malicious message from being injected into the CAN bus. Without the knowledge of each specific CAN bus command, the security monitoring system would not work properly.
- **Vehicle hacking.** Since CAN bus commands are essential for vehicle control and diagnosis, they can be leveraged to perform attacks on automobiles [44] [45] [46]. For instance, by injecting the corresponding CAN bus commands to an IVI system, Miller and Valasek [46] demonstrated how to shut down the engine of a Jeep Cherokee. Generally, for an attacker with little knowledge about the CAN bus protocol, she could interfere the control of a vehicle by sending random CAN bus messages. However, in order to achieve desired attack consequences on a vehicle, such as shutting down the engine or unlocking doors, one must know the precise CAN bus commands of interest in advance.
- **Autonomous driving.** CAN bus commands are also crucial for the development of autonomous driving nowadays.

We have examined the source code of the three notable autonomous driving software: ApolloAuto [2], Autoware [7] and Openpilot [16]. We noticed the only software interface that interacts with the vehicles is the corresponding CAN bus commands, such as braking, steering, acceleration, and checking tire pressure and fuel. Since CAN bus commands are diversified across vehicle models, ApolloAuto [2] only supports Lincoln MKZ, and Autoware only supports a few models such as Toyota Prius. At this point, these state-of-the-art autonomous driving software platforms can only support a very limited set of car models, which is partly due to the lack of the knowledge of the CAN bus commands for other car models [21].

III. Overview

A. Running Example

To understand clearly the challenges we have to address, we present a running example from an iOS dongle app called Carly f.Toyota to show exactly how a CAN bus command is generated and used. This example is illustrated in Figure 3 with a piece of decompiled code and also the UI component of the app. In particular, the UI shows a button titled "`Engine Controls`", which guides users to check the status of the engine control units of the vehicle. To achieve such functionality, the app must send a specific CAN bus command to the vehicle and fetch data from it. When the button is clicked, the control flow goes to the triggering function where a constant string "`0x7E0`" is initialized (line 4). Then, the string goes through a series of operations and a complete CAN bus command syntactics "`7E0 30 00 02`" is produced (line 19). Finally, the command is sent to the vehicle through a low-level API `writeValue` (line 42).

This example helps us better understand how the syntactics and semantics of a CAN bus command can be inferred. Since the CAN bus commands eventually will be captured by a specific network API and sent to the vehicle, we can adopt a known backward slicing algorithm that starts from these low-level APIs and traces back to locate the commands. Then we can follow the execution to recover the syntactics of the commands. Interestingly, the semantics and the car model information (if an app supports multiple cars) can also be inferred. In this example, the string "`Engine Controls`" appears in the text of the UI button (line 14) as well as the argument of function call `initWithRequestId("0x7E0", "Engine Control")` (line 4). The constant string "`Corolla VIII`" is also integrated as an argument of function `initWithName` (line 13), associating the command with the car model `Corolla`.

B. Technical Challenges

While the motivating example in Figure 3 concretely illustrates the potential of leveraging car companion apps for CAN bus command reverse engineering, how to systemically perform such reverse engineering is still non-trivial. More specifically, we notice there are still three key technical challenges we have to address:

Precise identification of the generation path. For car companion apps, there is neither standard guidance nor unified

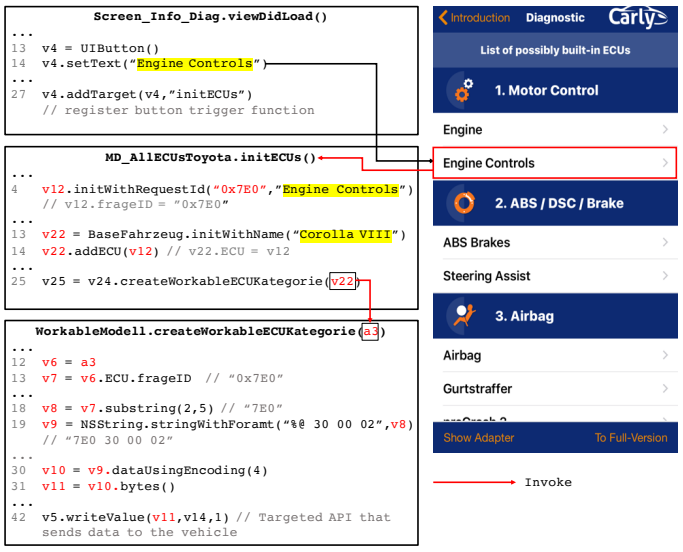

Fig. 3: A motivating running example illustrating the generation of a CAN Bus command in a dongle app.

programming interfaces for developers to construct a CAN bus command. Thus, each car companion app is actually highly customized in the construction logic due to different coding practices. However, to perform our reverse engineering task, identifying the CAN bus command generation path at the car companion apps code level is a necessary step. As a result, how to design a general and systematic solution for such identification is the first challenge.

Automatic syntactics recovery. After identifying the generation path, the operations along the path need to be executed in order to recover the corresponding CAN bus command syntactics. To perform such a task, a known challenge is automatic preparation of the required execution context, *e.g.*, internal program states such as user account, and external input such as those from user or network [42] [29]. Specific to car companion apps, we find that all the ones we collected (detailed in §VI) require successful network connections with actual IVI systems or OBD-II dongles before their vehicle control functions can be accessed. In addition, even with network connections, the majority of them is found to require user authentication or vehicle authentication, *e.g.*, by providing a valid VIN number. Since many of these inputs are difficult to obtain without access to actual hardware such as real cars, the context preparation task for car companion apps become difficult to automate.

Automatic semantics recovery. To complete the reverse engineering process, in addition to the CAN bus command syntactics, we also need to extract the associated semantics, *e.g.*, "`Engine Controls`" in Figure 3. In prior works on protocol semantics recovery (*e.g.*, [24] [25] [57] [38]), dynamic network traces are needed in the recovery process. However, without real cars or dongles to generate network traffic, these prior solutions cannot be applied directly to car companion apps. Another possibility is to find out the semantics from the documentations of the vehicle control APIs used in the generation path. However, no such unified programming

interfaces exist today for vehicle control behaviors in car companion apps.

C. Key Insights

Fortunately, in this work we are able to identify the following insights to address each of the challenges described above:

Backward program slicing. To identify CAN bus command generation path, a key insight is that no matter how these commands are generated, a car companion app will always send out them to the vehicles through network interfaces, *e.g.*, WiFi and Bluetooth, by design. Therefore, we can first locate the standardized network APIs and then use a well-known program slicing algorithm to trace backward from these APIs to find all the code-level operations related to the generation of CAN bus commands. Such a backward slicing based technique has been widely in mobile app analysis including our own prior works (*e.g.*, [63] [64]). Another approach to solve this problem is using forward data flow analysis which starts from UI components and traces to the network sinks. However, the backward slicing approach is preferred for two reasons. First, the backward approach is more precise, since the forward approach explores more program paths including those irrelevant to CAN bus commands generation. Second, the backward approach can discover more CAN bus commands that are not triggered by the UI. For example, as discussed in §VII, many commands we discovered are never used in the app (*e.g.*, some dead code) and are possibly left there for debugging during the app development.

Dynamic forced execution. As described earlier, it is indeed very difficult to prepare the proper execution context of car companion apps, *e.g.*, network connections and user/vehicle authentications. However, we find that the actual construction process of the CAN bus commands is usually independent from external input, after the checking logic of these operational contexts. One such an example is illustrated in Figure 3. This is very likely because CAN bus commands are hex strings and car companion apps can hardly rely on end users to directly enter them, and instead they are embedded in the app code somewhere and rely on human beings to trigger them. Such triggering operations would also not require sophisticated input from users (typically through buttons, lists, and clicks) in mobile apps.

Therefore, this creates a unique opportunity to adopt a widely explored technique called forced execution [60] [50] [36] [37] [34], which is designed for brute-force executing certain part of a program without providing concrete inputs or setting proper environment. While this technique normally has limited effectiveness since it does not consider the execution contexts (*e.g.*, the heap has to be properly modeled, otherwise it will often lead to crashes [50]), such limitation does not affect our problem due to the independence of CAN bus command generation on external input. Thus, by adapting this technique to our problem context, we can achieve automatic and effective syntactics recovery without any real car or dongle connection.

UI and function argument assisted semantics recovery. Since we cannot directly use existing protocol semantic inference approaches with desktop programs (*e.g.*, [24] [25] [57]), we have to look for potential clues inside the app code for our semantics recovery. Encouragingly, we have identified a few common patterns with semantic information inside the app code. In particular, we find that car companion apps typically contain strings that are visible in the app UI to inform users about the semantic meanings of the vehicle control functions. For example, the CAN bus command generation path is associated with the UI element "`Engine Controls`" in Figure 3. As such, we can recover the semantics of a given CAN bus command by tracking the association of its generation logic in the app code with the corresponding strings in the UI elements.

In addition, we also find that functions in car companion apps often take both a string parameter and a CAN bus command together in the generation path. For example, at line 4 in Figure 3, where both "`0x7E0`" and "`Engine Control`" are passed as parameters to function `initWithRequestId`. Thus, we can then use an association heuristic to infer the semantic meanings of the generated CAN bus command from this function. In the car companion apps we collected in §VI, we find this type of argument association widely exist (about 30% of the apps that have CAN bus commands), which very likely is implemented for logging and debugging purposes.

Based on these observations, we therefore design a app code based semantics recovery algorithm that analyzes such semantics clues in car companion app code, *e.g.*, from UI elements and function argument associations, to automatically recover semantics of a CAN bus command.

D. Scope and Assumptions

In this work, we focus on car companion apps from the two mainstream mobile app platforms namely Android and iOS, from their corresponding official app market namely the Google Play Store and the Apple App Store. We assume that these apps are not obfuscated, such that they can be disassembled and decompiled by the state-of-the-art analysis tools such as IDA-Pro [14] and Soot [19]. The implementation of CANHUNTER also assumes car companion apps send out CAN bus commands through Wi-Fi, Bluetooth, and Bluetooth Low Energy (BLE), and thus focuses only on the data transmission APIs of these channels.

IV. DESIGN

The workflow of CANHUNTER is presented in Figure 4. At a high level, CANHUNTER is divided into three components: backward slicing (§IV-A), syntactics recovery (§IV-B), and semantics recovery (§IV-C). It first takes the binary code of a mobile app as input, disassembles and decompiles it, and produces the execution paths through backward slicing of the decompiled code. Then, it uses dynamic forced execution to run the apps with the execution path of interest, from which to uncover both the syntactics and semantics of CAN bus commands. In this section, we present the detailed design of each of these components.

A. Backward Slicing

Since not all the code in a mobile app contributes to the generation of CAN bus commands, we use backward slicing to identify the code path of our interest such that our analysis can be performed efficiently. Backward slicing needs to begin

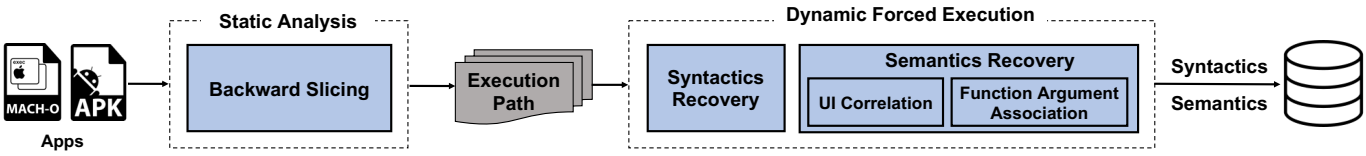

Fig. 4: Overview of CANHUNTER.

Algorithm 1: Backward slicing

```
   Input  : F: current block name, V: a set of target variables, G: the
            CFG of F
   Output: A node containing the block name and instructions
1  Function recursiveSlice(F, V)
2      S ← ∅
3      for each stmt i ∈ G do
4          if i is a relevant statement of any variable in V then
5              if i is a programmer-defined function then
6                  V' ← target variable set of function i
7                  S ← recursiveSlice(i, V').S ∪ S
8              else
9                  if i contains externally defined variables then
10                     for each external variable setter function F'
                       do
11                         V' ← target variable set of F'
12                         S ←
                             recursiveSlice(F', V').S ∪ S
13                     end
14                 S ← i ∪ S
15                 V ← V removes the target variables in i
16                 V ← V∪ all local variables in i
17             end
18         end
19     end
20     node ← Node(F, S)
21     for each caller F' of F do
22         V' ← set of target variables in the caller
23         node.addChild(recursiveSlice(F', V'))
24     end
25     return Node(F, S)
```

from some low-level interfaces. To identify these entry points, we find that CAN bus commands are usually sent to vehicles from apps through wireless network such as Bluetooth and Wi-Fi which have fixed low-level network APIs. Therefore, the backward slicing algorithm in our targeted domain first identifies these documented interfaces and initializes a set of target variables which carry the data to be sent (*i.e.*, the data-use). Then, it backward iterates the decompiled program statements (examples of such statements are shown in Figure 3) and produces the execution paths according to how the target variables and their closures are generated (*i.e.*, trace back to the data-definition).

The pesudocode of how CANHUNTER generates the backward slice S of a function block F with a given target variable set V is presented in algorithm 1. The set V is path-sensitive where each control flow path maintains its own copy of variable set and propagates it backward. Initially, the algorithm takes F, V and the control flow graph (CFG) G of F as input, and sets the slice S as empty (line 2). To begin with, the algorithm backward iterates each statement i in G (line 3-19) and checks if i is a statement affecting the value of any variable in set V (line 4). If so, it is regarded as a relevant statement. For these statements, the algorithm further detects whether i is a programmer-defined function. If so, the algorithm recursively jumps into it to slice that function and finally appends S with the returned slice (line 5-7). Otherwise, i is either a standard library function call or a basic operation (*e.g.*, numeric operation or assignment), and the algorithm detects if there is any externally defined variables (*e.g.*, global variables) in i. If so, it further backward slices the external variable setter functions to find out the data definitions of these variables (line 9-13). Finally, it records statement i into S, and removes all target variables in i and adds all other local variables in i into the trace set V (line 14-16).

After the algorithm traverses all the statements in G, some of the target variables in V may flow into the previous block. In this case, the algorithm continues to trace them in the callers to see how they are generated (a context-sensitive inter-procedural analysis). To achieve this, we first initialize a node containing the current block information including the block name and the program slice S (line 20), and properly set up another target variable set V' which indicates the target variable locations in each caller (line 22). Then a recursive call of itself is invoked on each of the caller F' with V' (line 23). Eventually, the algorithm produces a node which is marked by the block name F and its slice S (line 25), which has multiple children nodes returned from the recursive call of the callers F'. Ultimately, after an initial call on callers of the network API, the algorithm generates all the program slices on the CAN bus command generation paths.

During the slicing, it is important to make sure the resulting program slices do not miss any statements that generate the data of our interest. We therefore consider both data dependence and control dependence in our slicing:

- **Data dependency.** It is quite a standard procedure to identify the data dependency, based on how data is used and defined. With G of F, we backward iterate each statement starting from the statement of our interest: if a variable used belongs to V, or a variable in V gets defined, we add it to the target set and the statement into the slice S.
- **Control dependency.** As our objective is to identify all possible entry points such that our forced execution can exercise them, our algorithm considers control dependencies in order to conservatively include all the possible cases. This process is also quite simple. Specifically, when encountering branches and loops, there are conditions which are relevant to the variables in set V. As such, we add the condition statement into the slice S, and all the variables in the loop or branch body are included in V as well.

Execution path generation. The backward slicing generates a tree-based structure including the execution paths of CAN bus commands. Each node of the tree represents a block and contains its program slices. The leaf nodes of the tree

represent the blocks where the algorithm terminates, which means the target variables are initialized without introducing extra dependencies. We call the leaf node as execution point where the forced execution starts. The root node of the tree refers to the target low-level network API. From each of the leaf node to the root, there must be a path which generates a message that sends out through a network API. Therefore, we traverse the whole tree from the leaf to the root recursively using depth-first traversal, and construct the execution paths by joining the program slices of each node.

B. Syntactics Recovery

The recovery of syntactics is to compute the concrete value of CAN bus commands, which is achieved through forcefully executing the instructions involved in the execution paths. Since the instructions are independent of external inputs, they can be directly executed. Specifically, the execution starts from the leaf nodes of the tree and finally ends in the root. Within each node representing specific blocks, the algorithm executes each instruction based on the order they appear in the slice. The forced execution is performed dynamically on a real mobile device running a car companion app. Each invocation of the network sending APIs generates a CAN bus command through our dynamic forced execution. We log this command and keep its value (which implicitly captures the structure and format) as its syntactics.

We have to note that our design is general and we may identify other non CAN bus commands. For instance, we also observe some AT commands [54] that are sent from mobile apps to OBD-II dongles, and some privately defined commands (*e.g.*, human-readable strings or decimal numbers) sent to IVI systems or remote clouds. Fortunately, as we have described in §II, a CAN bus command always has a distinct structure [33], which thus enables us to easily filter those that are inconsistent with the defined structure.

C. Semantics Recovery

During the syntactics recovery of CAN bus commands through forced execution, CANHUNTER also infers their semantics and also vehicle model information when available. As discussed in §III, there is no accurate way to uncover the semantics of the CAN bus commands, and thus we design a semantics recovery algorithm based on our empirical observations: the semantic meaning of a CAN bus command often appears as a constant string in the app code, and these strings will not be recognized or used by either the OBD-II dongles or the CAN inside the vehicles. The reason for developers to integrate these human-understandable semantics into the car companion apps is to help users associate the CAN bus commands with the related functionalities. As such, our semantics recovery algorithm uses the following two heuristics: (i) the UI component correlation and (ii) function argument association, to infer the semantics of CAN bus commands.

(I) UI component correlation. Users often trigger car related operations with UI components such as buttons in the car companion apps. The texts of the UI components are helpful for guiding them to the specific operations. For instance, a button titled "`Reading Speed`" triggers a speed reading command sent to the vehicle, and then the app will fetch the data and display it to the user. During backward slicing, the algorithm terminates when it reaches all the data definition points. At this time, the semantics recovery algorithm continues to trace backward from the data definition to find the associated UI elements. Then, CANHUNTER extracts the texts of these elements as the command semantics.

(II) Function argument association. The semantics of the CAN bus commands are also likely to be exposed by function calls with their arguments. For instance, as in the function call `initWithRequestId ("0x7E0", "Engine Controls")` of Figure 3, the CAN command syntactics "`0x7E0`" appears together with a constant string "`Engine Controls`" (the semantics) as an argument. Interestingly, this technique is also useful for recovering car model information. Due to the diversity of CAN bus commands, we notice app developers often integrate car model information as arguments to help themselves distinguish the CAN bus commands from different manufacturers. To infer the model information, we cross-compare the extracted arguments with a pre-built set of all vehicle models. Therefore, when executing a function call instruction, CANHUNTER dynamically extracts the constant string arguments of the function calls associated with the command to infer the semantics and model information.

V. IMPLEMENTATION

We have implemented CANHUNTER[1] for both Android and iOS platforms with $3,000$ and $2,000$ Lines of Code, respectively. In this section, we describe the implementation details on how we identify the target APIs, perform both static analysis and dynamic forced execution, as well as how we handle the native library issue.

Target API identification. To start the backward slicing, we focus on two categories of low-level APIs defined by Google's and Apple's official SDKs, which may take CAN bus commands as parameters. Since car companion apps usually interact with vehicles via wireless network, the first category is the BLE API (`setValue` in Android and `writeValue` in iOS). These APIs are provided by the official framework [6] [3] for developers to implement communication between an app and the peripheral following the BLE standard. Besides BLE, CAN bus commands can also be transmitted through Wi-Fi or Bluetooth Classic. Consequently, the second category of low-level APIs includes the socket APIs such as `write` for both Android and iOS, and the HTTP APIs such as `openConnection` for Android, and `dataTaskWithRequest` for iOS.

Backward slicing. The backward slicing of CANHUNTER is implemented based on IDA-Pro [14] and IDAPython (for iOS) as well as Soot [19] (for Android), targeting decompiled Soot IR or Objective-C code. As the two of the most popular static analyzers for Android and iOS, they both provide rich APIs for basic program analysis such as decompilation, building call graph, and locating function callers, which allows us to easily perform static backward slicing of the app code to trace the variables from the target APIs. In addition, they support programming language binding so we are able

[1]The source code is available at https://github.com/OSUSecLab/CANHunter

to develop scripts based on Java and Python to automate the static analysis. Furthermore, we also implemented multi-threading to analyze various executables simultaneously, which significantly accelerates the reverse engineering process.

Dynamic forced execution. The implementation of forced execution is based on Cycript [8] and Frida [12] which are dynamic code instrumentation toolkits compatible with various platforms including Android and iOS. They provide JavaScript interfaces that allow us to dynamically execute the instructions instead of implementing the execution process by ourselves. To achieve automatic execution, we use the Python binding to automatically load each instruction in the execution path and generate a snippet that is injected into the running app to execute these instructions. The environment is set up by connecting our PC to an Android device or a jail-broken iPhone through SSH and hooking the running app process.

The dynamic forced execution may lead to abnormal behaviours and even crashes. Therefore, we added exception handling rules to make sure it can automate without human intervention. When the app crashes or does not respond, CANHUNTER restarts the app and retries the instruction when a timeout limit is reached. Our implement of the exception handling is also based on Frida, which enables CANHUNTER to automate the operations on the attached process without any human intervention.

Handling native libraries. It is common for developers to integrate native libraries in their apps due to its convenience for developing cross-platform apps. A typical example is the `.so` library written in C/C++. In addition, since these libraries are more difficult to reverse engineer, developers can hide important code and data in them. However, with such native libraries used in apps, CANHUNTER is not able to directly construct a complete function call graph when tracing back to the UI components to recover the semantics of the CAN bus commands. Therefore, we designed a solution which enabled CANHUNTER to successfully recover the semantics of over 50% of all the commands. Specifically, we have to address two additional challenges when using disassemblers in IDA Pro to statically disassemble and analyze the native binaries in both Android and iOS platforms.

- **Locating callers of virtual functions**. In the native binaries, the invocation of a virtual function is through referencing the virtual function pointer in its virtual function table. To achieve this, the assembly code first loads the address of the virtual function table, and computes the virtual function address by adding an offset value. Therefore, the caller of a virtual function cannot be directly inferred. To solve this issue, CANHUNTER starts by finding the virtual function table address based on the virtual function pointer, and then it scans through all the references of the table address and checks which virtual function is called to eventually locate the actual caller of the virtual function.
- **Associating the native code with the app code**. For iOS apps, the native code is mixed with the Objective-C code so their association can be explicitly inferred in the same binary. However, for the Android apps, the Java byte code and the libraries are separated, and the invocation of native functions is through the Java Native Interface (JNI) from the Native Development Kit (NDK) [1]. In this circumstance, CANHUNTER recognizes the native functions in Java byte code and associates them with the related ones in the native code according to the function signatures. Afterwards, CANHUNTER constructs the function call graph of the native binaries and then associates the native functions with the app code, which thus establishes a complete call graph.

	# Total	# Diagnostic App (%)	# IVI App (%)
Android	122	74 (60.7%)	48 (39.3%)
iOS	114	72 (63.2%)	42 (36.8%)
Total (Android ∪ iOS)	236	146 (61.9%)	90 (38.1%)
Overlapped apps (Android ∩ iOS)	79	38 (48.1%)	41 (51.9%)

TABLE I: Distribution of collected car companion apps.

VI. EVALUATION

A. Experiment Setup

Car companion app collection. To perform our evaluation, we have crawled all of the free vehicle related apps we were aware of from both the official Apple App Store and Google Play Store in April 2019. Specifically, we used a crawler to start from some known keywords (such as OBD-II, IVI, and vehicle mobile apps) and exhaustively collect apps from the relevant app pages. Then, we manually confirmed each crawled app to make sure they are vehicle related (for instance, we can further filter those dongle manual apps that never interact with a vehicle, or general dongle terminal apps that just provide a terminal for expert users), and meanwhile classify them according to the types described in §II. In total, we eventually collected 122 and 114 car companion apps from Android and iOS platform respectively, including 146 dongle apps and 90 IVI apps. Table I shows the distribution of these apps. Note that there were also 71 paid apps on both app markets, but we did not collect them due to our budget constraints.

Experiment environment. We ran the static analysis with the 236 tested apps on a 10.14 MacOS MacBook Pro with six Intel Core i7 CPU and 16GB RAM as well as a Linux server running Ubuntu 16.04 equipped by twelve Intel Core i7-8700 CPUs and 32 GB RAM. The dynamic forced execution is performed on a Google Nexus 4 with Android 7.0, as well as a jail-broken iPhone 6 with iOS 10.3.3.

B. Experiment Result

In total, CANHUNTER discovered 182,619 CAN bus commands[2] from 107 out of the 236 apps we tested. Among them, 157,296 commands (86.1%) are recovered with semantics. Although not all the commands are recovered with semantics, we would like to emphasize that *the automatic syntactics recovery provided by* CANHUNTER *is already useful* since it can accelerate the reverse engineering process compared to existing approaches that rely on manual efforts or fuzzing to construct CAN bus commands in real car testing [13] [38]. Note that CANHUNTER also discovered standardized CAN bus commands called OBD PIDs [53] which are for diagnostic purposes and are ubiquitous in the diagnostic apps. Since these

[2]The reverse engineered commands are also available at https://github.com/OSUSecLab/CANHunter.

App Name	Size (MB)		# Commands		Car Model Recovered		Semantics Recovered		Backward Slicing				Forced Execution				Semantics Recovery			
									Slicing Cost (s)		# Branches		Execution Cost (m)		# Instr.		By UI %		By Function Argument %	
Dongle apps:																				
Carista	12	8	105,198	105,198	100%	100%	100%	100%	343	729	105,301	105,301	71	78	257,673	257,673	26%	26%	74%	74%
Carly for VAG	58	18	18,627	16,402	44%	66%	61%	56%	153	512	19,236	17,638	31	28	112,313	105,137	100%	10%	0	90%
Carly for BMW	67	38	14,377	16,427	100%	100%	100%	100%	125	502	15,132	18,898	27	29	97,354	113,494	100%	0	0	100%
Carly for Toyota	50	19	5,305	39	100%	99%	100%	100%	64	235	6,405	706	7	11	23,741	4,034	100%	100%	0	0
Carly for Renault	47	18	5,199	1,255	100%	96%	0	3.5%	50	197	5,284	1,384	8	15	30,189	5,557	0	100%	0	0
Carly for Mercedes	51	20	7,921	1,698	82%	79%	100%	100%	88	203	8,021	1,704	9	14	32,437	4,878	0	0	100%	100%
Carly for Porsche	46	18	1,963	278	100%	97%	0	0	17	155	2,122	384	1	4	4,234	1,521	0	0	0	0
BimmerCode	4	2	0	42	0	100%	0	100%	4	15	20	90	0	3	78	986	0	100%	0	0
BlueDriver	24	8	304	304	100%	100%	100%	100%	27	64	313	313	1	4	1,245	1,360	100%	100%	0	0
CarVantage	87	15	41	41	0	0	0	0	21	24	45	45	1	1	48	0	0	0	0	0
ANCEL	26	3.7	1	16	0	0	0	94%	14	11	41	34	1	1	101	292	0	60%	0	40%
CHIPOBD	10	3	0	29	0	0	0	76%	1	19	0	70	0	3	0	1,004	0	0	0	100%
Dr.OBD	4	2	0	2	0	0	0	100%	0	2	0	23	0	1	0	328	0	0	0	100%
inCarDoc	15	18	160	160	0	0	0	100%	39	50	251	251	1	7	2,555	2,273	100%	100%	0	0
iOBD2	57	5	5,007	218	0	57%	100%	100%	47	84	5,034	240	3	5	12,304	1,684	0	4%	0	96%
Kiwi OBD	7	4	220	6	0	0	100%	100%	23	20	227	227	1	1	252	260	100%	100%	0	0
MOSX	6	6	13	4	0	0	100%	0	9	9	71	25	1	1	280	127	100%	100%	0	0
OBDPlus	8	18	0	24	0	0	0	100%	0	22	0	92	0	1	0	558	0	0	0	100%
OBD Mate	34	4	1	11	0	0	0	100%	17	11	42	34	3	1	104	29	0	55%	0	45%
SekurTrack OBD	5	2	10	18	0	0	100%	100%	13	11	57	46	1	1	765	410	100%	100%	0	0
Spinn	26	6	0	32	0	0	0	32%	0	11	0	14	0	1	0	60	0	0	0	100%
Engie	28	11	144	68	0	0	100%	100%	40	30	98	15	1	1	1,540	170	100%	100%	0	0
Easy OBD	3	1	28	5	0	0	100%	82%	13	10	50	18	1	1	542	108	100%	100%	0	0
Savy Driver	65	21	15	0	0	0	0	0	22	1	227	1	1	0	1,040	2	0	0	0	0
DashLink	44	28	5	0	0	0	100%	0	107	1	9	4	2	0	2,361	11	60%	0	40%	0
Car Scanner	51	31	4	0	0	0	0	0	43	1	14	1	1	0	386	2	0	0	0	0
DashCommand	8	47	3	0	0	0	33%	0	8	1	112	4	1	0	195	11	100%	0	0	0
SekurLot	8	2	2	0	0	0	50%	0	4	0	52	0	1	0	121	0	100%	0	0	0
IVI apps:																				
HondaLink	49	8	0	52	0	100%	0	100%	14	19	108	83	0	2	2504	716	0	0	0	100%
HondaLink Aha	12	5	0	52	0	100%	0	100%	23	14	60	80	0	2	278	666	0	0	0	100%
Land Rover Comfort	21	3	0	19	0	100%	0	100%	18	11	183	98	1	1	1058	106	0	0	0	100%

TABLE II: Experiment results from CANHUNTER for the car companion apps available on both Android and iOS platforms (overlapped apps that have CAN bus commands). Numbers on the left are for for Android and on the right are for iOS.

commands are well-documented and have distinct features with reserved identifiers, we are able to distinguish them from the non-standardized CAN bus commands. Overall, these documented OBD PIDs take up approximately 15% of our results, while the rest 85% are customized CAN bus commands.

Among the 107 car companion apps that have CAN bus commands recovered by CANHUNTER, interestingly 49 of them exist in both Android and iOS platforms (with the same app name). The distribution of the reverse-engineered CAN bus commands from these car companion apps is presented in Table II, which summarizes the statistics of CAN bus command syntactics and semantics recovered from each app, as well as the detailed intermediate running statistics including the cost of backward slicing, dynamic forced execution, and the total number of branches and instructions. For the remaining 58 apps that are available on only one of the Android and iOS platforms, their results are presented in Table III.

1) **Result Characteristics by App Categories:** Among the 107 car companion apps that expose CAN bus commands, only 3 of them are IVI apps while the other 104 are dongle apps. There are some interesting findings on these apps described as follows:

IVI apps. The 90 IVI apps in our experiment indeed provide remote vehicle control functions, but we find that CANHUNTER produced CAN bus commands from only 3 of them. For these three apps, we discover that their developers accidentally integrate the CAN bus commands into the apps, because these commands in fact can never be triggered and sent out from the app, though CANHUNTER is able to find them from the network APIs. We further investigated all other IVI apps and find that this is actually not due to the false negatives of our system. In fact, the IVI apps do not directly generate CAN bus commands for the supported control functions on the app side, but rely on a third entity to interpret the requests to CAN bus commands. Particularly, a majority of them is found to first send the control commands, usually in the form of constant strings such as "`UNLOCK`" for door unlocking, to a cloud server, and rely on the connection between the cloud server and the vehicle to control it. In this paper, we call this type of commands *interpreted commands*. Among the 90 IVI apps, we found that 82 (91.1%) of them adopt this design. For car companion apps with such a design, it is impossible to recover CAN bus commands directly by only analyzing the mobile apps without using the actual cars or at least the actual IVI units. For the rest 8 IVI apps, CANHUNTER did not detect any obvious interpreted string commands and instead detected the URLs for each particular type of commands in the cloud server which will eventually inject the CAN bus commands to the vehicles.

Dongle apps. Among the 146 dongle apps, CANHUNTER was able to discover CAN bus commands from 104 (71.2%) of them, which shows that it is much more common for dongle apps to directly construct CAN bus commands on the app side than IVI apps. We also investigate the reasons for the remaining 42 dongle apps, and find that 22 of them also adopt the design of interpreted commands for CAN bus command construction. Specifically, the OBD-II dongles compatible for these apps are specially designed to translate control request strings from the dongle apps to CAN bus commands. For the rest 20 apps, CANHUNTER did not identify any commands due to obfuscation (*i.e.*, anti-analysis techniques) that prevented our analysis from building a complete CFG in the backward slicing step.

App Name	Size (MB)	#Commands	Car Model Recovered	Semantics Recovered	Backward Slicing		Forced Execution		Semantics Recovery	
					Slicing Cost (s)	# Branches	Execution Cost (m)	# Instr.	By UI %	By Function Argument %
Android apps:										
Obd Harry Scan	5	141	0	100%	19.3	54	0.13	482	100%	0
WEG Motor Scan	12	105	0	100%	14.41	244	0.73	2,638	100%	0
Dr. Prius / Dr. Hybrid	2	31	0	100%	17.65	170	3.00	10,798	0	100%
AlfaOBD Demo	32	27	0	0	211.12	269	1.13	4,058	0	0
Zyme Pro	15	12	0	66.7%	20.51	56	0.06	227	100%	0
OBDmax	6	12	0	25%	20.4	199	0.17	596	100%	0
OBD2 Scaner	6	9	0	0	20.68	84	0.17	604	0	0
Vyncs	24	8	0	0	35.28	229	0.38	1,350	0	0
Torque Lite	6	8	0	100%	14.94	147	0.13	470	100%	0
StarLine	25	8	0	100%	34.51	141	1.55	5,587	100%	0
OBD ECU Access Tester	0.3	7	0	100%	0.9	14	0.02	69	100%	0
CarSys Scan	4	6	0	16.67%	25.86	67	0.04	161	100%	0
Obd Arny	3	5	0	0	15.65	60	0.1	367	0	0
OBD Boy	5	5	0	0	20.43	52	0.09	334	0	0
Best Top NH OBD II 2018	15	5	0	20%	8.1	35	0.06	231	100%	0
GPS7 CLIENTE2	7	4	0	0	5.56	28	0.05	162	0	0
Auto Agent	31	4	0	0	21.7	375	1.34	4,826	0	0
FAPlite Citroen/Peugeot OBD2	3	4	0	0	16.54	72	0.07	254	0	0
RYKA GPS Track	4	4	0	0	9.75	66	0.04	157	0	0
MotorData OBD Car Diagnostics	10	4	0	0	8.11	22	0.04	146	0	0
Clear And Go	4	4	0	0	20.33	35	0.06	233	0	0
OBD Driver Free	3	4	0	0	13.88	150	0.81	2,908	0	0
OBD II SYSTEM	15	4	0	0	8.75	26	0.03	119	0	0
OBDeleven PRO	25	4	0	0	25.41	75	0.11	405	0	0
OBD JScan	38	4	0	0	11.36	74	0.11	411	0	0
ChevroSys Scan Free	4	4	0	0	52.52	17	0.04	136	0	0
PHEV Watchdog	5	4	0	0	15.95	93	0.1	354	0	0
OBD Codes	4	4	0	0	8.32	65	0.05	175	0	0
Volvo Penta Easy Connect	37	4	100%	0	15.96	74	0.09	335	0	0
AutoNiveau	1	2	0	100%	4.64	7	0.11	386	0	100%
Bosch Mobile Scan	45	1	0	0	8.69	75	0.16	583	0	0
Piston	2	1	0	0	6.07	8	0.01	30	0	0
U-Scan	44	1	0	0	7.4	43	0.1	365	0	0
UltraGauge	4	1	0	0	5.11	45	0.18	651	0	0
Garage Pro	10	1	0	0	20.51	56	0.06	227	0	0
Fuel Economy for Torque Pro	6	1	0	0	20.74	118	0.47	1,709	0	0
iOS apps:										
Carly for Partners	35	809	89%	100%	297.48	1,858	20.82	7,495	100%	0
Car2Mobile	2	2	0	100%	0.42	4	0.53	12	100%	0
DrivePro OBD	7	160	0	100%	17.33	245	3.28	1,020	0	100%
ForScan Viewer	2	512	100%	0	80.97	255	6.2	2,350	0	0
FourStroke	1	47	0	100%	2.35	16	2.58	993	100%	0
Gauged	2	25	0	100%	8.46	26	2.85	1,068	100%	0
Konnwei OBD	1	16	0	94%	8.32	43	1.4	533	0	100%
Leagend OBD	3	28	0	79%	14.71	116	4.11	1,580	0	100%
Mini OBD II	3	52	0	100%	11.72	50	0.82	312	100%	0
Smart Connect	4	16	0	63%	1.7	8	0.34	126	0	100%
Smart OBD	3	2	0	100%	29.72	36	1.15	440	100%	0
Zhinengpeijia	6	22	0	86%	19.45	218	5.63	2,194	0	100%
V	1	9	0	56%	10.7	39	0.85	333	0	100%
OBD Fusion	49	2	0	0	0.27	3	0.01	6	0	0
MaxiAp200	183	495	0	13%	675.78	8,972	24.78	74,589	0	100%
Diag-Asia	41	322	0	13%	228.15	4,325	9.20	30,374	0	100%
Diag-China	163	585	0	4%	609.73	7,522	20.91	64,815	0	100%
MaxiAp	6	49	0	8%	229.88	3,285	8.30	24,893	0	100%
Diag-Europe	194	788	0	14%	772.64	9,234	21.42	83,545	0	100%
Diag-VW	182	201	0	23%	470.02	7,132	20.45	65,438	0	100%
Auto Diag	119	42	0	24%	405.55	5,132	11.76	43,420	0	100%
Diag-USA	123	313	0	2%	417.32	5,756	13.65	48,576	0	100%

TABLE III: The experiment result for the remaining car companion apps in addition to those in Table II.

2) **Result Characteristics by Car Models:** Overall, CANHUNTER recovered car model information of 161, 819 (88.6%) CAN bus commands, covering over 360 car models from 21 car makers. The distribution of the commands over part of the car makers and models are shown in Table IV. As shown, the reverse engineered results cover most of the popular car makers and models today such as Audi A4, Toyota Corolla, Honda Civic, *etc.*

3) **Result Characteristics by Semantics:** In total, CANHUNTER successfully recovered the related semantics of 157, 296 (86.1%) commands. We manually interpreted and categorized them and found 3, 439 different kinds of semantics. Among them, 1,309 command semantics are for vehicle control while the remaining are for diagnosis. In Table V, we show part of the semantics associated with the top number of recovered commands along with their semantics categories. Typical examples for control semantics are locking doors, sounding horns. Typical examples for diagnosis semantics are reading parameters from the vehicle such as speed, temperature, voltage, *etc.*, which enable users to monitor the status of their cars.

4) **Additional Commands:** Since the target APIs are low-level network programming interfaces, the reverse engineering capability of CANHUNTER is not limited to CAN bus commands but all communication data sent from car companion apps to the vehicles or clouds. In particular, we find that in our results CANHUNTER also discovered 41 unique AT commands from the dongle apps and 267 interpreted commands from the IVI apps, which are presented in Table X and Table XI in Appendix. Note that these commands can be easily distinguished from the CAN bus commands due to the significant differences of the structure and format. Specifically, AT commands are used for configuring the ELM interface of

Car Maker	# Commands	Car Model
Audi	51,517	A3, A4, A5, A6, A7, A8, Q3, Q5, Q7, S3, S4
Volkswagon	44,504	Cabrio, Corrado, Caddy, Gol, Golf, Jetta, Lupo, New Bettle, Passat, Polo, Santana, Transporter, Octavia, Kombi, Roomster
Skoda	11,009	Citigo, Fabia, Rapid, Superb, Yeti
Toyota	9,030	Auris, Avensis, Camry, Corolla, Prius, RAV4
BMW	8,963	Series 1, 3, 5, M5, X5
Seat	8,277	Ibiza, Leon, Altea, Mii, Toledo, Arosa
Mercedes	7,247	Benz
Lexus	6,087	CT200, ES350, GS350, GX460, RX450, IS460
Renault	5,025	Cilo, Megane2
Porsche	2,326	987FL, 997, 996, Cayman
MINI	2,118	Clubman, Cooper
Scion	570	FRS
Chevrolet	198	Lagunall
Subaru	159	Brz
Honda	104	Civic
Bently	60	Arnage, Azure, Brooklands
Lincoln	52	Continental
Ford	26	Galaxy
Lamborghini	20	Gallardo

TABLE IV: Distribution of reverse-engineered CAN Bus commands over part of car makers.

Semantics	# Commands	Category
Engine speed	460	Diagnosis
Coolant temperature	281	Diagnosis
Throttle angle	256	Diagnosis
Oil temperature	176	Diagnosis
Engine load	134	Diagnosis
Boost pressure actual value	133	Diagnosis
Motor temperature	123	Diagnosis
Battery voltage	119	Diagnosis
Comfort function remote incl roof	77	Control
Comfort function key	73	Control
Single door lock remote	60	Control
Blink on unlock key	42	Control
Sound on remote lock volume	40	Control
Blink on lock	37	Control
Auto unlock when moving	27	Control
Marker lights when unlock	24	Control

TABLE V: Distribution of reverse-engineered CAN Bus commands over part of command semantics.

OBD-II dongles [10], which are sent through the Bluetooth or socket interfaces to the dongles. The other type is the interpreted commands from the IVI apps mentioned in §VI-B1. These commands are often interpreted into decimal values or human-understandable strings, and are either pushed to the remote cloud server or to the vehicle.

C. Correctness Evaluation

To evaluate the effectiveness of CANHUNTER, we need to validate the correctness of the CAN bus commands recovered from our experiments. However, due to the high difficulty in obtaining the ground truth (*i.e.*, the publicly known and validated CAN bus commands), performing a comprehensive correctness validation is very challenging for two reasons. First, it is almost impossible to solely use real cars to validate all these commands since it is both inefficient and costly. Second, as introduced in §II, car manufactures and third-party developers treat CAN bus commands confidential and never post any of them in public documents.

Nevertheless, in this paper we tried our best to evaluate the effectiveness from three sources: public resources, cross validation, and real car tests. This allows us to successfully validate 130,011 (71.2%) command syntactics as well as 128,298 (70.3%) command semantics. To summarize, in cross-platform and cross-app validation, we observe *no inconsistency* in command syntactics and semantics. From public resources and real-car testing, we discover no false positive in syntactics recovery and only *3 (1.2%)* false positives in semantics recovery among the 241 manually validated CAN bus commands. Though there are 3 false positives found in the semantic validation, we have confirmed that they are due to programmer mistakes instead of the design or implementation of CANHUNTER (detailed later).

Evaluation criteria and threats to validity. Since the commands across car models are often quite different, we only compare those of the same model. There are three means to evaluate our results: (1) public resources, (2) cross-app and cross-platform validation, and (3) real-car testing. The syntactics is validated by comparing the hexadecimal values of the commands. As for the semantic validation, we compare the extracted semantics strings and thus manual validation is also needed since the comparison is at natural language level.

We have to note that our cross-app and cross-platform validation assumes developers never make the same mistakes of the corresponding CAN bus commands in both apps or both platforms, though it is very unlikely that such mistakes could happen. Similarly, we assume the public available CAN bus commands are also truly validated.

(1) Public resources. While CAN bus commands are confidential, we find that some of them are still publicly available, *e.g.*, posted at car hacking forums, but scattered all over the Internet. We tried our best to search for the ground truths from online forums and documents (*e.g.*, [18] [5] [9] [11] [16]), and picked the commands that are validated by car hackers or researchers through real-car testing. In addition, we also checked the source code of 3 open-sourced self-driving car platforms: ApolloAuto, Autoware, and Openpilot, which may also have CAN bus commands. Among them, we find matched CAN bus commands from Openpilot, but not from the other two code bases since their car models do not match with the ones in our results. Eventually, we collected 69 CAN bus commands in total across 4 car models that are present in our results: Toyota Prius, Audi A3, Seat Ibiza I-Tech, and Honda Civic.

Among these 69 CAN bus commands, 38 of them can find matched syntactics with the commands uncovered in our experiment, which are listed in Table VI. As shown, for these 38 commands, CANHUNTER is able to recover the semantics of 33 (86.84%) commands. Within these 33 commands, 30 (90.91%) have matched semantics, while the remaining 3 are false positives which are indicated by cross marks in the table. For example, for command `0x324`, the correct semantics should be "Water Temperature", but our tool recovered it as "`ENG_TEMP`". We went back to check the app code, and found that the semantics strings associated with these 3 CAN bus commands are indeed correctly recovered by our tool based on our semantics recovery algorithm. Thus, these false positives are actually not caused by the design or implementation of CANHUNTER, but very likely due to programmer mistakes.

(2) Cross validation. Due to the limited number of publicly available ground truth, we also attempted to evaluate the correctness of our system by checking the uncovered CAN bus

Car Model	Syntac.	Semantics (Ground Truth)	Semantics (Our Result)	Matched
Toyota Prius	0x727	Transmission	Transmission	✓
	0x750	Main Body	MainBody	✓
	0x780	Air Bag	Airbag	✓
	0x781	Precrash	preCrash	✓
	0x790	Distance Control	Radar	✓
	0x791	Precrash2	preCrash 2	✓
	0x7A1	Steering Assist	Steering Assist	✓
	0x7A2	Park Assist	APGS	✓
	0x7B0	ABS Brake	ABS	✓
	0x7C0	Instrument	ComboMeter	✓
	0x7C4	Air Conditioner	Air Conditioning	✓
	0x7D0	Navigation	Navigation	✓
	0x7E0	Engine Controls	ECT	✓
	0x7E2	Hybrid System	Cruise Control	✓
Audi A3	0x70C	SteeringWheel	Steering wheel	✓
	0x70A	EPHVA14AU37		
	0x710	GatewLear		
	0x712	SteerAssisMQB		
	0x713	Brake1UDSCondi	ABS Brake	✓
	0x714	DashBoard	Instrument	✓
	0x715	Airba		
	0x746	AirCondiFront	Auto HVAC	✓
	0x773	MUStd4CPASE		
	0x7E0	ECM	Engine	✓
	0x7E1	TCMDQ	Transmission	✓
Seat Ibiza	0x711	ImmoUDS	Immobilizer	✓
	0x713	Brake1ESP	ABS Brakes	✓
	0x714	KombiUDS	Instruments	✓
	0x7E0	ECM	Engine	✓
Honda Civic	0x158	Speed	EAT_TRANS_SPEED	✓
	0x17C	Engine RPM	ENG_STATUS	✓
	0x188		EAT_CHANGE_RESF	
	0x1A3		TM_CHANGE_RESF	
	0x324	Water Tempreature	ENG_TEMP	✗
	0x1A4	VSA_STATUS	VSA_WARN_STATUS_ABS	✓
	0x305	SEATBELT_STATUS	SRS_EDR_DELTA_VMAX	✗
	0x35E	CAMERA_MESSAGES	FCM_WARN_STATUS	✗
	0x391	GEARBOX	CVT_ATF_CHANGE_RESF	✓

TABLE VI: Commands validated with public resources.

commands across different apps from the same or different platforms. Since the CAN bus commands are independent of the car companion apps, the command syntactics and semantics of the same car model should be consistent across different apps, which thus makes it possible to perform correctness evaluation. Although the cross validation does not directly reflect the validness of the commands, it sufficiently implies the effectiveness and generality of our system since the results come from different mobile app implementations.

Cross-platform validation. Among all car companion apps that expose CAN bus commands, 31 of them are available on both Android and iOS platforms. We compared the CAN bus commands that CANHUNTER uncovered from them, and found that 129,266 (70.8%) commands uncovered from 15 of these apps have matched syntactics by cross comparing the hexadecimal values. In Table VII, we show the names for these 15 apps, and the statistics for the total numbers of recovered syntactics and semantics from each platform as well as the number of matched syntactics and semantics. For the remaining 16 apps, CANHUNTER could not extract CAN bus commands from either the Android one or the iOS one, which thus prevents us from validating their results. For example, CANHUNTER found 123 CAN bus commands from the 3 IVI apps in iOS, but did not find any from their Android versions. We manually checked these 3 apps and found that this is due to different implemented logic in the app code instead of analysis inaccuracies in our system. We suspect that this is because their Android and iOS versions are developed by separate teams. Among the CAN bus command pairs with matched syntactics, we then compare the 128,200 pairs (99.2%) where the semantics of both of the commands are recovered. In this

App	Android		iOS		Overlapped	
	# Syn.	# Sem.	# Syn.	# Sem.	# Syn.	# Sem.
ANCEL	1	0	16	15	1	0
BlueDriver	304	304	304	304	304	304
Carista	105,198	105,198	105,198	105,198	105,198	105,198
Carly for BMW	14,377	14,377	16,427	16,427	13,480	13,480
Carly for Mercedes	7,921	6,528	1,698	1,698	1,393	1,393
Carly for Porsche	1,963	0	278	0	7	0
Carly for Renault	5,199	0	1,255	44	1,058	0
Carly for Toyota	5,305	5,266	39	39	39	39
Carly for VAG	16,402	7,283	18,627	10,429	7,283	7,283
CarVantage	41	41	41	41	41	41
Engie	144	144	68	68	68	68
inCarDoc	160	160	160	160	160	160
iOBD2	5,007	5,007	218	218	218	218
Kiwi OBD	220	220	6	6	6	6
SekurTrackOBD	10	10	18	18	10	10

TABLE VII: Statistics of overlapped CAN bus command pairs in cross-platform validation.

Car model	# Overlapped		App1	App2
	Android	iOS		
Audi A3	0	18	Carly for VAG	Carly for Partners
Audi A4	52	52	Carista	Carly for VAG
Audi A6	22	22	Carista	Carly for VAG
Audi A8	0	26	Carista	Carly for VAG
Seat Leon	19	19	Carista	Carly for VAG
Skoda Fabia	0	24	Carista	Carly for VAG
VW Caddy	0	12	Carista	Carly for VAG
VW Polo	52	52	Carista	Carly for VAG
VW Jetta	0	46	Carista	Carly for Partners
VW Passat	0	42	Carista	Carly for Partners
VW Golf	0	168	Carista	Carly for Partners
VW Touareg	0	50	Carista	Carly for VAG
VW Up	0	20	Carista	Carly for VAG
VW Tiguan	8	0	Carista	Carly for VAG
Skoda Superb	0	20	Carista	Carly for VAG
Porsche Cayenne	0	72	Carly for VAG	Carly for Partners
Toyota Prius	39	39	Carly for Toyota	Carista
Toyota Camry	18	0	Carly for Toyota	Carista
Toyota Corolla	21	0	Carly for Toyota	Carista
Porsche	0	4	Carly for Porsche	Carly for VAG
Porsche	0	4	Carly for Porsche	Carly for Partner
BMW 550i	8	8	Carly for BMW	Carista

TABLE VIII: Statistics of overlapped CAN bus command pairs in cross-app validation.

comparison, we observed *no inconsistency*, which shows the high effectiveness of our system.

Cross-app validation. In addition to cross-platform validation, we also perform same-platform cross-app validation. Specifically, we check if there are overlapping command pairs of the same car model from different apps within the same platform (either Android or iOS). Overall, as presented in Table VIII, we are able to find 745 CAN bus command pairs with matched syntactics, which belong to 22 car models from 6 different apps. Among them, 98 (13.2%) pairs have both commands in the pairs recovered with semantics, and *no inconsistent semantics* is observed between these pairs.

(3) Real-car testing. We also tested the uncovered CAN bus commands on the real automobiles that we can access. The two tested car models are Toyota RAV 4 2014 and Toyota Corolla 2014 and the tested app is com.prizmos.Carista which has the same set of CAN bus commands for Android and iOS. We implemented a dynamic testing framework based on Frida and Python, which hooks the target APIs described in §V and captures the CAN bus commands sent from the app to the vehicle. In the experiment, we manually triggered all possible UI components that generate the CAN

Command (RAV4)	Command (Corolla)	Semantics
750 ... 14 1A 26	750 ... 1A 65 02	Wireless door locking
750 ... 14 92 26	750 ... 92 65 02	Blink turn signals
750 ... 14 9A 06	750 ... 9A 45 02	Panic Function on remote
750 ... 14 9A 25	750 ... 9A 61 02	Relock automatically
750 ... 9A 26 00	750 ... 65 02 20	Beep volume
750 ... 14 9A 26	750 ... 8A 65 02	Beep when locking
750 ... 13 00 40	750 ... 98 65 02	Warn beep when sunroof open
750 ... 14 9A 66	750 ... 9A 25 02	Unlock via remote
750 ... 11 00 60	750 ... 14 06 00	Unlock via physical key
750 ... 11 80 20	750 ... 11 C0 20	Unlock when shifting into gear
750 ... 11 80 40	750 ... 11 C0 60	Unlock when shifting into park
750 ... 11 80 70	750 ... 11 C0 70	Unlock when driver's door open
750 ... 3B 15 00	750 ... 00 80 40	Daytime running light
750 ... 3B 12 10	750 ... 3B 12 10	Turn on interior lights
7C0 ... 3B A2 40	7C0 ... 3B A2 40	Display unit (MPG)
7C0 ... 3B 74 A0	7C0 ... 3B A7 C0	Seat belt warning (driver)
7C0 ... 3B A7 C0	7C0 ... 3B A7 C0	Seat belt warning (passenger)
7C0 ... 61 AE 40	7C0 ... 3B A1 28	Key in ignition sound
7C0 ... 61 A7 C0	7C0 ... 3B AF 40	Lane-change signal auto flasher
7C0 ... 61 AB 00	7C0 ... 3B AB 00	ECU Drive indicator zone
7C0 ... 61 AE 40	7C0 ... 3B AE 00	Display odometer
7CC ... 00 00 00	7CC ... 3B 81 00	A/C power
7CC ... 00 01 00	7CC ... 3B 82 00	Fan Speed

TABLE IX: Part of commands validated with real-car testing.

bus commands, intercepted the commands, and compared the testing results with the ones reported from CANHUNTER. In total, we obtained 88 commands from Toyota RAV4 and 84 commands from Toyota Corolla, and all of them match the command automatically discovered from CANHUNTER, which thus concretely demonstrates the effectiveness of our system. In addition, corresponding physical behaviors (*e.g.*, disabling wireless door locking) were observed on the vehicles, which also shows the validness of these commands. Table IX presents a selected part of validated CAN bus commands (46 in total) from these two vehicles.

D. Performance Evaluation

To evaluate the system efficiency, we executed CANHUNTER on the 236 car companion apps in our dataset and collected the running time and intermediate results. During the experiments, we find that CANHUNTER is reliable when analyzing all the apps without any human intervention and crashes. The detailed statistics are shown in Table II as well as Table III and the performance results are broken down into three parts: backward slicing, dynamic forced execution, and semantics recovery. Overall, the backward slicing usually takes several minutes to complete while the dynamic forced execution costs from several minutes to hours. We have broken down the contributions of UI and function arguments to the semantics recovery results, which shows both of them are useful in the recovery process. There are some interesting findings when comparing the apps between the two platforms. For instance, we can notice that the size of the apps tend to be larger in iOS platform than that of Android (*e.g.*, the largest app in iOS is 194M whereas the largest one in Android is only 45M). Second, since there are more commands to recover in iOS apps, their analysis time usually took longer. Finally, it is interesting to notice that function argument association heuristics plays a more critical role for semantic recovery in iOS platform.

VII. DISCUSSIONS AND FUTURE WORK

A. Root Cause and Countermeasure

CANHUNTER has discovered a large set of CAN bus commands from mobile apps. The root cause is that CAN bus commands or their indirect mappings must exist in the app in order to achieve the desired diagnosis or remote control functionalities. Interestingly, as shown in §VI-B1, the exposure level of CAN bus commands differs greatly between dongle apps and IVI apps. In particular, the developers of IVI apps tend to interpret CAN bus commands and thus do not directly integrate them in the app code, while dongle app developers tend to directly hardcode them.

We suspect that the causes of the differences between IVI and dongle apps may be two folds. First, from the design perspective, the IVI systems are far more sophisticated than OBD-II dongles. More specifically, IVI systems usually have cellular network connections and operating systems, which can enable more complicated capabilities such as interacting with a third entity (*i.e.*, the cloud). However, OBD-II dongles usually do not have cellular network and thus completely rely on the input from nearby mobile apps connected through WiFi or Bluetooth. Second, IVI systems and the corresponding IVI apps are typically well-engineered by car manufacturers who are aware of the sensitivity and safety-criticalness of their private CAN bus commands. On the contrary, the dongle apps are usually developed by third-party developers who may not be fully aware of the importance of CAN bus commands. Interestingly, we also find that a small portion of the CAN bus commands are never used in some IVI and dongle apps, since there is no UI correlation to let users trigger them. We suspect they are only for debugging and testing purposes, and thus should have been removed when the apps are released.

To prevent CAN bus commands from being reverse engineered from companion mobile apps, there are two countermeasures. First, developers can leverage interpreted commands which can only be recognized by cloud or device firmware, as demonstrated in the IVI apps. When specific functions are triggered, the interpreted commands are sent to cloud or device which will translate them into valid CAN bus commands. Therefore, reverse engineer can only obtain these manufacture-specific commands instead of the CAN bus commands. Second, anti-analysis techniques such as encryption and obfuscation can be deployed to encrypt the hardcoded CAN bus commands and obfuscate the program control flow, which increases the difficulty of reverse engineering the mobile apps.

B. Limitations and Future Work

While CANHUNTER has uncovered a significant number of CAN bus commands, it still has several limitations. First, due to the limited ground truths and resources, we are only able to validate 70% of our results. Meanwhile, a majority of the validated commands comes from the cross validation and as a result, the evaluation is possibly not 100% correct because the developers can also make mistakes. As mentioned in §VI-C that we observe inconsistent semantics between our result and the public resources even though CANHUNTER functioned correctly. The most reliable approach for validation is testing the commands on real automobiles. In addition, false negative

may exist because CANHUNTER only focuses on the CAN bus commands that sent from the low-level network APIs.

Second, as found in §VI, our current implementation of CANHUNTER is not resilient to anti-analysis techniques such as control flow obfuscation, which can fail our backward slicing. Note that obfuscation is not a specific limitation of our work, and it has long been a challenge for static program analysis and reverse engineering [34]. Nevertheless, there also exists some deobfuscation solutions (*e.g.*, [23]), which we plan to explore and integrate into CANHUNTER.

Third, CANHUNTER reported a great number of AT commands and also interpreted commands during our experiments (§VI-B4). Though not directly involved in the CAN bus network, they can still affect the behavior of the vehicle by interacting with cloud or OBD-II dongle, especially for the interpreted commands which are capable of unlocking the vehicle and shutting down the engine. Therefore, a further study of the security impact of these interpreted commands is worth of exploration.

Finally, we have witnessed the bloom of the IoT industry recently, and there are a great number of IoT devices having companion mobile apps such as those in smart home systems [55] [22] [61]. In order for these mobile apps to communicate with the IoT devices, various IoT protocols are designed. In particular, the car companion apps in this work can be considered a specific case of such IoT companion mobile apps, and CAN is a special type of IoT protocol and the CAN bus commands are the protocol message data that determines the function. Therefore, CANHUNTER has the potential to be extended to reverse engineer the syntactics and semantics of other IoT protocols. In addition, since CANHUNTER adopts dynamic forced execution, the analysis does not require real device connections and thus has the great potential to enable large-scale IoT companion mobile app analysis. As such, exploring the use of CANHUNTER to IoT protocol analysis with companion mobile apps is our another future work.

VIII. Related Work

CAN and vehicle security. CAN has been acting as an essential part of vehicle security research. Previous work has proposed many exploits to remotely control a vehicle as well as reverse engineering the CAN protocol and commands. For instance, Miller *et al.* [45], Checkoway *et al.* [27], and Mazloom *et al.* [43] studied various possible attack surfaces such as Bluetooth, Internet, and apps, which inspires our research. Reverse engineering of CAN bus commands is also an important building block for subsequent attacks on in-vehicle systems. Previous work tried to observe the traffic inside the automotive to obtain the CAN packets and replayed them back into the CAN to attack the vehicle (*e.g.*, [46] [51] [44] [38]). Recently, there are also efforts on the defensive approaches to protect the attacks on CAN such as anomaly detection [28] [48] [49], forensics measures [35] and delayed data authentication [49]. Compared with our work, the previous reverse engineering works on CAN are not comprehensive since each of them only focuses on one or two car models. Besides, our research is novel in that we approach the same reverse engineering problem from a completely different angle by using the car companion apps emerged in recent years.

Protocol reverse engineering and semantics recovery. Reverse engineering of CAN bus commands is a type of network protocol reverse engineering, which is a well studied area especially in desktop applications and malware. For example, Polyglot [26], AutoFormat [40], Discoverer [30], Tupni [31], and ReFormat [58] use information from program executions and/or network traces to reverse engineer network protocols such as HTTP. Besides protocol syntactics, some previous work also studied the recovery of protocol semantics based on the communication traces [62]. Netzob [24], Dispatcher [25] and ProDecoder [57] combine dynamic program analysis and NLP techniques to extract semantics of protocols of interests. Compared with previous works, we have a unique problem of uncovering the CAN bus commands from highly interactive mobile apps, and we solve this problem by using dynamic forced execution to uncover the syntactics and code-level clues, *e.g.*, UI and function argument associations, to uncover the semantics.

Forced execution. Forced execution has been used to discover potential security threats since it can execute programs ignoring regular constraints and external inputs. Recently, a number of forced execution tools were built including J-Force [37] for JavaScript applications, X-Force [50] and Limbo [60] for binaries, and Dexism [34] and Forced-Path execution [36] for Android apps. In these tools, they brute-force execute the program flows by ignoring conditions in order to cover all possible branches. CANHUNTER is inspired by these works, and it adopts forced execution to our particular application domain with a slicing-based forced execution, which executes only the partial instructions involved for generating the CAN bus commands.

IX. Conclusion

We have presented CANHUNTER, the first cost-effective and automatic cross-platform reverse engineering system of CAN bus commands through analyzing only the companion mobile apps without using real cars. It features syntactic recovery of CAN bus commands using backward slicing and dynamic forced execution, and semantic recovery using UI component correlation and function argument association. A prototype of CANHUNTER for both Android and iOS platforms has been developed, and applied to test 236 car companion apps from both app markets. Evaluation results show that CANHUNTER is able to uncover 182,619 unique CAN bus commands of 360 car models from 21 car makers, and recover the semantics of 86.1% of them. Through public resources, cross validation, as well as real car tests, we have validated the syntactics and semantics of over 70% of the recovered CAN bus commands. No inconsistency is observed in cross-platform and cross-app validation, and only 3 false positives (which are caused by mistakes from app developers, not the limitation from CANHUNTER) are discovered in semantics recovery from public resources and real-car testing. The rationale behind CANHUNTER and also the countermeasure against our reverse engineering is also discussed in the paper.

Acknowledgment

We would like to thank our shepherd Manuel Egele and also the anonymous reviewers for their helpful comments that

have significantly improved the paper. This research was supported in part by National Science Foundation (NSF) Awards 1834215 and 1850533. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

REFERENCES

[1] Android ndk. https://developer.android.com/ndk/.

[2] Apolloauto/apollo: An open autonomous driving platform. https://github.com/ApolloAuto/apollo.

[3] Bluetooth low energy overview — android developers. https://developer.android.com/guide/topics/connectivity/bluetooth-le.

[4] Can bus sniffer - reverse engineering vehicle data (wireshark). https://www.csselectronics.com/screen/page/reverse-engineering-can-bus-messages-with-wireshark/language/en.

[5] clear-mind ii: Fit3 hv(gp5)canł. http://clear-mind-mark2.blogspot.com/2016/12/fit3-hvgp5can.html.

[6] Core bluetooth — apple developer documentation. https://developer.apple.com/documentation/corebluetooth.

[7] Cpfl/autoware: Open-source to self-driving. https://github.com/CPFL/Autoware.

[8] Cycript. http://www.cycript.org/.

[9] Detecting anomalies in controller area network for automobiles. http://cesg.tamu.edu/wp-content/uploads/2012/01/VASISTHA-THESIS-2017.pdf.

[10] Elm327 obd to rs232 interpreter. https://www.elmelectronics.com/wp-content/uploads/2016/07/ELM327DS.pdf.

[11] Explorative techniques and vulnerability assessment on automotive networks. https://www.politesi.polimi.it/bitstream/10589/141804/1/2018_07_Calin.pdf.

[12] Frida a world-class dynamic instrumentation framework. https://www.frida.re/.

[13] How to Hack a Car - A Quick Crash Course. https://medium.freecodecamp.org/hacking-cars-a-guide-tutorial-on-how-to-hack-a-car-5eafcfbbb7ec.

[14] Ida. https://www.hex-rays.com/products/ida/.

[15] An introduction to the can bus: How to programmatically control a car. https://news.voyage.auto/an-introduction-to-the-can-bus-how-to-programmatically-control-a-car-f1b18be4f377.

[16] openpilot. https://github.com/commaai/openpilot.

[17] Polysync/oscc: Open source car control. https://github.com/PolySync/oscc.

[18] Raspberry pi - virtual sensors for car computers. http://www.grandprixforums.com/general-grand-prix-discussion/96047-raspberry-pi-virtual-sensors-for-car-computers.html.

[19] Soot - a framework for analyzing and transforming java and android applications. http://sable.github.io/soot/.

[20] Vehicle reverse engineering wiki. http://vehicle-reverse-engineering.wikia.com.

[21] Why ford, lincoln, and lexus testers rule the self-driving roost. https://www.caranddriver.com/news/a15344273/why-ford-lincoln-and-lexus-testers-rule-the-self-driving-roost/.

[22] Amr Alanwar, Bharathan Balaji, Yuan Tian, Shuo Yang, and Mani Srivastava. EchoSafe: Sonar-based Verifiable Interaction with Intelligent Digital Agents. In *ACM Workshop on the Internet of Safe Things (SafeThings)*, 2017.

[23] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 7–12. ACM, 2017.

[24] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 51–62. ACM, 2014.

[25] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451–474, 2013.

[26] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[27] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*, 2011.

[28] Kyong-Tak Cho and Kang G Shin. Fingerprinting Electronic Control Units for Vehicle Intrusion Detection. In *USENIX Security Symposium*, 2016.

[29] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

[30] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[31] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *ACM conference on Computer and Communications Security (CCS)*, 2008.

[32] Roderick Currie. Hacking the can bus: basic manipulation of a modern automobile through can bus reverse engineering. *SANS Institute*, 2017.

[33] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[34] Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. Resilient and scalable cloned app detection using forced execution and compression trees. In *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8. IEEE, 2018.

[35] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive can networkspractical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011.

[36] Ryan Johnson and Angelos Stavrou. Forced-path execution for android applications on x86 platforms. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pages 188–197. IEEE, 2013.

[37] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*, pages 897–906. International World Wide Web Conferences Steering Committee, 2017.

[38] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental Security Analysis of a Modern Automobile. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[39] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 817–821. IEEE, 2015.

[40] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[41] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.

[42] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[43] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. A Security Analysis of an In-Vehicle Infotainment and App Platform. In *Usenix Workshop on Offensive Technologies (WOOT)*, 2016.

[44] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *Def Con*, 21:260–264, 2013.

[45] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *black hat USA*, 2014:94, 2014.

[46] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[47] Michael Müter and Naim Asaj. Entropy-based Anomaly Detection for In-vehicle Networks. In *IEEE Intelligent Vehicles Symposium (IV)*, 2011.

[48] Michael Müter, André Groll, and Felix C Freiling. A structured approach to anomaly detection for in-vehicle networks. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 92–98. IEEE, 2010.

[49] Dennis K Nilsson, Ulf E Larson, and Erland Jonsson. Efficient in-vehicle delayed data authentication based on compound message authentication codes. In *Vehicular Technology Conference, 2008. VTC 2008-Fall. IEEE 68th*, pages 1–5. IEEE, 2008.

[50] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *USENIX Security Symposium*, pages 829–844, 2014.

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

[52] Jittiwut Suwatthikul, Ross McMurran, and R Peter Jones. In-vehicle Network Level Fault Diagnostics Using Fuzzy Inference Systems. *Applied Soft Computing*, 11(4):3709–3719, 2011.

[53] Ashraf Tahat, Ahmad Said, Fouad Jaouni, and Waleed Qadamani. Android-based universal vehicle diagnostic and tracking system. In *2012 IEEE 16th International Symposium on Consumer Electronics*, pages 137–143. IEEE, 2012.

[54] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem. In *USENIX Security Symposium*, pages 273–290, 2018.

[55] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. Smartauth: User-Centered Authorization for the Internet of Things. In *USENIX Security Symposium*, 2017.

[56] Jiande Wang, Yunshan Zhou, and Quan Li. Research on Fault Diagnostic System in CVT based on UDS. *Advances in Mechanical Engineering*, 7(1):128432, 2015.

[57] Yipeng Wang, Xiaochun Yun, M Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–10. IEEE, 2012.

[58] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[59] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[60] Jeffrey Wilhelm and Tzi-cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*, pages 219–235. Springer, 2007.

[61] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[62] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Li Zhiqiang. Geo-locating drivers: A study of sensitive data leakage in ride-hailing services. In *Annual Network and Distributed System Security symposium, February 2019 (NDSS 2019)*, 2019.

[63] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

[64] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1469–1483, 2019.

APPENDIX

The interpreted commands reported from CANHUNTER are presented in Table XI which shows the distribution of the 267 interpreted commands with the total 41 IVI apps that adopt this design combining all duplicated ones for Android and iOS. The interpreted commands are often represented by human understandable strings (*e.g.*, `UNLOCK`, `LOCK`) and numbers (*e.g.*, 2 for unlock and 0 for flash light). In particular, we also distinguish different implementations of the IVI apps. Some of them push the commands to the cloud which forwards control messages to the automobile, while others directly send the interpreted commands to the vehicle via Wi-Fi or Bluetooth.

Table X shows the distribution of the AT commands with the 20 dongle apps that expose them. In the experiment, we only reserve one command for each kind of syntax, and totally we have discovered 41 types of AT commands. Specifically, the AT commands are sent through Bluetooth or socket APIs for configuring the ELM327 interface [10] of the OBD-II dongles. For instance, command `AT AL` is for restricting the number of data byte in a CAN bus message to 7 [10].

App	# Command	The Detailed Commands
Carly f. BMW	24	`ATE, ATH, AT FC SH...`
Carly f. Toyota	23	`ATE, ATH, AT PB A...`
Carly f. MB	24	`ATE, ATH, AT ST FA...`
Carly f. VAG	24	`ATE, ATH, AT CEA...`
Carly f. Partners	18	`ATSH, ATMX, AT ST FF...`
Carly f. Renault	24	`ATE, AT FC SD, AT PB...`
Carly f. Porsche	24	`ATE, AT FC SD, ATH...`
Carista	17	`ATCEA, ATD, ATFSCH...`
SekurTrack	5	`ATED, ATE, ATA...`
BlueDriver	18	`ATE, ATA, ATL, ATSP...`
Dr.OBD	2	`ATE, ATCH`
StarLine	7	`ATE, ATA, ATH, ATCH...`
ezOBD	18	`ATI, ATE, ATCRA, ATL...`
ANCEL	12	`ATI, ATE, ATRA, ATCM...`
ForScanViewer	35	`ATCEA, ATCRA, ATSH...`
FourStroke	10	`ATZ, ATE, ATA, ATH...`
Gauged	17	`ATED, ATD, ATP, ATZ...`
iOBD2	20	`ATE, AT ST, AT CA F...`
LeagendOBD	12	`ATE, ATB, ATTR, ATQ...`
Engie	8	`ATE, ATSC, ATI, ATST...`

TABLE X: AT commands extracted from dongle apps.

App	# Command	Content	Sent to Cloud	Sent to vehicle
AcuraLink	9	HORN_LIGHT, HORN_ONLY, UNLOCK, LOCK, LOCATION ...	✓	
Alpine	2	frontSpeakerPattern, rearSpeakerPattern		✓
Alpine TuneIt	3	RESUME, PHONE_DIAL_END, AUDIO_FOCUS	✓	
Audi MMI Connect	10	LOCK, UNLOCK, G_STAT, FIND_CAR, G_DLIST...	✓	
Carbin Control	15	Climate_Control_Temperature, Control_Fan_Speed...		✓
Car-Net	4	Unlock:2, Lock:3, Flash:0, Hornlight:1		✓
Companion	2	UNLOCK, LOCK		✓
Mini Connected Classic	1	PlayListCommand:0x88	✓	
Nissan Connect Services	19	RemoteServiceDoorLock, RemoteServiceHornBlow...		✓
E20 Remote	12	UnlockCar, TurnOnCharging, ManageHVAC...	✓	
Mini Connected	7	UNLOCK_DOORS, FUEL_STATE_MAX, HONK_HORN...		✓
Nissan Leaf	5	REMOTE_DOOR_UNLOCK, HORN_LIGHT, LIGHT_ONLY...		✓
Ford Play	3	VPlayState:1/2/3		✓
Genesis	8	RemoteFlashLight, EditSpeed, RemoteStart...	✓	
Infinity Connection	8	REMOTE_DOOR_LOCK, REMOTE_STOP, LIGHT_ONLY...	✓	
Infinity InTouch Services	8	REMOTE_DOOR_LOCK, REMOTE_STOP, LIGHT_ONLY...	✓	
Kia Hands-On	4	open. close, horn, alert	✓	
Lexus Enform Remote	4	Unlock:1, Lock:0, Start:1, Stop:0	✓	
Mahindra Blue Sense	4	Audio: ED0D3EE, Climite:ED0EEE...		✓
Mercedes Me	4	Unlock:1, Lock:0, Start:2, Stop:3	✓	
Mazda Mobile Start	4	Lock, Unlock, Start, Stop, Horn	✓	
MyBuick	6	Unlock:3, Lock:2, PanicAlarm:6, RemoteStart:4		✓
MyCarKia	2	EngineStart, EngineStop	✓	
My Ford Mobile	4	UNLOCK_CMD, START_CMD, CANCEL_START_CMD...	✓	
MyHyundai	9	start:41, stop:42, lock:40, unlock:43...	✓	
My Mittsubishi Connect	5	HORN_REMOTE_OPERATION, ENGINEOFF_REMOTE...	✓	
NissanConnectServices	8	REMOTE_DOOR_LOCK, REMOTE_STOP, LIGHT_ONLY...	✓	
Porsche Car Connect	8	D_DOORS_LOCKING, D_OPENINGS, DC_MIRROR_CLOSE...	✓	
OnStar RemoteLink	9	lockDoor, unlockDoor, start, alert, diagnostics		✓
Lexus RES+	4	start, stop, lock, unlock	✓	
Hyundai SmartRemote	3	Volume, Channelist, Follow TV		✓
Ford Remote Access	3	unlock_ford, start_foed, lock_ford	✓	
Tesla	7	UNLOCK, HONK_HORN, FLASH_LIGHTS	✓	
Uconnect	4	Unlock, Lock, Start, Stop	✓	
UVOeco	9	Lock Doors, Stop Climate, Horn & Lights Activation...	✓	
Volvo On Call	4	Unlock, Lock, Start, Stop	✓	
Toyota Entune Remote Connect	4	Unlock:1, Lock:0, Start:1, Stop:0	✓	
Tesla Plus	17	remote_start_drive, sun_roof_control, trunk_open...	✓	
HondaLink	6	LOCK, UNLOCK, START, STOP, HORN_ONLY, LIGHT_ONLY	✓	
HondaLink Aha	6	LOCK, UNLOCK, START, STOP, HORN_ONLY, LIGHT_ONLY	✓	
Land Rover Comfort Controller	19	Fan:1024, LeftSeatBack: 1007		✓

TABLE XI: Interpreted commands from IVI apps.