DETROIT: Data Collection, Translation and Sharing for Rapid Vehicular App Development

Mert D. Pesé[+], Dongyao Chen[*+], C. Andrés Campos[+], Alice Ying[+], Troy Stacer[+], and Kang G. Shin[+]
[+]Department of Computer Science and Engineering, University of Michigan
[*]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University
{mpese,chendy,andcmps,acying,trstacer,kgshin}@umich.edu

***Abstract*—DETROIT is an open-source vehicle-agnostic end-to-end framework for vehicular data collection, translation and sharing that facilitates the rapid development of automotive apps. With vehicles becoming increasingly connected, unlocking sheer amounts of data from the in-vehicle network (IVN) can accelerate the development of many useful apps. Unlike existing commercial and academic solutions that can only access a restricted set of standardized emission-related sensor data and lack feasible data accessibility by third-party developers, DETROIT offers a convenient interface to develop apps which can access a broad range of powertrain-related sensors and car-body events thanks to crowd-sourcing vehicular translation tables by fully automated CAN bus reverse-engineering.**

DETROIT is developed with the objectives of simplicity, scalability, privacy and liability. To the best of our knowledge, this is the first end-to-end framework consisting of a frontend, backend and a developer portal to cover vehicular data collection, translation and sharing with app developers. Besides an extensive framework benchmark to show the light resource overhead and feasibility of DETROIT, we also have evaluated it by re-implementing two existing mobility apps from academia. Developers have reported that DETROIT offers high sensor fidelity, enhanced application flexibility, as well as low implementation complexity.

***Index Terms*—data collection, CAN bus, automotive apps**

I. Introduction

Connected vehicles are estimated to generate 25 GB of various data per hour, including data from cameras, LiDARs, radars, etc. The in-vehicle network (IVN) itself, consisting of multiple network buses, such as the Controller Area Network (CAN), produces only a fraction of this amount of data, but still carries highly valuable and critical sensor information. Data collected from two CAN buses in 2017-2019 model vehicles were in excess of 170 MB per hour.

The generation and sharing of this driving data will create an additional source of revenue for OEMs and third-party services. According to PwC, the connected-car market is expected to grow to $155.9B by 2022 [1]. OEM-independent, universal access to data by third-party services can help the latter in automotive data monetization. Third-parties already offer vehicle dongles that can access the IVN and obtain publicly available data (OBD-II PIDs [2]). In particular, usage-based insurance (UBI) companies [3], [4] have already been distributing dongles to track their insurees' driving behavior. Furthermore, new companies such as Carloop [5] promise third-party app support to develop and deploy vehicular apps. Otonomo [6] wants to be the middleman to connect third-party developers with data collected by OEMs. Finally, academic solutions such as CarTel [7] have also been proposed for data *collection* from cars, but not for data *translation* or *sharing* with third-party developers.

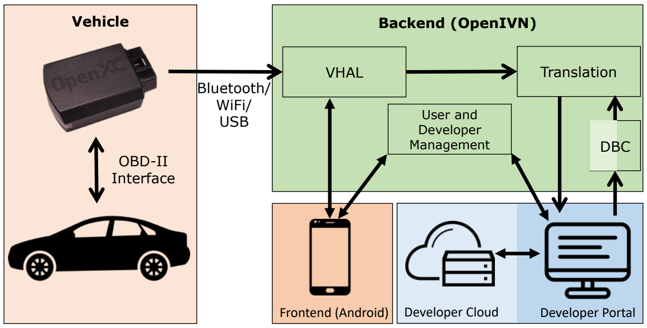


Fig. 1: System diagram of DETROIT: End User (orange), Developer (blue) and OpenIVN (green).

Vehicular data is not only a growing target for monetization by the above commercial solutions, but also enabling academic and basic research. Numerous mobile sensing apps for automotive safety, as well as awareness of privacy concerns on rich vehicular context data have recently emerged (see Sec. III-C). Note that these apps rely on phone sensor data, with mobile IMU data being the most popular. Considering these developments, vehicle mobility is poised to become a main stream of future mobile computing and networking.

Despite several available solutions and tools for vehicular data *collection* as mentioned above, one may wonder why these platforms are not currently widely used by academic researchers and commercial companies. We present below a list of limitations of existing solutions that explained the limited adoption:

(1) Data Sources. All commercial and academic data collection platforms proposed thus far operate using the OBD-II protocol. Some emission-related sensors are standardized for all US vehicles and can be obtained through the aforementioned OBD-II protocol. These are only a subset of information available in the vehicle. Body-related information, such as door or turn signal status, is completely missing, as it is not emission-related. Even some kinematic-related signals, such as the steering wheel angle are not included in this protocol. Obtaining those unknown signals can be leveraged to build

978-1-6654-8643-9/22/$31.00 ©2022 IEEE

more powerful third-party apps. Unfortunately, these raw CAN data cannot be interpreted by researchers and third-party app developers without possessing the respective translation tables, called CAN database (DBC) files. The latter are proprietary to the OEM and contain the mapping of the raw CAN data to human-readable data. Recent advances in CAN bus data *translation* can overcome this issue (see Sec. IV-E), but have not yet been leveraged in data collection tools so far. Finally, vehicle-mobility solutions based on smartphone data only are limited due to the lack of versatility and the high noise levels in installed vehicles.

(2) Independent Hardware and Infrastructure. Nearly all commercial third-party solutions that have been introduced above rely on their own custom OBD-II dongle that works in combination with their platform. For instance, UBI companies provide free dongles, but they can only be used with their companion smartphone apps for a limited purpose. Furthermore, Otonomo's pre-defined APIs can only be used by vehicles from two partner OEMs.

(3) Data Accessibility. Key limitations of commercial solutions are aforementioned proprietary hardware and libraries [5], as well as restrictive programming language support that hinders the universal deployment of third-party apps on multiple platforms. Furthermore, tools do not support *online data sharing* (i.e., real-time data upload) for real-time, safety-critical third-party apps. Only CarLog [8] seem to provide limited online data sharing support for developers.

Table I summarizes existing solutions and their capability to address the above three challenges. No existing platform addresses all challenges of vehicular data *collection*, *translation* and *sharing*. To meet this need, we design, implement and evaluate a new tool for **D**ata Coll**e**ction, **Tr**anslati**o**n and Sharing for Rap**i**d Vehicular App Developmen**t** (DETROIT) which is depicted in Fig. 1.

TABLE I: Comparison of DETROIT with Existing Solutions

	Carloop [5]	Otonomo [6]	CarTel [7]	CarLog [8]	DETROIT
Data Sources	OBD-II	CAN	OBD-II	OBD-II	CAN
Independent HW & Infrastructure	No	No	Yes	Yes	Yes
Data Accessibility	Limited	Limited	No	Limited	Yes

DETROIT is the first modular and open-source [1]end-to-end tool comprising all three stakeholders in vehicular data collection and sharing: **End User**, **Developer**, and **Platform Operator**. Its contributions for these stakeholders are:

End User. The frontend of DETROIT consists of an Android app which interfaces the vehicle via an OBD-II dongle. *Data collection* is standardized through the *Vehicle Hardware Abstraction Layer* (VHAL) so any hardware dongle can be interchangeably used to overcome limitation (2), although currently only OpenXC [9] is supported. The End User can immediately interface with available third-party apps by simply enabling them.

Developer. Third-party apps from independent developers can be registered and managed through a developer portal. Developers have access to more sensors due to translated CAN data, and can choose between *offline* or *online* data-access modes. In the former mode, the developer can download the translated data in JSON, MATLAB or Numpy formats for further processing. In the latter mode, translated data is streamed to the developers' backend, where the app is implemented in their preferred programming language. These contributions overcome limitations (1) and (3).

Platform Operator. To bridge End Users with Developers, a backend solution, called OpenIVN, is deployed by the Platform Operator which can be a commercial entity such as an OEM or Tier 1 for production use, as well as an academic research entity for limited deployment within their organization. OpenIVN is responsible for local *data translation* and offering an interface to the respective developers for *data sharing*. Vehicle-agnostic data translation is done by either importing a DBC (e.g., in the OEM case) or automatically generating a (partial) DBC file through CAN bus reverse-engineering tools such as LibreCAN [10]. DETROIT integrates the latter framework into its front- and back-end, and is thus the first tool that supports fully-automated, vehicle-agnostic data pre-processing on high-quality raw CAN data, a significant solution to limitation (1).

We evaluate DETROIT in three ways:

Benchmark. DETROIT is benchmarked from end to end in Sec. V with regards to latency, recording size, energy consumption, CPU utilization, and memory consumption to show its light resource overhead and thus feasibility.

Improved Accuracy over Phone Sensing Alone. In Sec. III-C, we introduce some existing apps from academic researchers that rely solely on phone sensor data (such as IMU) to reconstruct the vehicle's motion data. The accuracy of these apps is thus bounded by noisy and restrictive phone sensor readings. In Sec. VI, we show that by re-implementing two phone data-based apps with vehicular sensor data made available by DETROIT, the apps' accuracy can be improved up to 10.11% and 12.36%, respectively.

Enhanced Developer Experience. Developer feedback in Sec. VI-E show that DETROIT significantly reduces the development effort of the aforementioned two apps with regards to lines of code (LOC) up to 36.9%.

II. Background

A. CAN Primer

Vehicular sensor data is collected from ECUs located within a vehicle. These ECUs are typically interconnected via an on-board communication bus, or in-vehicle network (IVN), with the CAN bus being the most widely-deployed in current vehicles. Fig. 2 depicts the structure of a CAN 2.0A data frame, the most common CAN data-frame type.

Highlighted with non-white color in this figure are the three fields that are essential for the understanding of DETROIT:

[1] https://github.com/detroit-framework

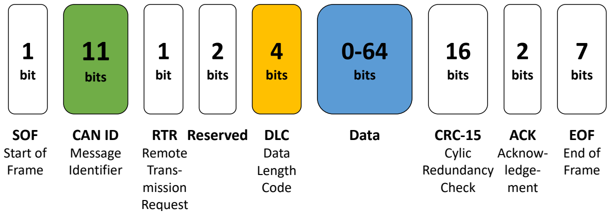


Fig. 2: CAN Data Frame Structure (from [10])

CAN ID: CAN is a multi-master, message-based broadcast bus. CAN is message-oriented, i.e., CAN message frames do not contain any information regarding their source or destination ECUs, but instead each frame carries a unique message identifier (ID) that represents its meaning and priority.

DLC and Data: Data is the payload field of a CAN message containing the actual message data of length of 0–8 bytes depending on the value of the DLC. The data payload field consists of one or more "signals," each representing information such as vehicle speed or engine RPM.

B. DBC Files

All recorded CAN data can only be interpreted using the translation tables for that particular vehicle. These tables can come in different formats, as there is no single standard. The most common format used for this purpose is DBC. They contain a myriad of information. However, to understand this paper, one must be aware of the following minimum information stored in the DBC files:

- Message structure: CAN ID, Name, DLC, Sender;
- Signals located within messages, such as Name, Start Bit, Length, Byte Order, Scale, Offset, Unit, Receiver.

Raw CAN data is not encoded in a human-readable format and does not reflect the actual sensor values. In order to obtain the actual sensor values, raw CAN data must first be decoded. Let r_s, m_s, t_s, and d_s be the raw value, scale, offset, and decoded value of sensor s, respectively, then the actual value can be found from: $d_s = m_s \cdot r_s + t_s$.

C. In-Vehicle Network Architecture

The most widely used IVN architecture is the *central gateway architecture*, with CAN being the most popular bus. The major point of entry into a vehicle for data collection (and diagnostics) is the on-board diagnostics (OBD-II) interface which has been mandated in the US after 1996. Emission-related sensors such as vehicle speed, engine speed, intake temperature, mass airflow, etc., are universally available in all vehicles via the standardized OBD-II protocol [2]. This port can also be used to both read and write raw CAN data. The latter contains more powertrain-related signals, in addition to body-related data. Note that the OBD-II protocol and OBD-II interface are different and should not be confused. Furthermore, the OBD-II protocol is public and does not use DBC files at all.

III. Related Work

A. Collection of Automotive Data

Data-collection service and products for vehicles have been increasingly popular over the last decade, especially thanks to the wide deployment of smartphones. There are several academic projects that have built apps on top of collected vehicular data. Of them, only two generic platforms focus on the data-collection process itself. One of them is CarTel [7] which deployed GPS, an OBD-II reader, and wireless interfaces to transmit the collected data back to the server from 6 vehicles over the course of a year. This platform was used for a variety of vehicular research efforts, but does not offer the flexibility of a developer API for other researchers.

Commercial products range from app-specific OBD-II dongles (e.g., usage-based insurance dongles [3], [4]) to built-in OEM data-collection platforms (e.g., Android Automotive [11]). All of the above app-specific solutions operate on OBD-II data due to its vehicle-agnostic nature. The most prominent OBD-II dongle is ELM327 [12] which works well with the OBD-II protocol, but does not offer enough bandwidth to read raw CAN data without dropping frames. For performance reasons, we will use the premium OpenXC VI [9] in this paper.

B. Automotive Data Translation

In contrast to the work discussed earlier, there are very few related publications on automotive data translation. They all deal with automated ways to reverse-engineer CAN data [13]–[15], although they differ in how *much* of the available CAN data they cover and how *accurate* their algorithm is. As of now, `LibreCAN` [10] offers both the best accuracy and coverage. It is capable of reverse-engineering both kinematic- and body-related data that we are interested in offering to developers and cover more than half of all signals present in the car with a significantly higher accuracy than other tools.

`DETROIT` is independent of the source of DBC. To keep the framework as generic as possible, Platform Operators can obtain their own DBC files (e.g., under contract from an OEM) or use `LibreCAN` to generate a DBC and use it for data translation.

C. Phone Sensing vs. Car Sensing

The issue of having to rely on mobile device sensors instead of vehicular sensors has been pointed out in [16] which compares the context sensing capabilities of vehicular and mobile device sensors. It highlights two fundamental challenges to improve the performance of third-party apps that can be overcome by using vehicular data sources, namely *sensor availability* and *sensor placement and movement*. The latter discusses the impact of phone placement inside the vehicle (e.g., cupholder, pocket or windshield) as well as movement during data collection on the app performance. After evaluating phone vs. car-sensing on four example apps (lane-change detection, pothole detection, road-grade estimation and stop-sign detection), car-sensing is found to yield better performance for some apps, whereas the precision drops for others. They do not discuss what vehicular sensors have been used to replace the phone sensors and thus a direct comparison is not possible. Furthermore, there is no reference that they explicitly used proprietary CAN data.

VSense [17] is a mobility application that uses the phone's gyroscope and can be replaced with the steering wheel angle as vehicular source alternative that is only available via the CAN bus. A turn detection app that uses the phone's microphone to detect turn signals can directly leverage turn signal data from the CAN bus. Another application is BigRoad [18] that uses accelerometer and gyroscope and can be replaced by speed and steering wheel angle. With the exception of vehicle speed, all those vehicular source alternatives to the originally used phone sensors are only available through interpreting raw CAN data and cannot be globally obtained using the OBD-II protocol.

IV. System Design

A. Design Goals

Simplicity. For easy creation and use of vehicular apps, we would like to make the interface to the developer and the driver simple. We provide a web UI for the developer to create a third-party app and an Android UI for the driver to interface their vehicle and enable apps.

Scalability & Standardization. DETROIT makes vehicle-agnostic data collection and thus works on any car that has an OBD-II interface. Data translation is an integral part of DETROIT and enables developers to "see" an unprecedented amount of vehicular data, independent of make, model or year. Both interfaces to the vehicle and developer will be standardized. Implementing a *Vehicle Hardware Abstraction Layer* (VHAL) allows DETROIT to work with any hardware to interface the vehicle. The modules above VHAL will not be affected by the choice of hardware dongle. Furthermore, the developer obtains the data in a translated, well-formed and well-documented format, so the pre-processing code among different apps of the same developer can be reused.

Security & Privacy. Unlike the OBD-II protocol (SAE J1979) which is used in most of the state-of-the-art as discussed in Sec. III, DETROIT does not require write access to the CAN bus at any point during regular operation and only listens passively to the CAN bus broadcast. This has a significant security advantage over all existing solutions since it reduces the risk of accidentally writing messages to the CAN bus, e.g., if the backend of DETROIT is compromised.

Liability. DBCs are OEMs' proprietary translation tables that should not be shared with the *End User* or *Developer*. Since DETROIT makes use of these files for data translation in OpenIVN, we ensure that only the *Platform Operator*, i.e., the OEM itself or another commercial entity, has control of these files. The acquisition of DBCs made by the operator of DETROIT mitigates OEMs' liability concerns. Fully automated and crowd-sourced CAN bus reverse engineering by LibreCAN also occurs on the backend, further mitigating liability concerns. This design primitive also forces DETROIT to upload the entire raw CAN data for processing to the backend. Storing the necessary DBC information for local translation and selective upload on the frontend may create liability problems.

B. Setting up DETROIT

Developer: Registering an App. Developers can register their apps in the *Developer Portal* and verify their identity using the Google Sign-In service with OAuth 2.0. This way, it is possible to ensure that each developer is the sole entity which can access vehicular data intended for their app(s).

Authenticated developers are required to complete a form in the *Developer Portal*, where they must provide a representative *app name* and a *description* of the app's functionality. Furthermore, developers must choose their desired *data access mode*, i.e., offline or online. In the latter case, developers will be required to specify their own backend endpoint which consists of an IPv4 address and a port. For offline mode, this is unnecessary, since the processed data will become available for download. Finally, developers can select which coarse-grained vehicular data permissions they would like their app to access. An abstracted version of the user interface for registering a third-party app is shown in the top left of Fig. 3. An SQLite database stores those metadata about an app registered through the Developer Portal. Developers have *enhanced application flexibility* since they can edit their registered apps at any time and add or remove permissions. Screenshots for this process are provided as (3a)-(3d)[2].

OpenIVN: Configuring the Backend. DETROIT's backend component OpenIVN requires some initial steps to set up for first-time use. OpenIVN is a web application written in Python using the Flask framework. An NGINX server reverse-proxies requests to a Gunicorn server which then interfaces with the Flask application. The latter handles requests to several REST API endpoints which are used by both the *Developer Portal* and *Frontend*. An overview of all API endpoints to interface the frontend is given in Table II.

DBC Repository. On the backend, we will maintain a repository of DBC files that will be pulled for translation depending on the vehicle make, model and year (see Sec. IV-E). As mentioned before, these will be kept exclusively on the backend due to liability concerns and are exposed only to the *Platform Operator*. Translation files on the backend that we refer to as DBC files throughout the paper are in fact a customized version of the original format in the broader sense. Since every OEM can refer to signals differently, it is necessary to standardize the nomenclature in the custom DBC.

End User. The frontend is an Android app and the only UI to the *End User*. It communicates with OpenIVN in order to not only retrieve vehicular information and app details from the developer portal, but also to send back recorded data as well as some meta-information about the vehicle. Upon first-time startup of the frontend app, the *Vehicle Identification Number* (VIN) is queried via a well-formed OBD-II/SAE J1979 Request [19]. The VIN is a unique 17-character identifier that allows to look up make, model and year through

[2] https://www.dropbox.com/sh/3b567sda2qmfvj7/AACiHjgHw_wuhrHMDMdM6F46a?dl=0

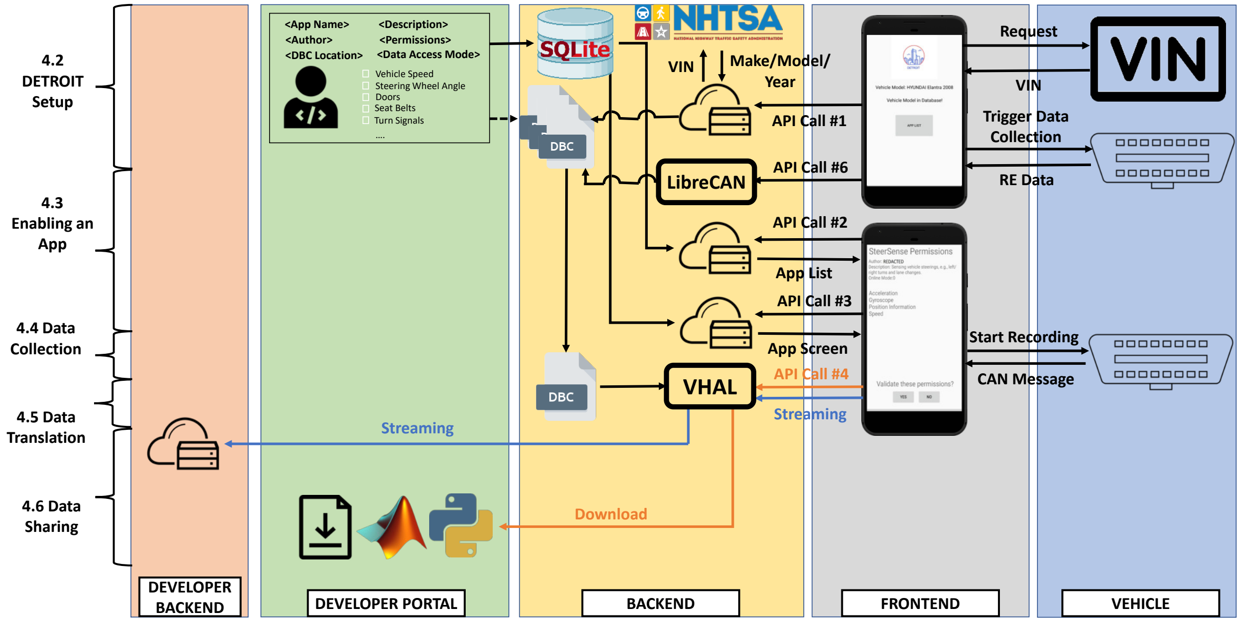


Fig. 3: Dataflow overview of DETROIT . Orange arrows indicate offline and blue arrows online data-access mode.

public online APIs, such as the *Product Information Catalog Vehicle Listing* API [20] provided by the National Highway Traffic Safety Administration (NHTSA). After obtaining the VIN from the vehicle, an HTTP call to Endpoint #1 from Table II is made. OpenIVN then sends a request to the API.

Make, model, and year are important parameters to determine the correct DBC for a specified vehicle. OpenIVN checks if the DBC is present in its repository for the vehicle, else it notifies and prompts the *End User* to automatically reverse-engineer the DBC with LibreCAN [10]. For this purpose, at least 30 minutes of free driving data and roughly 10 minutes of body data collection are sufficient as described in LibreCAN. The compressed data is then uploaded to OpenIVN by an HTTP call to Endpoint #6. Screenshots (1a)-(1d) depict this process. The reverse-engineered DBC will then be added to the DBC repository and made available for other End Users as well as part of crowd-sourcing.

C. *Enabling an App*

After third-party apps have been registered through the *Developer Portal*, OpenIVN has been set up and the vehicle of the *End User* has been identified, the latter is presented with available apps on the frontend. Each time the frontend Android app is launched, it sends an HTTP call to Endpoint #2 to get a refreshed list of available apps. Once an app is selected, the *End User* is presented meta-information about the app, such as author, description, data access mode, as well as the requested coarse-grained permissions. This is done by an HTTP call to Endpoint #3. The user is allowed to accept or reject enabling this app. Finally, note that *enabling* an app is unlike *installing* an app as in traditional mobile operating systems. No binaries are downloaded and installed on the user's frontend device. Furthermore, due to *enhanced application flexibility* as outlined in Sec. IV, no updates/patches have to be installed at any time by the End User. Screenshots for this process are provided as (2a)-(2c).

TABLE II: API Documentation

#	Name	Endpoint	Description
1	Get VIN	/api/v1/vin/ <str:vin>	Sends OBD-II Request to IVN to Obtain VIN
2	Get App List	/api/v1/apps/	Requests App List to Display on Frontend
3	Get App	/api/v1/apps/ <int:app_id>/	Upon App Selection, Meta-data is Requested to Show on Frontend

4	Translate (Offline)	/api/v1/translate/?
		make=<str:Make>&
		model=<str:Model>&
		year=<int:Year>&
		app_id=<int:app_id>
			HTTP POST Sent with Zipped File to Translation Endpoint on Backend

5	Get Dev Message (Online)	/api/v1/messages/ <int:app_id>	Polls Backend to Check for Developer Message

6	Generate DBC	/api/v1/generateDBC/?
		make=<str:Make>&
		model=<str:Model>&
		year=<int:Year>
			HTTP POST Send zipped trace file to reverse-engineer DBC

D. *Data Collection*

Vehicle Interface. We interface the vehicle via its OBD-II port to access one or more CAN buses. Different hardware devices can be used for this purpose, as laid out in Sec. III-A.

VHAL. To meet the design goal of *Standardization*, we implemented a *Vehicle Hardware Abstraction Layer* (VHAL) on the frontend. Since different data-collection devices output a differently formatted recording file, it is crucial that the data transmitted to the backend is standardized. As of now, we only support the OpenXC VI. Other data loggers can be added to the VHAL and the rest of the data flow is abstracted and does not affect the functioning of DETROIT.

Offline Data Collection. In the offline data-access mode, the data is first stored in a formatted file on the phone while data is being recorded from the vehicle. After *End User* terminates the data collection by disabling the app, the file will be compressed to save data before eventual transmission via the phone's network connection. The transmission is triggered by an HTTP POST to Endpoint #4 from Table II. The endpoint includes the vehicle make/model/year as well as the third-party app ID for proper data translation and sharing.

Online Data Collection. Online data-access mode allows vehicular data to be streamed in real time. As mentioned before, some app developers might want to process the data *on-the-fly* to notify the *End User* in real time. An example for this is the Intrusion Detection System (IDS) that will trigger an alert when an anomaly or intrusion in the vehicle has been detected. The formatted data will be streamed via TCP. This is necessary since we cannot assume that the network connection during a trip will always be reliable (e.g., traveling through tunnels) so that data can be re-transmitted. In offline data-access mode, the app ID and vehicle make/model/year were included in the HTTP POST. These two parameters are sent in the first TCP segment before data collection starts.

E. Data Translation

Once the data is transmitted to OpenIVN, it needs to be translated into human-readable sensor data using the aforementioned custom DBC files. The process is similar for offline and online data-access modes: For the selected coarse-grained permissions of each app specified by the developer, the fine-grained signals are mapped to it in the first step. Once we know the location of those signals (CAN ID, Start Bit, End Bit, Scale, Offset), we can filter the specific signal from the raw CAN data with a bitmask and then calculate the absolute value for each signal as explained in Sec. II-B.

Offline and online data-access modes differ as follows. For the former, the compressed file has to be extracted first, and then each message can be translated. For the latter, each TCP segment received from the established TCP socket is decoded into 4-tuples and then added to a buffer to ensure correct packaging of all vehicle data. Complete messages are removed from the buffer, translated and added to a queue. Multi-threading is used to prevent the translation process from becoming the bottleneck, especially given real-time requirements for online third-party apps. OpenIVN ensures that the translated messages are added to the queue in order.

F. Data Sharing

Offline Data Sharing. Recorded vehicular data is made available for developers to download as a JSON file, as well as convenient MAT- and NPY-files for further MATLAB or Python processing in the *Developer Portal*. The JSON file will include the fine-grained permission name, its base timestamp, sampling frequency as well as time-series data vector. The MAT and NPY-files will have a 2D array for each sensor, consisting of the re-calculated timestamp from the base timestamp and sampling frequency.

Online Data Sharing. Based on the developer's indicated choice in the *Developer Portal*, data can also be streamed to an endpoint they indicated. The developer must specify an IP address and a port to which the translated data will be relayed over a separate TCP connection. The developer will receive only the data they have requested. The first entry sent to the developer will contain the app ID and vehicle information. The following packets in that TCP connection will contain a list of messages which are composed of timestamp and data points. The latter consists of the fine-grained permission and the signal value. If multiple messages are ready at the same time, up to 10 of them will be sent together in the same packet with the same timestamp to minimize the total number of TCP packets sent.

DETROIT also offers real-time feedback from the developer. The latter can send a notification to the End User via OpenIVN. For this purpose, the frontend periodically polls the backend for the developer messages by an HTTP call to Endpoint #5. Messages are displayed on the UI of the frontend.

V. Framework Benchmark

A. Experimental Setup

We collected two 1-hour traces from a North American full-size crossover SUV. Trace 1 features both urban and highway driving, while Trace 2 contains only urban driving data. For interfacing the vehicle, we used the OpenXC Bluetooth VI. We did not experiment with multiple vehicles since DETROIT's performance only depends on the number of CAN messages being processed, which is around 1.6M per hour in data acquired from multiple vehicles. All experiments were conducted using Python 3 on a computer running 64-bit Ubuntu 18.04.4 LTS with 128 GB of registered ECC DDR4 RAM and two Intel Xeon E5-2683 V4 CPUs (2.1 GHz with 16 cores/32 threads each). Furthermore, the smartphone used for collecting data for Trace 1 was a Google Pixel XL with 1.6GHz quad-core Qualcomm Snapdragon 821 CPU and 4 GB of RAM running Android 10. For Trace 2, we used a Samsung Galaxy S10+ with 2.84 Ghz octa-core Qualcomm Snapdragon 855 CPU and 8 GB of RAM running Android 10. For all the metrics in the following discussions, we ran each trace in both *offline* and *online* mode with varying trace lengths (benchmarked for Trace 1). Furthermore, the test apps for both access modes include the collection of coarse-grained *Acceleration*,*Gyroscope*, *Position Information*, *Speed* and *Vehicle Turning* permissions, accounting for a total of 15 of 55 fine-grained permissions. We only analyze the following metrics when data collection has been started by the frontend.

B. End-to-End (E2E) Latency

Offline Mode. Table III reports the time required for *offline mode*. The transmission time is dependent on the radio access technology (and thus bandwidth) used between frontend and backend, for which we chose a 100 MBit/s WiFi network.

Offline E2E latency for both traces is similar, standing around half a minute, respectively, for a 1-hour trace. For varying trace lengths (15, 30 and 60 minutes), we can observe a linear pattern. The times for both frontend and backend are reduced proportionally as expected.

TABLE III: Offline E2E Latency (in Seconds)

		Trace 1	**Trace 2**
Frontend	Compress and Upload	26.157	21.582
Backend	Store File and Extract	0.289	0.270
	Translate	5.334	5.237
	Total Backend	6.305	6.220
`DETROIT`	Total Time	32.462	27.802

Online Mode. The critical path in online mode is the delay introduced by `OpenIVN`, since it is responsible for processing incoming messages from the TCP connection by translating and sending it to the developer. In order to support real-time apps, the developer needs to be able to receive the data with minimal latency. Hence, it is desirable to have a reliable and fast network connection between frontend and `OpenIVN`, as well as between between `OpenIVN` and the developer backend. In what follows, we are only interested in the latency induced by `OpenIVN`. As a result, we set the developer backend on the same machine running `OpenIVN`, with the messages being sent to a different port.

We calculate the backend latency as the time difference between the last TCP segment arriving before the socket is closed and the last translated packet being sent out to the developer. For Trace 1, we observed 5 ms and 7 ms latency for Trace 2. These results highlight that `DETROIT` is highly capable of real-time apps since the induced latency is negligible. For traces with varying lengths, the latency stands at 6 ms and 5 ms, respectively, which shows that the processing on `OpenIVN` is independent on the amount of data streamed.

C. Recording Size

Offline Mode. In offline mode, we transmit the compressed data log once the recording has been completed. Compression significantly reduces the file size as can be seen in the top part of Table IV. For both traces, only around 10 MB of data are transmitted. For 30-min and 15-min traces, the compressed sizes stand at 5.13 MB and 2.53 MB, respectively. Once again, we can observe a linear pattern.

Online Mode. In online mode, we calculate the size of TCP segments that are transmitted to the backend. The bottom part of Table IV shows that the transmitted size is much larger than for offline mode, due to the lack of compression, as well as overhead induced by TCP/IP headers. For both traces, the transmitted data from start to end stands at 77.46 MB. This is roughly equivalent to one hour of streaming music through the popular Spotify app in high quality (160 kbps) [21]. For traces of varying length, the transmitted data stands at 38.75 MB for 30 minutes and 20.12 MB for 15 minutes, respectively, which shows a linear pattern as well.

D. Energy Consumption

Energy consumption is an important concern to smartphone users. In the following benchmark, we are interested in the energy overhead of the `DETROIT` app in particular. Measuring the current drain on a non-rooted Android phone without a modified kernel is difficult. Nevertheless, measuring the battery drain is possible over the *Android Developer Bridge* (ADB). Google provides *Battery Historian* that allows to display the per-application battery drain using battery statistics and bugreport recorded via ADB. In order to have a fair comparison between the traces and phones, we also make sure that the phone is fully charged at the beginning of the experiment. The values provided by Battery Historian are merely estimates. A large estimate provided by it are "UNACCOUNTED" or "OVERCOUNTED" which is what happens when Android estimates that apps have used more battery than has actually been used (as measured by the voltage on the actual battery). For both modes, we calculate the worst-case overhead o induced by `DETROIT` as follows, with $b_{DETROIT}$, b_{TOTAL} and $b_{UNACCOUNTED}$ being the battery drain by `DETROIT` , total battery drain and over-counted battery drain by Android, respectively: $o = \frac{b_{DETROIT}}{b_{TOTAL} - b_{DETROIT} - b_{UNACCOUNTED}}$. We evaluate both the energy overhead o as well as battery drain of the frontend app $b_{DETROIT}$ which are summarized in the top two rows of each access mode in Table IV. For each of the four experiments, we made sure that the battery was fully charged upon launching the data recording. As can be seen in the table, the frontend app of `DETROIT` accounts for around 10% of energy overhead in offline mode. The numbers are expectedly higher in online mode (due to heavy use of networking). Nevertheless, the more practical metric is the actual battery drain $b_{DETROIT}$. According to [22], the average American driver spends 51 minutes per day in the vehicle. Both our traces are one hour long and we can thus derive a good comparison. Even if we add the battery drain caused by OpenXC Enabler (ranging 0.57-3.33%) and account all network (WiFi/Cell) drain for `DETROIT` (ranging 1.58-2.22%), we can see that the total battery drain for the necessary components never exceeds 7%. Given that a phone battery shall last one day, this overhead is relatively low.

E. Other Metrics

Memory Consumption. By using the ADB command $adb\ shell\ top\ -m\ 30\ |\ grep\ PACKAGE\_NAME$, the memory of a specific Android app can be logged. For offline traces, the RAM consumption tends to be lower than for data collected in online mode. The RAM consumption peaks at ≈150 MB in online mode whereas it peaks at 90 MB in offline mode. We can verify the same results for smaller traces. Our results show that the memory consumption in both modes is relatively low. Nevertheless, on lower-end Android devices with <512 MB of RAM, using `DETROIT` together with the OpenXC Enabler companion app to interface the vehicle (RAM consumption stands under 70 MB) might be challenging.

TABLE IV: Recording Size, Energy Consumption and CPU Usage

Access Mode	Metric	Trace 1	Trace 2
Offline	Uncompressed Recording Size	58.99 MB	56.97 MB
	Compressed Recording Size	10.46 MB	9.77 MB
	Energy Overhead	9.14%	11.30%
	Battery Drain	1.33%	1.09%
	Avg. CPU Usage	11.17%	4.69%
Online	Uncompressed Recording Size	77.46 MB	74.47 MB
	Energy Overhead	13.04%	29.36%
	Battery Drain	1.82%	3.20%
	Avg. CPU Usage	9.62%	10.87%

CPU Usage. Finally, we summarize the average CPU utilization of the frontend application in the bottom part of Table IV. *Battery Historian* also provides parameters to calculate CPU usage on a per-app basis. Given the user time t_u, system time t_s, total measured time t_t and the number of CPU cores N, the CPU usage c can be calculated as [23]: $c = \frac{t_u + t_s}{t_t \cdot N}$. For Trace 1, the metric stays at around 10% for both access modes. For Trace 2, the CPU usage in online mode is comparable to Trace 1, but significantly lower for offline mode. This might be due to newer and more performant memory structures while writing the data to a file. We can verify the same results for smaller traces which shows that DETROIT adds a low computational overhead.

VI. Evaluation

To study and examine the efficacy of DETROIT in benefiting mobility apps, we first articulate how to adapt the CAN data. Then, by studying the two demonstrative mobility apps, we will show how their performance (e.g., detection accuracy) can be improved with DETROIT over smartphone data only. We collected 30-min and 35-min CAN data from the aforementioned North American full-size crossover SUV (Vehicle A) and an Asian electric two-seater microcar (Vehicle B), respectively.

A. Using CAN Data for Mobility Apps

CAN bus arbitration prioritizes messages with a lower CAN ID, and thus creates varying signal sampling rates. Timestamp traces of different CAN signals (i.e., speed, gyroscope, and steering wheel angle) show a slightly nonlinear pattern which is the natural effect of CAN arbitration. The *overall* sampling rate of aforementioned three signals are 2.81 Hz, 9.38 Hz, and 8.83 Hz, respectively. App developers should thus be aware of CAN arbitration and align timestamps.

B. Demonstrative Apps

Steering Detection. The detection of vehicle steering on mobile devices exploits gyroscope readings to retrieve two-level information from the vehicle's steering maneuvers: the angular change and type of steering maneuvers (i.e., lane-changes vs. left/right turns). As in prior work [16]–[18], the angular change caused by a steering maneuver can be *reconstructed* by fusing gyroscope, accelerometer and GPS data that are available on mobile platforms. For example, one can use the gyroscope data to capture the angular change; to differentiate a lane-change from a turn, the accelerometer and GPS are used to reconstruct the horizontal displacement of the vehicle. In particular, VSense [17] is shown to achieve 93% accuracy (i.e., recall) in detecting all 21 lane-changes.

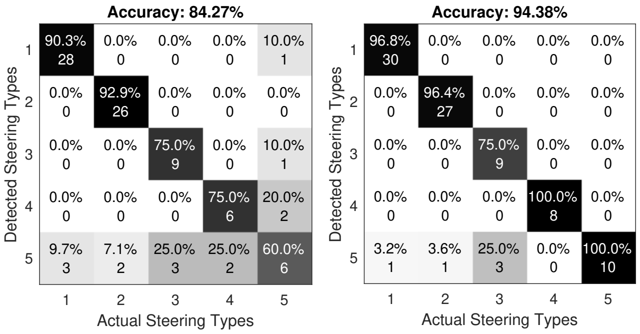
However, smartphone sensors can be noisy due to poor sensing fidelity and phone movements. DETROIT can naturally overcome these problems by making translated vehicle steering data available. To demonstrate the improvement by using DETROIT, we collected the required phone data (i.e., gyroscope, accelerometer, magnetometer, and GPS data) and CAN data (steering wheel and speedometer).

Detection of Turn Signals. Characterizing the driver's turn-signal usage would be an essential element for assessing the driver's attentiveness. For example, drivers' failure to use turn signals alone accounts for more than 2,000,000 accidents annually [24], [25]. Due to the inaccessibility of translated CAN data, prior work infers the turn-signal usage by analyzing context information, such as the periodic clicking sound that is triggered by the turn-signal usage. Specifically, the sound-based approach [17], [26] uses a series of filtering and feature engineering steps, e.g., matched filtering, to reconstruct the clicking sound. However, the performance of this prior work is limited due to environmental noise, including loud music, speaking and the vibration noise induced by bumpy roads. Instead of detecting the clicking sound, turn-signal data can be directly obtained from the CAN bus and translated, thus providing more reliable information for assessing the driver's turn-signal usage. DETROIT can also detect a wrong turn signal — another dangerous driving maneuver that is not detectable with the phone data only.

C. Detection of Steering Maneuvers

To compare DETROIT-collected data with phone-collected data, we use the VSense algorithm in [17] for steering detection data. For this purpose, we reconstructed the vehicle's yaw rate from steering wheel angle (SWA) data by using an estimation formula [18]. To evaluate the performance of steering detection, we used driving data from Vehicle A with accumulated driving of 23.67 km in a suburban area. In total, the driver made 31 left turns, 28 right turns, 12 left lane-changes, and 8 right lane-changes, respectively. Driving data from Vehicle B consists of 10.06 km on a university campus with 28 left turns, 32 right turns, respectively. Due to the single-lane design of campus roads, no lane change maneuvers were performed. During the test, a Google Pixel XL (placed in the cup holder) is used for Vehicle A and a Redmi Note 8 for Vehicle B to collect the IMU and GPS sensor data required for the VSense steering detection. We asked the driver to not move the phone during data collection.

Figs. 4(a) and (b) show confusion matrices for VSense performed with smartphone and CAN data for Vehicle A, where labels 1, 2, 3, 4, and 5 denote left turn, right turn, left lane-change, right lane-change, and non-steering maneuvers, respectively. A non-steering maneuver is not any of the four aforementioned maneuvers, but may also induce a bump-shaped curve in the gyroscope reading. These experimental results show that CAN data is superior to smartphone data in correctly classifying all maneuvers with VSense, i.e., the overall accuracy of steering detection with CAN data is 10.11% higher than with phone data. Note that the data extracted by DETROIT reduces false positives (i.e., detecting non-steering maneuvers as a steering maneuver). The reason for this improvement is that due to the vehicle's motion and loose phone placement, the momentum and/or centrifugal force may cause a slight phone displacement. Hence, the high data fidelity of DETROIT can help developers build high-performance apps. For Vehicle B, the overall accuracy improves by 13.3% when CAN data is used. While the CAN data-based approach provides similar results, the performance of the phone-based method decreased due to driving on a university campus which usually has several speed bumps for speed limit enforcement. Passing through a speed bump induces a bump-shaped curve in the gyroscope reading, which may be misclassified as a steering maneuver by VSense.



(a) Smartphone data.

(b) CAN data.

Fig. 4: Confusion matrices of steering detection.

D. Detection of Turn-Signal Usage

We collected turn-signal data from the CAN bus for both vehicles. To compare to the sound-based turn-signal detection [17], we recorded the sound while driving with an iPhone 7+. The ground truth of turn-signal usage was annotated by the passenger. For a thorough statistical analysis (i.e., both true positive and false positive rates), the driver was instructed to omit turn-signal usage during certain steering maneuvers. For Vehicle A, 43 of 79 steering maneuvers are associated with proper turn-signal uses. For Vehicle B, the blinker is used in 53 of 60 steering maneuvers.

Fig. 5 shows the use of both left and right signals for Vehicle A. SWA data is also provided to illustrate how turn-signal data is associated with steering maneuvers. The zoom-in view clearly shows that the binary turn-signal indicator denotes whether the signal is used or not. Here, the right turn signal is triggered when the steering wheel is turned to the right (negative bump), and vice versa. This information can enable several mobility apps, e.g., inattentive driver detection based on turn-signal usage during steering maneuvers. The app developer can first collect both turn-signal and SWA data with DETROIT. Then, to determine proper turn signal usage during turns and lane-changes, steering maneuvers are detected and checked for a positive indicator during them.

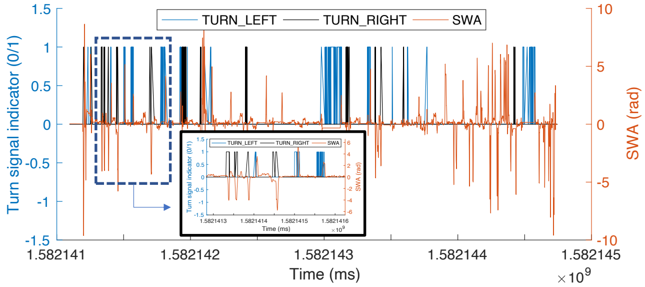


Fig. 5: Data trace of turn-signal usage during steering maneuvers.

TABLE V: Comparison of turn signal detection.

Method	Vehicle	Precision	Recall	Accuracy
Sound Detection	Veh. A	1.0	0.82	0.89
	Veh. B	0.92	0.94	0.88
DETROIT	Both	1.0	1.0	1.0

Finally, we compare the performance of turn-signal detection between phone and CAN data. As shown in Table V, CAN data can (unsurprisingly) achieve 100% in all evaluation scores. For Vehicle A, this approach provides 100% precision (i.e., no false positives) while it suffers from false negatives as indicated in the recall score, because the high noise floor may saturate the filtering capability of the signal processing pipeline. For Vehicle B, recall is higher due to the EV's reduced mechanical noise, but precision suffers from the microcar's lower signal sound volume.

E. Low Implementation Complexity

One of the key benefits of DETROIT is that app developers can directly use high-quality translated CAN data, thus lowering the implementation complexity by (1) shortening the development time via the reduction of the lines of code (LOC) in data pre-processing steps. Specifically, for the steering detection algorithm (i.e., V-Sense), the developers can reduce LOCs up to *10.8%* in Matlab and *12.7%* in Java (Android), respectively. For the turn-signal detection, using DETROIT data can free the developer from implementing the sound signal analysis pipeline — accounting for 36.9% LOC in Java.

VII. Discussion

One important question is *who* will deploy and operate DETROIT to connect all three stakeholders with each other. We envision two possibilities for the Platform Operator:

(1) Commercial Entity. DETROIT can naturally serve as a complete solution for an OEM with DBCs already at hand.

OEMs can embed this platform into their infrastructure to build an OEM-specific app store. The frontend would be implemented on their *In-Vehicle Infotainment* (IVI) instead of a phone. It is also possible that start-ups can deploy DETROIT to bridge drivers with developers for data monetization.

(2) Academic Entity. Researchers can also deploy DETROIT to accelerate their vehicular research. Since it is difficult for researchers to obtain DBC files, crowd-sourced CAN reverse engineering can be leveraged to accelerate their research.

Although DETROIT provides a highly functional end-to-end solution for vehicular data collection, translation and sharing, it still comes with certain limitations which are part of our future work. Mass testing of DETROIT will also help identify more issues. In what follows, we elaborate on limitations and possible extensions:

Multi-App Support. The current implementation allows multiple apps to be enabled simultaneously. For both modes, the data is processed separately on front- and backend. We can further save bandwidth by avoiding repetitive transmission of data over multiple TCP connections for multiple apps.

Privacy. Although DETROIT implements the data minimization privacy goal by supporting a customizable permission model, other privacy design primitives, such as data sanitization are not implemented on the backend. Different permissions/signals have different privacy sensitivities as shown in [27]. The use of sanitization algorithms, such as Differential Privacy, can help overcome the End User's privacy concerns while preserving the third-party app's utility.

VIII. Conclusion

In this paper, we have presented DETROIT , an open-source end-to-end solution for vehicular data collection, translation and sharing. It is the first tool that supports vehicle-agnostic data translation of raw CAN data which is made possible by a fully-automated, crowd-sourced CAN bus reverse engineering tool integration. We benchmarked the performance of DETROIT with several metrics to show its lightweight performance. Finally, mobility app developers used DETROIT to re-implement their apps that were designed with smartphone sensors to highlight both the performance enhancements with CAN data due to high sensor fidelity as well as improved application flexibility and low implementation complexity.

Acknowledgments

The work reported in this paper was supported in part by Ford Motor Company, ONR Grant No. N00014-22-1-2622, and NSFC 6210071207.

References

[1] Tecsynt Solutions, "How to reach the new business niche: Connected car app development approaches," https://medium.com/swlh/how-to-reach-the-new-business-niche-connected-car-app-development-approaches-7e4d3849b4fb, April 2018.

[2] "Obd-ii pids," Feb 2019. [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs

[3] "What is snapshot and how you can save," 2020. [Online]. Available: https://www.progressive.com/auto/discounts/snapshot/

[4] "Drivewise - allstate," 2020. [Online]. Available: https://www.allstate.com/drive-wise/drivewise-device.aspx

[5] "Make apps for your car," 2020. [Online]. Available: https://www.carloop.io/

[6] "Otonomo: Fueling the connected car revolution," http://otonomo.io/, 2017.

[7] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden, "Cartel: a distributed mobile sensor computing system," in *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 125–138.

[8] Y. Jiang, H. Qiu, M. McCartney, W. G. Halfond, F. Bai, D. Grimm, and R. Govindan, "Carlog: A platform for flexible and efficient automotive sensing," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014, pp. 221–235.

[9] "The openxc platform," 2020. [Online]. Available: http://openxcplatform.com/

[10] M. D. Pesé, T. Stacer, C. A. Campos, E. Newberry, D. Chen, and K. G. Shin, "Librecan: Automated can message translator," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: ACM, 2019, pp. 2283–2300.

[11] "Automotive : Android open source project," 2020. [Online]. Available: https://source.android.com/devices/automotive

[12] Elm Electronics, Inc., "Obd." [Online]. Available: https://www.elmelectronics.com/products/ics/obd/

[13] M. Marchetti and D. Stabili, "Read: Reverse engineering of automotive data frames," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 4, pp. 1083–1097, April 2019.

[14] M. E. Verma, R. A. Bridges, and S. C. Hollifield, "Actt: Automotive can tokenization and translation," *arXiv preprint arXiv:1811.07897*, 2018.

[15] M. Markovitz and A. Wool, "Field classification, modeling and anomaly detection in unknown can bus networks," *Vehicular Communications*, vol. 9, pp. 43–52, 2017.

[16] H. Qiu, J. Chen, S. Jain, Y. Jiang, M. McCartney, G. Kar, F. Bai, D. K. Grimm, M. Gruteser, and R. Govindan, "Towards robust vehicular context sensing," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 3, pp. 1909–1922, 2017.

[17] D. Chen, K.-T. Cho, S. Han, Z. Jin, and K. G. Shin, "Invisible sensing of vehicle steering with smartphones," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 1–13.

[18] L. Liu, H. Li, J. Liu, C. Karatas, Y. Wang, M. Gruteser, Y. Chen, and R. P. Martin, "Bigroad: Scaling road data acquisition for dependable self-driving," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 371–384.

[19] "How to request vin," Oct 2016. [Online]. Available: https://community.carloop.io/t/how-to-request-vin/153

[20] "Nhtsa product information catalog and vehicle listing (vpic) - vin decoder," Feb 2019. [Online]. Available: https://catalog.data.gov/dataset/nhtsa-product-information-catalog-and-vehicle-listing-vpic-vehicle-api-json

[21] "How much data does spotify use? - probably less than you think," Nov 2018. [Online]. Available: https://www.androidauthority.com/spotify-data-usage-918265/

[22] W. Kim, V. Anorve, and B. Tefft, "American driving survey, 2014–2017," 2019.

[23] Ina, Matīss, Dāvis, Olga, K. Skutelis, and J. Tipainis, "How we test mobile application performance as a third party," Dec 2018. [Online]. Available: https://www.testdevlab.com/blog/2018/12/how-we-test-mobile-application-performance-as-a-third-party/

[24] "Turn signals - common sense and common courtesy," https://www.transportation.gov/connections/turn-signals-common-sense-common-courtesy, 2017.

[25] "Turn signal neglect is a leading cause of motor vehicle accidents in the u.s." https://www.shanestafford.com/turn-signal-neglect-causes-motor-vehicle-accidents/, 2016.

[26] X. Liu, H. Mei, H. Lu, H. Kuang, and X. Ma, "A vehicle steering recognition system based on low-cost smartphone sensors," *Sensors*, vol. 17, no. 3, p. 633, 2017.

[27] M. D. Pesé and K. G. Shin, "Survey of automotive privacy regulations and privacy-related attacks," SAE Technical Paper, Tech. Rep., 2019.