Using Sensor Redundancy in Vehicles and Smartphones for Driving Security and Safety

by

Arunkumaar Ganesan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2020

Doctoral Committee:

Professor Kang G. Shin, Chair
Professor Alex Halderman
Research Professor Peter Honeyman
Associate Professor Gabor Orosz

Arunkumaar Ganesan

arungan@umich.edu

ORCID iD: 0000-0002-4137-1911

to my wife and my family

ACKNOWLEDGEMENTS

This PhD is not my own accomplishment. It is made possible by all those who supported me, gave me confidence, and helped create a nurturing environment where new ideas can grow and thrive.

First and foremost I am thankful for my advisor. Prof. Kang Shin had the difficult job of watching me repeatedly switch between different research topics before he guided my attention towards the topic which eventually became this dissertation. Prof. Shin generously gives his time and energy to each of his students and collaborators.

I also want to thank my committee members for helping refine my thesis and research.

I am thankful for my lab members. Doing a PhD has many ups and downs. Sharing it with others who are undergoing the same experiences helped me persevere and come out victorious. I am humbled by their intelligence, hard work and creativity. Specifically I would like to thank Kassem Fawaz, Huan Feng, Yu-Chih Tung, Dongyao Chen, and Kyu-Suk Han for joining me on this journey and being the source of many new ideas.

I am grateful for the unconditional support and sometimes stern guidance of my family. Doing a PhD is hard but it would have been much harder if my family wasn't always available providing silent moral support.

Finally, I am incredibly thankful for my wife. Evie has been a permanent source of support and inspiration for all the years we've known each other. Her warm and kind support has helped me through the hardest moments of my degree and encouraged me to keep pushing.

This thesis research has been supported in part by National Science Foundation under Grant CNS-1646130, University of Michigan MCity Program as well as Ford–UM Alliance Program.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

The average American spends around at least one hour driving every day. During that time the driver utilizes various sensors to enhance their commute. Approximately 77% of smartphone users rely on navigation apps daily. Consumer grade OBD dongles that collect vehicle sensor data to monitor safe driving habits are common.

Existing sensing applications pertaining to our drive are often separate from each other and fail to learn from and utilize the information gained by other sensing streams and other drivers. In order to best leverage the widespread use of sensing capabilities, we have to unify and coordinate the different sensing streams in a meaningful way.

This dissertation explores and validates the following thesis: **Sensing the same phenomenon from multiple perspectives can enhance vehicle safety, security and transportation.**

First, it presents findings from an exploratory study on unifying vehicular sensor streams. We explored combining sensory data from within one vehicle through pairwise correlation and across multiple vehicles through normal models built with principal component analysis and cluster analysis. Our findings from this exploratory study motivated the rest of this thesis work on using sensor redundancy for CAN-bus injection detection and driving hazard detection.

Second, we unify the phone sensors with vehicle sensors to detect CAN-bus injection attacks that compromise vehicular sensor values. Specifically, we answer the question: *Are phone sensors accurate enough to detect typical CAN-bus injection attacks found in literature?* Through extensive tests we found that phone sensors are sufficiently accurate to detect many CAN-bus injection attacks.

Third, we construct GPS trajectories from multiple vehicles nearby to find sta-

tionary and mobile driving hazards such as a pothole or a bicyclist on the side of the road. Such a tool will effectively extend the coverage of current navigation assistant applications such as Google Maps which detect and warn drivers about upcoming stationary hazards.

Finally, we present an easy-to-use tool to help developers and researchers quickly build and prototype data-collection apps that naturally exploit sensing redundancy.

Overall, this thesis provides a unified basis for exploiting sensing redundancy existing inside a single vehicle as well as between different vehicles to enhance driving safety and security.

CHAPTER I

Introduction

1.1 Background

Our daily driving commute is heavily augmented with sensor data from our phones and the vehicle's internal sensors. Navigation apps direct our routes, steer us away from traffic, and warn us of upcoming speed traps. The vehicle monitors its own internal state to warn if there is something wrong with the engine or if the fuel level is low. There are many commercially available OBD-dongles which provide statistics such as our driving score [6, 101].

These sensing streams remain largely isolated from each other. We can provide more useful functionality if we unify the different sensing modalities and integrate them with neighboring cars. Successful applications of this kind of information sharing have proven to be very useful in traffic prediction; over 77% of smartphone users regularly use navigation apps [66]. By exploiting this spatio-temporal redundancy of sensing information, we can enhance our daily lives.

In this thesis, we apply sensing redundancy to solve problems in two different domains — CAN-bus injection detection and driving hazard detection. Furthermore, we created a data collection app builder which makes it very easy for researchers and developers to launch data collection campaigns which take advantage of these naturally-existing redundancies. The following sections summarize the background

and state of the art in these domains.

1.2 CAN-bus Injection

The most widely-studied form of vehicular cyber-attack is CAN-bus injection [69, 57, 46]. The CAN-bus is a broadcast-only bus connecting different electronic components within the vehicle. By design, it doesn't have sender authentication and uses a very simple distributed medium-access protocol. This makes the CAN bus vulnerable to injection and spoofing attacks. An attacker would first gain access to the CAN bus through a variety of local or remote methods. Checkoway *et al.* [14] identified many such entry points to the CAN bus, even including a malicious file on a CD that is played through the infotainment system. The malicious file flashes a new firmware to the infotainment ECU which gives the attacker the ability to read and write to the CAN bus. Once the attacker has write access to the CAN bus, they have the ability to cause damage by spoofing falsified sensor readings. For example, in one attack demonstrated in [68], an attacker triggered the Park Assist system while the car is moving by falsifying multiple sensor values over the CAN bus.

1.2.1 CAN-bus Traffic Monitoring

One approach to detecting and defending against CAN-bus injection attacks is to monitor the CAN bus traffic and identify any abnormal patterns. These solutions model the normal behavior using various statistics of CAN-bus packets. For example, Müter *et al.* [72] uses CAN-bus entropy and Cho and Shin [20] use inter-packet arrival time to model normal behavior. In the presence of an attack, the CAN bus would exhibit abnormal behavior as the attack attempts to flood the bus with spoofed packets. Some other solutions propose the use of cryptography in the CAN bus to prevent injection attacks [43]. These solutions are thwarted by the attacker modifying the attacked CAN traffic to mimic the realistic CAN traffic. In fact, attacks such as

the Bootrom attack discussed in [68] change the CAN bus traffic to mimic realistic traffic patterns.

1.2.2 Additional Hardware

Commercial vehicular intrusion detection systems often propose adding new components inside the vehicle for IDS purposes [4, 89, 98]. These solutions require deep integration into the vehicle which often increase the cost for the consumer. This would be undesirable for all parties involved. Furthermore, they fail to exploit the redundancy already present such as other vehicles or the user's smartphone.

1.2.3 Modeling Specific Subsystems

Another class of systems detect CAN-bus intrusion by modeling sub-components of the vehicle — Cho *et al.* [21] modeled the acceleration brake response of a vehicle to detect compromised brake-by-wire systems and Wasicek and Weimerskirch [107] modeled the engine torque response to detect chip tuning attacks — or by modeling certain properties of the vehicle over road segments — Agamennoni *et al.* [2] modeled the normal trajectory of vehicles and Jiang *et al.* [51] predict speed and traffic given road-related properties. These methods all rely on in-vehicle information. If an attacker has the ability to perform CAN-bus injection, they also have the ability to spoof multiple sensor values in agreement with the model thereby evading detection.

We propose a solution for CAN-bus injection detection that overcomes these limitations. Our solution uses information from the near-ubiquitous smartphones and therefore provides an external source of knowledge to verify the sensors broadcast within the vehicle.

1.3 Detecting Stationary and Mobile Driving Hazards

Unseen and surprising hazards on the road lead to fatal accidents. There were 5,977 pedestrian deaths in 2017 in traffic accidents [27]. Early warning of upcoming hazards will bring the driver's attention to the road and help reduce such accidents. A driving hazard can be stationary (e.g., stopped car), slow moving (e.g., pedestrian) or even fast-moving objects (e.g., a reckless drunk driver).

Most existing work focuses on detecting stationary hazards on the road. Commercial applications like Google Maps or Waze notify the driver of an upcoming stopped car or a speed trap. Academic works also use IMU and GPS data to detect stationary landmarks such as potholes [32, 91], speed bumps [3, 52, 7] or unprotected turns [16]. However these methods fail to detect moving hazards such as pedestrians, which has led to thousands of pedestrian deaths in 2017 [27].

Other mobile hazards such as pedestrians or animals on the road do not actively transmit their location therefore it is impossible to directly collect data from these hazards in order to warn future drivers. They must be inferred through sensors. Using a camera or other vision-based sensors is currently the best way to detect these hazards on the road. However, this is limited to vehicles equipped with these sophisticated sensors and has its own challenges such as bad weather conditions occluding the view.

A reckless driver is also a mobile hazard to other well-behaved drivers on the road. Existing work on detecting dangerous drivers use in-vehicle data [45, 67, 117, 104, 116, 107], smartphone data [45, 113, 60, 61, 18], camera data [104, 61, 112, 96, 53] or GPS trajectories from the reckless driver's vehicle [115, 111, 47, 2, 78]. These methods fail to detect the reckless driver if they disable the data collection mechanism. Ideally, we need a method that can detect such driving hazards, whether stationary or mobile, without requiring their explicit participation.

We propose a system that detects and tracks mobile driving hazards using only

the GPS trajectories of nearby vehicles. This takes advantage of the near ubiquitous GPS samples and can track stationary and mobile sensors even if they are not actively broadcasting their location.

1.4 Vehicular Data Collection Platforms

Vehicular research spans a diverse set of areas including driver monitoring [87, 54, 96], road anomaly detection [32, 3, 103], and vehicular security [72, 49, 68]. Due to the lack of a very flexible reconfigurable data collection builder, most of these researchers build their own data collection tools. This is a high barrier of entry for non-technical researchers who would like to enter this field and investigate vehicle-related research questions. Furthermore, since platforms are built for a specific purpose, they often lack the flexibility to take advantage of redundant ways to measure the same information.

Most data collection apps have similar requirements. They must all first access and process low level sensor data from the vehicle or the phone. This requires understanding how to interface with different hardware devices and consolidating all the information in one place. Next they often need to upload the data to a remote server for later processing, and handle user management of the uploaded data. In many cases, the data needs to be communicated between multiple aspects of the application that is deployed on the user's phone. A general data collection platform needs to address these requirements.

We present an overview of the current state of vehicular data collection platforms below.

1.4.1 Specialized Data Collection

Certain use cases require custom-built data collection tools and therefore cannot be automated with a general, configurable tool builder. For instance, the IVBSS

study [42] requires data from modified Honda Accords and Bender *et al.* [8] require integration with the vehicle's LiDAR and other sensors. Other data collection platforms require adding sensors to vehicles. The Safety Pilot Model Deployment [10] outfits cars with a DSRC antenna, an aftermarket safety device, and sometimes with a MobileEye camera [100]. CANOPNR [95] is an OBD-II data logger built using Arduino that can run local processing and offload the data to the cloud. This platform was used to study slippery road conditions [30]. BigRoad [64] uses an easy-to-deploy data collection platform [63] consisting of an IMU sensor attached to steering wheel angle and a smartphone app. These research undertakings require a heavy engineering effort and custom platforms to suit their special needs.

1.4.2 General Data Collection

Other vehicular research efforts can benefit from a general data collection builder tool. For instance, SenseMyCity [84] is a crowd-sourcing mobile platform that collects data from the smartphone and the vehicle through the OBD-II port. This has been used to study city-wide fuel consumption [83] and the mental state of bus drivers [85]. Chen *et al.* [17] built V-Sense, which uses smartphone-based sensing to detect steering maneuvers. Walhstörm *et al.* [103] include many such examples in their survey. These investigations can be expedited by the existence of a simple data collection tool builder that can be configured to meet their specific needs. This would enable non-technical researchers to undertake similar research projects.

1.4.3 Reusable Data Collection Platforms

There are a few notable platforms that have been re-used across multiple investigations. The CarTel hardware data collection platform [48] was customized with additional sensors and used in several follow-up works [31, 71, 32, 97]. However, this platform wasn't designed to be easily extended to additional use cases and must be

manually modified for each investigation. In contrast, we develop a data collection builder that is extensible for future required functionality.

Sensibility Testbed [82] has a web interface through which researchers can submit their data collection tasks. It automatically deploys the task to users who have installed the Sensibility Testbed app. This tool makes it very easy to do data collection, however it does not allow for developers to prototype any real-time custom functionality on top of the data collection platform.

To meet these needs and facilitate vehicular research, we present a tool that can be used to rapidly build data collection apps.

1.5 Thesis Contributions

In this dissertation we unify multiple sensing streams to the advantage of vehicular safety and security. Each chapter of this dissertation contributes towards the following thesis statement:

> **Sensing the same phenomenon from multiple perspectives can enhance vehicle safety, security and transportation.**

1.5.1 Exploratory Analysis of OBD-Sensor Redundancy

I started my exploration into sensor redundancy by investigating the sensors inside the vehicle, often broadcast on the internal communication network. Working with a dataset of 117 drivers, we had access to numerous vehicular sensors sampled at 100 ms for thousands of miles of driving. We use three different techniques to explore relationships between this data.

First, we explored pairwise correlation between sensors *within a vehicle*, and applied that to CAN-bus injection detection. Natural redundancy occurs when the same physical phenomenon causes related effects across multiple sensors. For in-

stance, pressing the accelerator pedal causes the engine to pump faster and increases the speed of the vehicle. Engine RPM and vehicle speed respond in a related fashion to the same cause, the accelerator pedal. We use pairwise correlation to study this effect. We found that there is usually very high variation in pairwise correlation, but if we restrict by context, this variation drops. We show that it can be useful to detect some forms of CAN-bus injection but isn't sufficient or a precise-enough tool in general. This exploration lays the groundwork and gives rise to `CarSec`, presented in a later chapter.

Next we model the normal behavior of sensors *across vehicles in the same road segment*. We start with the hypothesis that as vehicles drive over the same road segment, some of the sensors must react the same way to adapt to the geometric constraints of the road. For example, the steering wheel must roughly match the road curvature. We model this normal behavior on many sensors from the IVBSS dataset using Principal Component Analysis (PCA) and Cluster Analysis (CA). These techniques find patterns in an unsupervised way and detect anomalies. Using these on the vast dataset, we find that in a set of manually-labeled anomalous cases, PCA and CA tend to find them as anomalous as well. Furthermore, PCA and CA are helpful in isolating a small set of cases that seem suspicious or anomalous. Manually inspecting those further helps us identify 12 cases of anomalous cases which were missed in earlier analysis of this dataset. On the other hand, PCA and CA often flag many additional cases as anomalous and cause false positives. These findings motivate the development of `Ubi`, an automatic and more precise way to detect anomalies on the road, presented in a later chapter.

1.5.2 `CarSec`: Using Smartphones as Car Security Assistants

Smartphones have increasingly sophisticated sensing capabilities and are ubiquitous. This gives us a unique opportunity to leverage them for car-security purposes.

In this work, we explore the hypothesis that *smartphones can be used for detecting CAN-bus injection attacks*. We built a system called `CarSec` that uses three smartphone sensors (magnetometer, IMU and GPS) to estimate six vehicular sensors (speed, steering wheel angle, change in odometer, change in fuel level, gear and engine RPM). By using smartphones for this purpose, we augment car-security at *no additional cost* and introduce an external source of information from the car sensors.

In order to answer the hypothesis, we perform two evaluations. Firstly, we implemented vehicle-estimation algorithms and measured their accuracy in a wide range of scenarios. We collected over 900 miles of driving for many different cars and use cases. We measured the accuracy of the vehicle estimation algorithms under such conditions and showed that it is, indeed, quite accurate.

Secondly, we answered the question: *How significant are CAN-bus injection attacks?* We surveyed CAN bus injection papers and characterized the types and magnitude of common attacks. We evaluated the estimation algorithms of `CarSec` against these attacks to show the strengths and limitations of using smartphones to detect CAN-bus injection attacks.

1.5.3 `Ubi`: Using GPS Trajectories to Detect Driving Hazards

Navigation apps that use GPS to provide services are in widespread use. One study found that 77% of users regularly use navigation apps during their commutes [66]. In addition to navigation functionality, these applications report upcoming objects of interest such as a stopped car or a speed trap. Currently these reports are limited to stationary obstacles and cannot report the presence of *mobile* hazards such as a bicyclist on the side of the road or a reckless driver.

Detection of dangerous mobile hazards is difficult to achieve. The most common approach is through direct sensing such as with a camera or a LiDAR. This requires sophisticated computer vision and tracking, which is lacking in most older vehicles.

Other approaches to detect dangerous drivers often require collecting GPS or IMU data from the hazard in question. This becomes impossible if the mobile hazard (e.g. an elusive reckless driver) purposefully disables the data collection or if the mobile hazard naturally doesn't collect any data (e.g. an animal on the road).

In this chapter, we address the problem of detecting mobile driving hazards. The main intuition behind our approach is to model the behavior of vehicles *around* the mobile driving hazard. For example, when approaching a bicyclist, cars tend to swerve out of the way to avoid a collision. Someone driving up to the same location a few minutes later will get an early warning that there is a bicyclist nearby and will receive the estimated distance.

We detect hazards by modeling their location as a three-dimensional graph of location (lat/lng) and time. As drivers sight the mobile hazard, we mark a node in the 3D graph, and then simulate their location forward in time based on the likely mobility of that hazard. As future cars approach this hazard, we use this predicted location to warn other drivers of upcoming hazards.

1.5.4 `CAB`: On-Demand Vehicular Data Collection Builder

Vehicular research often requires building vehicular data collection applications. Researchers tend to implement their own data collection platform for their specific purposes. This creates a large barrier of entry for non-technical researchers and results in wasted effort as even technical researchers sometimes duplicate their efforts. In this work, we built a data collection app builder which allows researchers without programming expertise to quickly build and launch their own data collection platforms.

Our data collection app builder, called `CAB`, defines *information* (data types about users) and *algorithms* (implementations that produce that information). These definitions are language and platform-independent which allows us to define multiple

redundant algorithms that may produce the same information. This allows decentralized development of algorithms by multiple developers as long as they adhere to the agreed-upon interface. We develop `CAB` and use it to build three collection apps to show its expressivity and flexibility.

CHAPTER II

Exploration in Leveraging OBD-Sensor Redundancy Within and Across Vehicles

2.1 Introduction

The vehicular CAN bus exposes numerous sensors about the vehicle state. A sophisticated vehicle may also measure the outside such as the outside temperature or the atmospheric pressure. In particular we explored three different ways of combining the vehicular sensor data in order to gain increased functionality. This study is an exploratory look at redundancy of different sensors within the same vehicle and sensors across multiple vehicles.

2.1.1 IVBSS Dataset

All of our experiments were done on the Integrated Vehicle-Based Safety System (IVBSS) database collected by the University of Michigan Transportation Research Institute (UMTRI) [42]. IVBSS contains diverse data collected from 117 drivers between April 2009 and May 2010 in Southeast Michigan. The purpose of this study was to evaluate the impact of collision avoidance and other safety systems on driver behavior. 16 vehicles (Honda Accord) were distributed to 117 drivers for 6 weeks and extensive data was collected from each vehicle. The vehicles were equipped with a

Sensors on the CAN bus	
Sensor	**Units**
→ Vehicle speed	m/sec
Acceleration pedal	%
→ Steering wheel	deg
Brake	On or Off
→ Throttle and Target Throttle	%
Coolant temperature	deg C
→ Engine speed	rpm
→ Master cylinder pressure	kpa
Intake temperature	deg C
→ Gear	1-7
IMU/GPS sensors	
Sensor	**Units**
GPS speed	m/sec
→ Acceleration in X, Y and Z	m/s^2
→ Yaw, Pitch and Roll	deg
Yaw rate, Pitch rate, and Roll rate	deg/sec

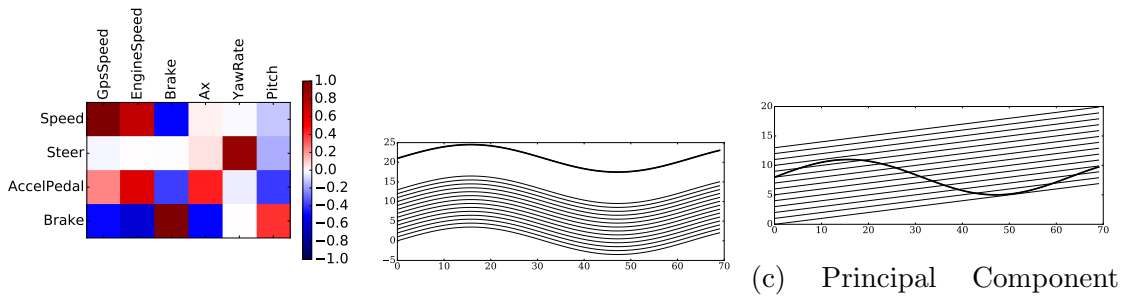
Table 2.1: IVBSS data sources used in our experiments. All sensors were used when calculating the pairwise correlation and the rows marked with a → were used in creating normal models per-drivers.

data acquisition system (DAS) and a custom-designed CAN bus for the purpose of the study. Details of the DAS and data sources are provided in [88].

2.1.2 Exploratory Methods Overview

I explored the relationship between sensors using the three exploratory methods summarized in Fig. 2.1.

Correlation. First I used Pearson's pairwise correlation coefficient to study the relationship between sensors in the same vehicle. The Pearson's correlation coefficient is defined as $cov(X, Y)/(\sigma(X)\sigma(Y))$ where σ is the standard deviation and cov is the covariance of the two. I observed that multiple vehicle sensors seem correlated and are caused by similar physical phenomenon. If we find a strong correlation coefficient, we can use the value of one sensor to detect if the second one is malfunctioning or is compromised.



(a) Pairwise correlation of sensors within a vehicle.

(b) Cluster analysis (CA) of same sensors across vehicles.

(c) Principal Component Analysis (PCA) across vehicles

Figure 2.1: Three different approaches summarized. Each approach was well suited for finding certain kinds of similarities between data. Pairwise correlation found relationships across *different kinds* of sensor data. CA and PCA modeled normal behavior for *same kind* of sensor data *across vehicles*. The highlighted line for CA and PCA are time-series examples which would be marked as anomalous using that approach. Each approach is described in more detail in their respective section below.

Application: CAN-bus injection attacks. The target application for this approach was to detect CAN-bus injection attacks which might compromise one of the sensors. If the correlation significantly differs from the normal case, then we can detect the attack.

Cluster Analysis (CA). Next we explored using cluster analysis to model the normal behavior of sensors *across vehicles*. In this approach, we collected multiple instances of the sensor value for the same road segment. We applied DBSCAN and K-Means to find the optimal cluster which maximizes the silhouette score. In some cases, this approach helped us identify multiple distinct clusters corresponding to different types of behavior (e.g. turning left or turning right).

Principal Component Analysis (PCA). We also used principal component analysis to find normal behavior models across vehicles. Similar to the previous approach, we collected multiple instances of the same sensor from different vehicles over the same road segment. In contrast to CA, PCA was able to find anomalous behaviors even if they are distributed within the same clusters of normal behavior, as shown in the example in Fig. 2.1c.

Application: Anomalous behavior detection. We used CA and PCA normal models to find abnormal driving events in naturalistic data. These methods were helpful in detecting reckless driving behavior or driving maneuvers to avoid deer.

2.2 Related Work

2.2.1 In-vehicle Sensor Relationships

We apply pairwise correlation of in-vehicular sensors towards detecting CAN-bus injection attacks. Koscher *et al.* [57] demonstrated a wide range of vehicular attacks that are enabled once the attacker can write to the CAN bus. Some of the reported attacks are extremely safety critical such as disabling the brakes or killing the engine. There are many defenses to detect CAN-bus injection attacks. Most related to our work are methods which use sensor-sensor relationships to detect these attacks. Our exploration of pairwise correlation for this application is along the spirit of these methods.

Cho *et al.* [21] detect anomalies in the brake sub-system by modeling vehicle dynamics. They use a tire friction model and the current road condition to model the expected braking behavior. We explored a broader set of sensors with pairwise correlation. Liu *et al.* [62] detect anomalies in cyber-physical systems using a spatiotemporal pattern network and a restricted Boltzman machine. They demonstrate how their technique can detect anomalies in smart home monitoring environments, where sensor values tend to be well-behaved and more bounded. In contrast with this domain, vehicular sensors naturally express large variation, many of which may be falsely considered as anomalous. Pajic *et al.* [77] develop an attack-resilient state estimator which functions in the presence of sensor noise. They demonstrate this on an automatic cruise-control for a ground vehicle. We explored using a general pairwise correlation between sensors thereby avoiding the use of fine-tuned models

for each sub system.

2.2.2 Across-vehicle Road-Level Anomalies

Agamennoni *et al.* [2] performed anomaly detection by building an expected trajectory of the vehicle over individual road segments. They measured 'trajectory' by the distance of the vehicle from the center of the lane. Likewise, we build normal models of multiple variables over road segments. In contrast to their work, we go beyond just the trajectory and consider 11 different variables related to vehicle safety.

Jiang & Fei [51] use various road-related properties to predict the traffic and speed of the vehicle. Our work collects data from individual road-segments, however, does not rely on road-specific properties. Moreover, we model normal behavior of many more sensors in addition to speed.

There are also numerous works which detect dangerous or anomalous driving behavior. Zheng *et al.* [117] use cluster analysis to determine which factors lead to near-crash scenarios in naturalistic data. They use K-Means to model the features extracted from the vehicle speed and driver braking behavior. Similar to their work, we apply cluster analysis to natural driving datasets. In contrast to their work, we build situation-agnostic normal models using cluster analysis across road segments.

Many existing approaches detect dangerous driving by extracting features from vehicles [45, 67, 117, 104, 116, 107], smartphone data [113, 60, 61, 45, 18], camera data [112, 96, 53, 104, 61], or GPS trajectories [115, 111, 47, 2, 78]. In contrast to these works, we explored automatically extracting normal models using PCA and CA without restricting to individual sensors or extracting hand-crafted features from the data. Our approach was inherently more exploratory rather than directed towards a specific behavior detection.

2.3 Exploratory Methods

In this section we describe the three approaches in detail.

2.3.1 In-Vehicle: Correlation

We study pairwise correlation in the short-time scale — within trips — and the larger time scale — across trips, drivers, and vehicles. The pairwise correlation coefficient is defined in Eqn. 2.1.

$$corr(X, Y) = \frac{cov(X, Y)}{\sigma(X)\sigma(Y)} \tag{2.1}$$

Normal behavior causes related change within the vehicle. For instance, pressing the accelerator pedal will result in an increase in the speed of the car, cause acceleration in the forward direction, an increase in the engine RPM, and a gear shift for automatic systems. However, in the presence of a fault or an attack, these relationships will no longer hold. If an attacker spoofs the speed of the vehicle, that will no longer correlate with the accelerator pedal behavior, and therefore can be identified as anomalous.

We performed pairwise correlation between all variables from Table 2.1. The variables are divided into two classes — sensors within the vehicle which are broadcast on the CAN bus, and sensors from an external IMU/GPS system. Correlating both external and internal sensors gives us additional redundancy and robustness of the system. In order to successfully fool the system, the attacker has to compromise both internal and external systems, thus increasing the threshold for a successful attack.

Based on the pairwise correlation matrix, we found unexpected and interesting correlations of sensors for individual trips. In many trips the acceleration and brake pedals are positively and negatively correlated with the pitch. This captures when the vehicle slightly dips forward or backward when the driver depresses the acceleration

pedal. We also found that the steering wheel angle is positively correlated with the yaw rate and that the brake is negatively correlated with many variables such as speed, throttle, engine speed and acceleration.

However, these correlations differ across multiple trips. To study this systematically, we explore which variables are consistently correlated for the same driver and how this changes across different drivers and different cars. This is presented in the next section.

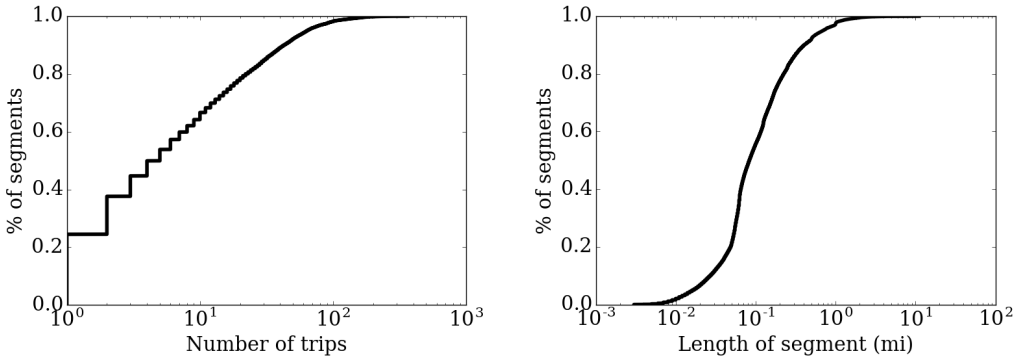
2.3.2 Across-Vehicles: PCA and CA

The guiding principle behind PCA and CA is that vehicles tend to behave similarly at the same location. Sensors such as the speed, steering angle, and IMU sensors are largely determined by the road segment and direction of travel. The exact relationship between road segment and these sensor values is complex. To model this relationship, we rely on data from many vehicles traversing the same road segment.

We operationalize this idea by dividing the map into road segments and aggregating all trips over each road segment. A "road segment" is a stretch of road between two intersections. It is defined this way because vehicles are more likely to enter or exit a road at the intersection points, and therefore, in between intersections we will likely have complete vehicle data. Complete data is useful when applying PCA or CA to the group of trips. For each vehicle driving over the road segment, we sample the sensor values for uniformly distributed points in the road segment.

We divide each road segment into 0.01 mile-long discrete units and average the sensor value within each interval of 0.01 mile. For a segment that is N miles long, this provides a vector of length $L = \lfloor N/0.01 \rfloor$. Some road segments have up to several hundred trips (including the same and different drivers). A "trip" is defined as one drive over the road segment, whereas a "driver" refers to a person with a vehicle.

The distribution of number of trips traversing each road segment and the length



(a) Number of trips over each segment (b) Length of each segment

Figure 2.2: CDFs describing the length of road segments and the number of trips across segments.

of road segment in shown in Figs. 2.2a and 2.2b. Out of 68,507 road segments, 1,206 of them have at least 100 trips traversing them. 50% of the road segments have at least 5 trips or more. 50% of the road segments are at least 0.083 mile long. We focus on road segments which have at least 100 trips.

Both PCA and CA techniques yield different results and taken as a whole we have a more comprehensive view of vehicular anomalies. In what follows, we describe the details of both techniques and compare them in Sec. 2.5.

2.3.3 Principal Component Analysis (PCA)

PCA is commonly used for dimensionality reduction [11] and anomaly detection [109, 102]. Given a set of points in an N-dimensional space, PCA finds a new set of orthogonal vectors, called *Principal Components* (PCs), such that each vector, in order, represents most of the remaining variance. The PCs form a basis of the N-dimensional space and can be found by computing the eigenvectors of the covariance matrix.

Each point in the dataset can be represented as a linear combination of the PCs, as shown in Eq. (2.3) where X is the set of points, V is the top k eigenvectors, Z is the projection of X to V and μ is the mean vector of points in X.

Forward transformation	**Inverse transformation**
$\tilde{X} = X - \mu$	$\tilde{X}V = Z$
$Z = \tilde{X}V$	$\tilde{X}VV^{-1} = ZV^{-1}$
	$\tilde{X}VV^{T} = ZV^{T}$
	$\tilde{X} = ZV^{T}$
	$X = ZV^{T} + \mu.$

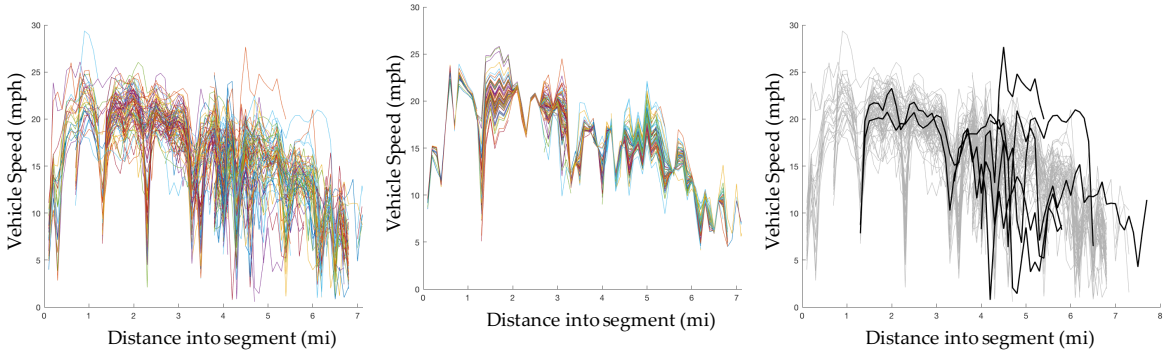
Figure 2.3: PCA forward and inverse transformation where $X \in \mathbb{R}^{n \times p}, V \in \mathbb{R}^{p \times k}$

If we use all eigenvectors (i.e., $V \in \mathbb{R}^{p \times p}$), we can fully reconstruct the original dataset by the inverse transformation shown in Eq. (2.3). If we use a subset of the eigenvectors (i.e., $V \in \mathbb{R}^{p \times k}$ where $k < p$), the reconstructed dataset will be an approximation of the original dataset. If most of the variance of X can be captured using only k principal components, then X can be accurately approximated using only the top k principal components.

In social network datasets, Viswanath *et al.* [102] observed that only 3–5 PCs are required to explain 85% of the variance. Their original dataset captures user behavior in social networks using 181–687 features. PCA-based anomaly detection exploits this phenomenon by representing the original data using a small number of PCs that can capture most of the variance. The key insight is that normally-behaved data can be accurately approximated using the top few PCs whereas anomalous data points are poorly approximated using the same number of PCs.

2.3.3.1 Application to vehicle behavior

We use this property to detect anomalies in vehicular data. First, we aggregate all sensor data over individual road segments and discretize the signal into uniformly-sampled sensor values, spaced 0.01 miles apart. Fig. 2.4a shows the speed values of multiple trips over one road segment. Then, we apply PCA to represent the data using the top N principal components that are needed to explain `Perc_Var` = 95% of the variance, shown in Fig. 2.4b. We vary `Perc_Var` from 90% to 99%, and report



(a) Aggregated speed values over the same road segment.

(b) Speed values represented using the top principal component.

(c) Anomalous speed values highlighted in black.

Figure 2.4: PCA-based anomaly detection technique applied to the IVBSS dataset.

the results later in this chapter.

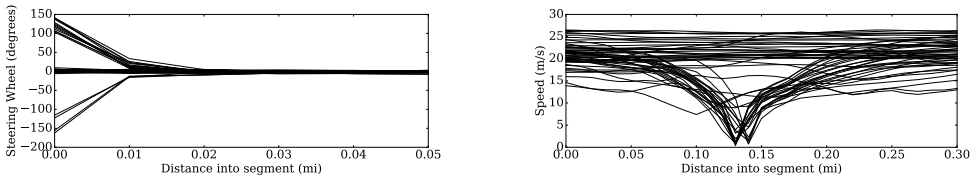
We calculate the Euclidean distance between the original signal and the transformed signal. The anomalous cases will have the greatest distance between the original and the transformed points. Some of these anomalous data are darkened in Fig. 2.4c.

2.3.3.2 Anomaly score

For each trip over a road segment we assigned an *anomaly score* for each sensor of the vehicle. For one vehicle over one road segment, this means we have 11 anomaly scores, one for each sensor in the vehicle. The "anomaly score" is the Euclidean distance of the original signal from the transformed signal using the top PCs. It is normalized by the standard deviation of all distances for each road segment and sensor. The anomaly score gives us a unified way to compare anomalies between PCA and cluster analysis presented in Section 2.3.4.

2.3.4 Cluster Analysis (CA)

This approach is motivated by the examples shown in Fig. 2.5. Fig. 2.5a shows the steering wheel angle of multiple trips over one road segment. One set of trips start



(a) Three clusters found in steering wheel angle (b) Two clusters found in vehicle speed

Figure 2.5: Clusters found in the IVBSS dataset

with a largely positive steering wheel angle and then turn the wheel to the neutral position, while another remains neutral all the way and the third set starts negative becomes neutral. These three cases represent trips when drivers turn into the road segment from the left, right, or continue straight onto the road segment. Similarly, Fig. 2.5b shows two visible clusters in the speed of trips over one road segment.

Similar driving patterns are reflected in clusters in the IVBSS dataset. A trip which fluctuates between multiple clusters may be anomalous. By finding the clusters, we can detect such anomalies. We use unsupervised CA to identify clusters of vehicle sensor data in each road segment. Furthermore, we flag data outside of all clusters as anomalous.

2.3.4.1 Cluster assignment search

CA has numerous variants tailored for different kinds of data; see Xu *et al.* for a thorough survey [109]. We vary many parameters and try two clustering algorithms to find the best possible clustering assignment of the trips over each road segment. Each cluster is evaluated using a relative measure called the *silhouette score* [25]. The silhouette score, defined in Eq. (2.2), compares the average distance of each point to other points within the same cluster and the average distance of each point to neighboring clusters. $b(i)$ is the average distance of i to the nearest cluster. $a(i)$ is the average distance of i with other nodes in the same cluster. The silhouette score ranges from -1 to 1. If it is negative, then the clusters overlap and they are not

Transformation	Values
Low-pass filter	Yes or No
Dimensionality Reduction	99% PCA, 85% PCA, 75% PCA, 2 PC, high dimension
Algorithm	**Parameters**
DBSCAN	$\epsilon \in \{0.01...3\}$, `Min_Samples` $\in \{5, 10, 20\}$
K-Means (K-Means++ [5])	$K \in \{2, 3, 4, 5\}$

Table 2.2: Parameters and algorithms varied during cluster assignment search.

well-defined.

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{2.2}$$

We chose the silhouette score because of the absence of ground truth data. Due to this limitation, we cannot calculate the accuracy of each cluster assignment. Instead, the silhouette score simply tells us if a well-defined cluster assignment exists in the database. This is often used to fine-tune hyper-parameters in existing clustering algorithms such as K-Means [86].

We varied two properties of signal transformations and applied two different clustering algorithms to find the configuration with the highest silhouette score. This search space is succinctly described in Table 2.2.

2.3.4.2 Signal Transformation

We first applied a **low-pass filter** to the vehicular signals. Some of the sensors such as accelerometer values or steering wheel angle exhibit very high frequency noise. We applied a low-pass filter to remove this noise such that two similar trips over a road segment will appear more similar to the clustering algorithm. We also used PCA to **reduce the dimensionality** of the signals before applying clustering algorithms. We represented the signal using N PCs such that they explain 75%, 85% and 99% of the variance. We also tried representing the signal using only 2 PCs and clustered the raw high-dimensional signal.

2.3.4.3 Clustering Algorithms

We explored **two clustering algorithms** – DBSCAN and K-Means (with K-Means++ initialization). DBSCAN is a density-based clustering algorithm which groups nearby points. DBSCAN requires two parameters, ϵ and `Min_Samples`. ϵ is the maximum distance between two points to be considered "nearby" each other. `Min_Samples` is the minimum number of samples required before a set of points is considered a cluster. We varied these values to search for the optimal clustering as shown in Table 2.2. We also varied the number of clusters for K-Means and considered $K = \{2, 3, 4, 5\}$.

We chose these two clustering algorithms because of available computationally effective implementations. In future work, we will explore alternate clustering algorithms such as CLIQUE (for clustering high-dimensional data) and BIRCH (for agglomerative clustering) [38].

2.3.4.4 Anomaly Score

Silhouette score is an indication of clear cluster structure. If the silhouette score is above a threshold (e.g., 0.8), we suppose there are well-defined clusters and extract points that are outside of all clusters. The "anomaly score" is the Euclidean distance from the nearest cluster normalized by the standard deviation of intra-cluster distance. If the silhouette score is below the threshold, we assume there is no clear cluster structure and simply measure the distance of all points from the average point, normalized by the standard deviation. This way we detect very abnormally distributed signals even if there is no clear cluster structure,

2.4 Findings: In-Vehicle

In this section, we share the patterns we found through pairwise correlation within the same vehicle.

2.4.1 Across-Trip Consistency

The correlation of sensors often changes between trips and drivers. We isolate which variables remain highly positively or negatively correlated across all trips of a single driver and explore the changes across multiple drivers. Fig. 2.6 shows the correlation and average change of the correlation across all trips for a single driver. From the full pair of sensors, we identified 14 pairs which have greater than 0.5 or less than -0.5 Pearson's correlation across all trips for at least one driver. The top sensors and the corresponding average correlation matrix are shown in the bottom row of Fig. 2.6 and listed in Table 2.3.

ID	Variable 1	Variable 2	Avg Corr	Avg δ
1	Speed	GPS Speed	1.00	0.00
2	Accel Pedal	Target Throttle	0.99	0.00
3	Throttle	Target Throttle	0.99	0.01
4	Accel Pedal	Throttle	0.98	0.02
5	Y Acc.	Yaw Rate	0.82	0.07
6	Throttle	Engine Speed	0.76	0.09
7	Target Throttle	Engine Speed	0.77	0.06
8	Speed	Engine Speed	0.75	0.09
9	GPS Speed	Engine Speed	0.74	0.10
10	Accel Pedal	Engine Speed	0.74	0.07
11	Steering Angle	Yaw Rate	0.75	0.19
12	GPS Speed	Gear	0.56	0.14
13	Brake	Engine Speed	-0.68	0.12
14	Speed	Gear	0.55	0.14

Table 2.3: Highly correlated pairs, their average correlation and their average change in correlation across trips for a single driver. Results were similar for other drivers and is omitted.

Among the highly correlated variables, we found four pairs to be nearly 100%

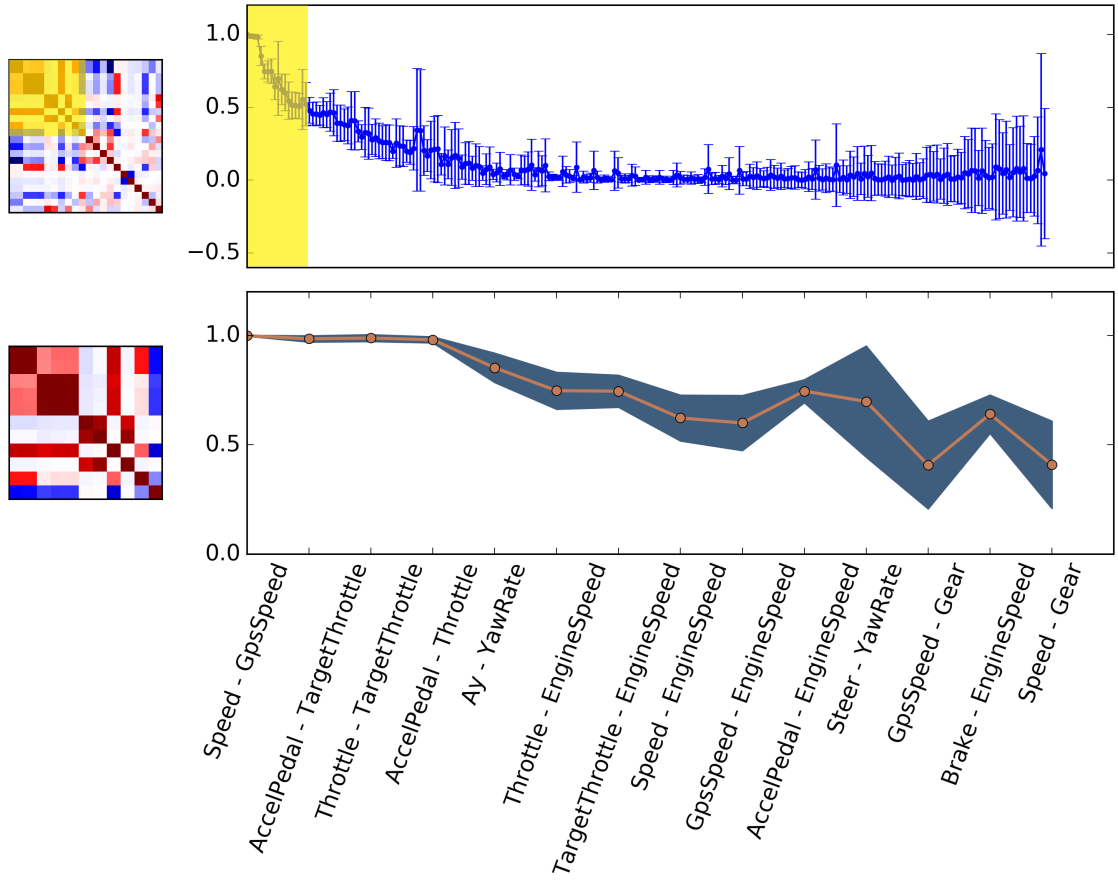


Figure 2.6: The right two figures show the average change of each pair sensors for one of the drivers in our database. The left two figures show the correlation matrix for one of the trips for that driver. The top row of figures correspond to the entire set of pairs. We selected the pairs which correlate more often and tend to have lower variance in the bottom two figures. The subset shown in the bottom two figures are highlighted in yellow in the top two figures. The bounds in the bottom right figure is the average change of that pair's correlation across trips for this driver. The axes labels have been removed due to lack of space when unnecessary. (Best viewed in color)

positively correlated in nearly all the trips. These four were speed × GPS speed, acceleration pedal × target throttle, throttle × target throttle, and acceleration pedal × target throttle. The vehicles in our dataset broadcast the target throttle and current throttle as separate values. Due to their high correlation, we can easily detect if an attacker modifies one of the variables in a sustained attack that lasts throughout the trip.

2.4.2 Vehicle- and Driver-Specific Models

In Section 2.4.1, we explored the long-lasting cross correlation properties for different pairs of sensors. We started to expand this to driver-specific and vehicle-specific models of cross-correlation. Figure 2.7 compares the average correlation of the top pairs of sensors as it varies across drivers and vehicles

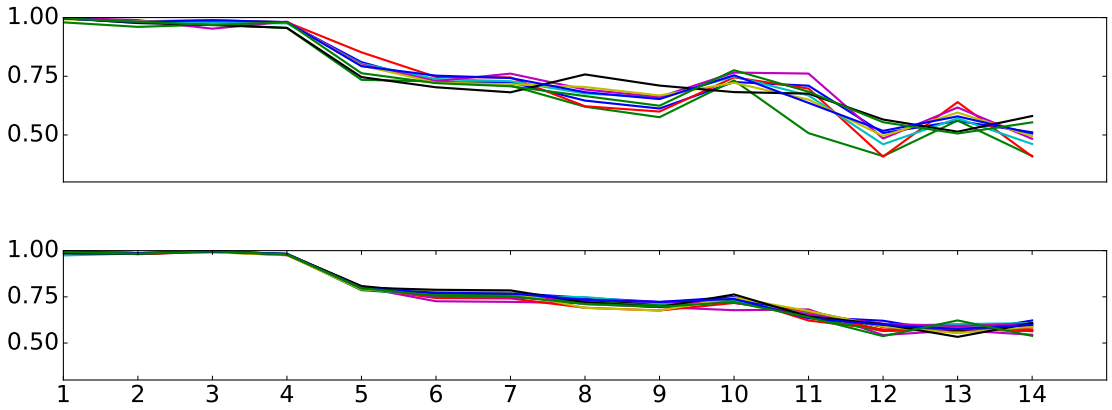


Figure 2.7: The aggregate correlation of all trips across different drivers and different vehicles. The top figure shows the average correlation for all 9 drivers using vehicle 1. The bottom figure shows the average correlation for all 16 vehicles. The ID in the X axis corresponds to the pair of sensors in Table 2.3.

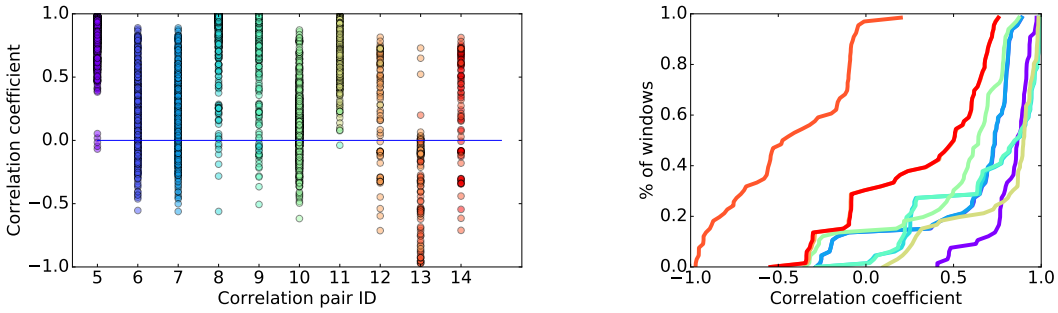
First we computed the pairwise correlation across all data from a single driver, and compared with other drivers. This is shown in the top half of the figure. The first four pairs remain highly correlated for all drivers, however, other pairs vary across drivers. For instance, between two drivers in the same vehicle, the correlation between vehicle speed and engine speed varies by 0.17 (out of 1 being perfectly correlated) and the correlation between steering wheel angle and yaw rate varies by 0.21. We hypothesize that this is caused by driver-specific patterns such as how aggressively the driver turns the steering wheel.

Second, we explored how these correlations vary across different vehicles. Each vehicle has between 7–10 drivers and there are 16 vehicles in total. For each vehicle, we computed the correlation of all pairwise sensor data to get an aggregate correlation

value. This correlation is shown for all 16 vehicles in the bottom of Fig. 2.7. The maximum difference between a pair of vehicles is 0.089 correlation between the brake and engine speed.

This suggests that the changes in correlation is a driver-specific phenomenon and not dependent on the vehicle. Therefore, we must learn this correlation matrix for individual drivers before attempting to use it to detect spoofed sensor attack. We can use the vehicle-specific model as a starting point and iteratively learn the driver-specific model to improve detection.

2.4.3 Within-Trip Consistency



(a) Correlation variation of highly correlated variables within one trip.

(b) CDF of variation for select pairs of variables

Figure 2.8: The distribution of pairwise correlation within a single trip. One trip was divided into multiple 10-second segments. Each pairwise correlation was calculated for each segment and shown above in the scatter plot and the accompanying CDF. The colors in the scatter plot correspond with the colored lines in the CDF.

In order to use the correlation of sensors to detect real-time attacks, we must look at the correlation matrix in a small time window in the current trip. If the current time-window correlation differs from the expected, we can flag it as an attack.

In this section, we investigate the nature of within-trip correlation fluctuations. Fig 2.8a shows the example variation within a trip of a subset of the highly-correlated pairs from Table 2.3. We calculated the correlation within sliding window of 10 seconds for every second of the trip. The entire trip was 2 hours and 32 minutes long.

Fig. 2.8b shows the CDF of the standard deviation of the correlation for each pair of variables for this trip.

On average, we found considerable deviation. Some of the pairs, such as the Throttle and Engine Speed, ranged from ≈-0.5 to 0.9 correlation for certain time windows. We found a similar variability for other trips and other drivers. In the previous section we found that when aggregated across trips and drivers, the correlation remains steady, however, if we look at small time windows within one trip, we found there to be such high variability.

2.4.4 Hypothesis: Contextual Factors

We formed two hypotheses to explain this high variation within one trip: (1) the variation is caused by different contexts of the driver, vehicle and surroundings at each point in time, and (2) within a single context, the variability of the pairwise correlations is much lower. If these hypotheses prove true, then we can use knowledge of the current context to draw bounds for expected behavior and detect anomalous behavior caused by attacks or other factors.

The above hypotheses are motivated by the following observation. Suppose the vehicle goes through a tunnel for a part of the trip. The GPS connectivity will suffer and it will report inaccurate location and speed. In this case, the correlation between the GPS-speed and the vehicle speed will be much lower than what is normally seen. Therefore, the context *"out of GPS range"* leads to a change in the correlation behavior.

Similarly, if the driver is going up-hill versus down-hill, there will be different measurements in the short time-window correlation. If going up-hill, the driver has to apply the acceleration pedal to maintain the speed of the car, but if he or she is going down-hill, then the car is more likely to maintain the speed without much application of the acceleration pedal. For up-hill, the correlation between acceleration

pedal and change in speed of the car will be more correlated. In this example, the context is *"road incline"*.

We developed the tools to ask our database whether such contexts exist and whether our above two hypotheses are correct. We present our results below and report on the amount of standard deviation within each context.

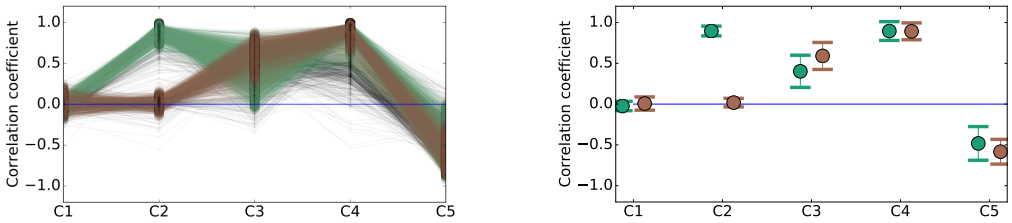
2.4.5 Cluster analysis

We modeled the idea of contexts using clusters of the correlations found across trips. We divided each trip into 10 second windows and calculated the correlations of certain pairs of sensors. We treat this as a point in an N-dimensional space, where N is the number of pairs being considered. We applied DBSCAN [33] to identify clusters in this N-dimensional space. By looking for clusters in this N-dimensional space as opposed to individual pairs, we are able to capture richer relationships between variables.

We chose a subset of the variable pairs from Table 2.1 which are likely to capture different contexts and calculated the correlations across multiple time windows for each driver, shown in Table 2.3. In addition, we explored the variables shown in Table 2.4. We chose these pairs – such as brake and master cylinder pressure – to better capture the context of aggressive or sudden driving. If the brake is applied in a forceful and sudden fashion, the master cylinder pressure will increase rapidly.

Fig 2.9 shows clusters for one of the drivers in our database for the sensor pairs in Table 2.4. We split each trip into 10 second windows and ran DBSCAN with ϵ=0.2 and min samples required to form a cluster = 100.

In Fig. 2.9, we can see the presence of two clusters. The right side of Fig. 2.9 shows the average correlation and standard deviation of members of each cluster. In the green cluster, the brake and master cylinder pressure were much more correlated when compared to the brown cluster. The pitch was also more positively and more



(a) Two clusters emerge in the pairwise correlation. (Best viewed in color)

(b) The average value and the standard deviation for each pair of variables.

Figure 2.9: Each trip for a driver was divided into 10 second windows. Within each 10 second window, we calculated the correlation and used DBSCAN to find clusters. For this driver, DBSCAN identified two clusters.

negatively correlated with brake and accelerator pedal respectively, when compared to the the brown cluster.

The average inter-cluster distance is 0.99 and the intra-cluster distance is 0.41, strongly suggesting the presence of well-defined clusters. The standard deviation of individual pairs within each cluster is quite small. For example, the standard deviation of the brake and master cylinder pressure is 0.06 for the green cluster and 0.05 for the brown cluster. However, when both are considered together, the standard deviation is much larger — 0.41 in total.

Aggressive Driving		
ID	**Variable 1**	**Variable 2**
C1	Accelerator pedal	Pitch rate
C2	Brake	Master cylinder pressure
C3	Brake	Pitch
C4	Steer	Yaw rate
C5	Accelerator pedal	Pitch

Table 2.4: A subset of variables from the IVBSS dataset specifically chosen to capture the context of aggressive driving. If the driver quickly applies the brake or jolts the vehicle when accelerating or turning, we expect to see a high positive or negative correlation among these pairs.

For the variables in Table 2.3 and in Table 2.4, we examined all 117 drivers in our database for the presence of clusters. For each driver, we used a 10 second time window through their trips and generated correlation signatures. For all experiments,

we empirically set the DBSCAN parameters to ϵ=0.1 and minimum points for a cluster=50. We fed this into the clustering algorithm and measured how many clusters are found for each driver. As shown in Fig. 2.10, clustering with Table 2.4 predominantly yields two clusters (59 drivers) and for all but one driver, it finds 2 or more clusters. Clustering with Table 2.3 yields a wider spread of clusters. For 22 drivers, it only identified 1 cluster, and identified at least 2 for the remaining.

The different number of clusters for different drivers can be explained by the types of data encountered by that driver. For example, if a driver lives near a tunnel and often drives through the tunnel, then a new cluster will form when they leave GPS range. Deeper manual inspection is part of our future work.

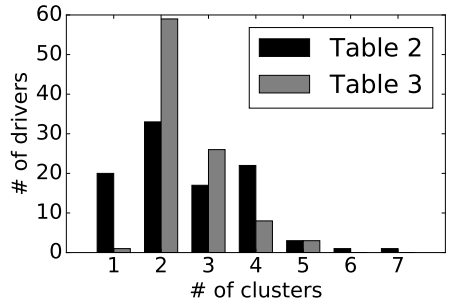


Figure 2.10: Histogram of how many clusters we found for each of the contexts specified in Table 2.4. For each driver, we collected all 10 second time windows for their trips and ran DBSCAN on the final aggregate plot. We used epsilon between clusters = 0.3 and minimum samples within each cluster = 50

Fig. 2.11 shows what percent of the time windows within a single trip are fall into one of the clusters or are marked as unclustered. Even if we find multiple clusters for a driver, it is possible that a subset of the 10-second time windows for that driver are actually unclustered by the DBSCAN algorithm. Fig 2.11 shows that 50% of the trips are clustered 51.4% of the time for Table 2.3 and 50% of the trips are clustered 62.4% of the time for Table 2.4.

Variables in Table 2.4 more consistently have two cluster and more of their trips fall under one of these clusters compared to Table 2.3. This highlights the impor-

tance of choosing the right variables when searching for clusters. The proper choice of variables is part of our future work. This is a challenging problem because we cannot exhaustively search through all subsets of variables (powerset of the variables is a combinatorial explosion) and must resort to heuristics or other simplifying transformations to reduce the search space.

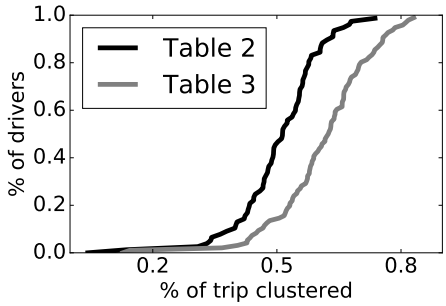


Figure 2.11: The percent of time windows which fall under a cluster across all drivers for each context.

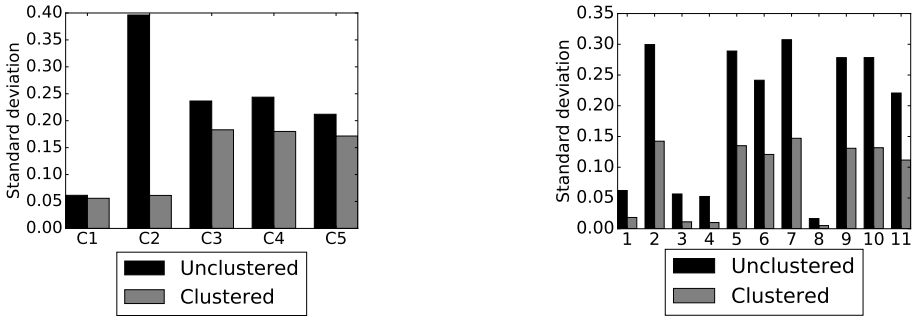
From these results we conclude that (1) clusters exist in the distribution of correlation data and that (2) majority of the time-window correlations fall inside these clusters. In future work, we will form the connection between clusters and contexts. For the purposes our analysis, if we can detect the cluster which belongs to a particular time in the drive (based on contextual clues such as "*GPS is out of range*"), then we can more tightly bound the expected pairwise correlation values.

2.4.6 Variation within each cluster

In the previous section, we established the presence of clusters and that a large portion of time-windows within a trip falls in one of these clusters. In this section, investigate the second hypothesis formed above – *the variability within a cluster remains small compared to across clusters.* If the variability is low, we can form a tighter bound of expected behavior and detect an attack more easily.

Fig. 2.12a and Fig. 2.12b show the change in standard deviation when clustering

the trip data for variables in Table 2.3 and Table 2.4 respectively. The figures show the standard deviation of the unclustered trips and the average standard deviation of all the clusters for both variable sets. For Table 2.4, clusters reduce the standard deviation to 15.5% of the unclustered standard deviation in the best case, and 91% in the worst case. For Table 2.3, they reduce the standard deviation to 19.8% in the best case and 50.6% in the worst case.



(a) Variables from Table 2.4 (b) Variables from Table 2.3

Figure 2.12: The average standard deviation for unclustered and clustered trips for each set of variables. We averaged the standard deviation of the clustered and unclustered across all drivers in the IVBSS dataset. In many cases, we found that clustering significantly reduces the standard deviation of the pairwise correlation, therefore making it a promising technique for attack detection.

2.4.7 Detecting CAN-bus Injection Attacks

In this section, we leverage our understanding of clusters and pairwise correlations to detect an anomaly which may be caused by a malicious attack or a system fault. For each time window, the context is first determined and the cluster describing that context is identified. The exact method of identifying the current context is outside the scope of this work. Then, we compared the current pairwise cross-correlation with the expected values for that cluster. For each pair, we calculated the deviation from the mean correlation value for that cluster and reports it in terms of number of standard deviations from the mean.

To test this idea, we spoofed the speed of the vehicle by modifying collected vehicle

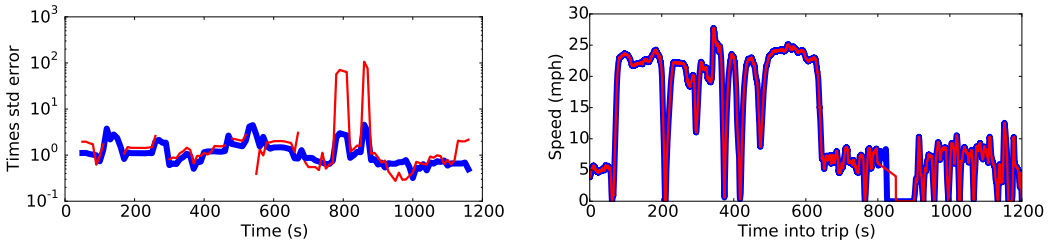
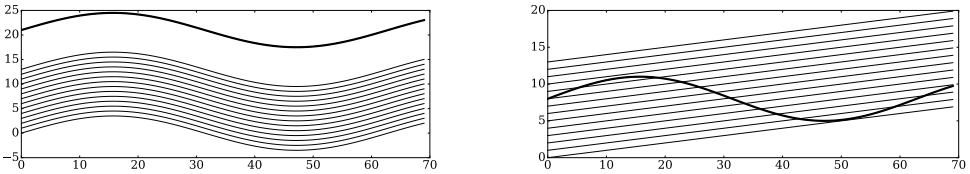


Figure 2.13: The bottom figure shows the attack on the speed sensor of the vehicle. From 800-850 seconds, the vehicle speed is spoofed to appear as though it is slowing down to 4 mph. Then it returns back to normal after a few minutes. The attacked signal is in red and the original trip is in blue. The top figure shows the normalized error (measured as a multiple of the standard deviation) with and without clusters, shown in red and blue respectively. The Y axis of the top figure is drawn in log scale to highlight the difference between unclustered and clustered cases.

traces. In our attack, the attacker injects fake speed values into the CAN bus for 50 seconds from 800-850. He brings the speed down from the current speed to 4 mph in that time frame. Then he stops the attack and the vehicle resumes to broadcast the correct value. Our attack and our detection results are shown in Fig. 2.13.

When we consider the context and cluster, we notice a considerable spike immediately at the attacked time. The error rises to 106.6 times the standard deviation for that cluster. However, when we fail to consider the cluster, the error only rises to 4.59 times the standard deviation, which is below the cut off point to be considered an attack.

This result shows the possibility of using pairwise correlation to detect CAN-bus injection, however this is a very noisy way to perform this analysis. This requires finding the right set of sensors which may form well-defined clusters, and even after doing so, the pairwise correlation sometimes rises to 3.05× and 4.48× the baseline for *non-injected* data. We concluded from this result that pairwise correlation is not well-suited for CAN-bus injection and explored alternate methods in follow up work, also included in this thesis.



(a) Cluster-based anomaly (b) PCA-based anomaly

Figure 2.14: Illustrative examples highlighting the strengths and weaknesses of each technique.

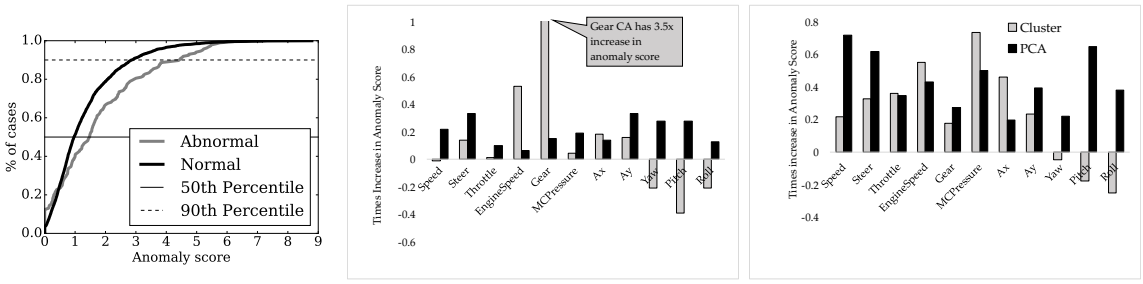
2.5 Findings: Across-Vehicles

We evaluate the accuracy of the anomalies detected through PCA and CA by comparing it with manually annotated cases of abnormal driving. As part of the IVBSS project [42], several researchers manually reviewed the video footage of drivers and vehicles to explain the root cause of forward collision warnings (FCW) and lane departure warnings (LDW). They identified 139 instances of abnormal behavior which triggered FCW or LDW. These include deer or other animals jumping in front of the car, texting while driving, excessive speeding or skidding out of control. We used PCA and CA to calculate the anomaly score of these instances compared to normal behavior. We first present the overall accuracy of PCA/CA and then delve deeper into the evaluation of cluster and PCA-based anomaly detection techniques.

2.5.1 Observations

At a high level, CA finds sets of similar behavior over each road segment. Anomalies detected using CA are outside of all sets of common behavior. PCA finds the common shape of the data signals. Therefore, anomalies detected using PCA are cases where the driver's behavior seems abnormal, even if it is within the bounds of other normally-distributed trips.

Figure 2.14 illustrates the different strengths and weaknesses of PCA and CA. Figure 2.14b shows an example where CA would flag the anomaly but PCA would



(a) CDF of Engine Speed anomaly scores of abnormal vs. normal driving cases.

(b) Average percent increase in anomaly score for abnormal versus normal cases.

(c) 90-th percentile increase in anomaly score for abnormal versus normal cases.

Figure 2.15: Analysis of anomalous and normal data using PCA and CA

miss it. The abnormal time series signal is clearly abnormal and much higher than the others in the set. However, since they all share the same shape, the "abnormal" case can be well represented using linear combinations of the top principal component. Because of this, PCA-based techniques would fail to flag this anomaly. Figure 2.14a shows an example of an anomaly that is better detected using PCA but would be missed by CA. The abnormal time series signal is well within the bounds of the others in the same set. However, due to their abnormal shape, PCA would be unable to reconstruct this using the top principal component, therefore correctly identifying the anomaly.

2.5.2 Detecting Abnormal Cases

We created two sets of data for abnormal and normal driving. Using the manually-annotated cases described above, we collected 139 instances of "abnormal driving". These cases triggered the forward collision warning, lane departure warning, or caused sudden changes in brake pressure. Along the same road segment, we collected the remaining trips and marked them as "normal driving". Note that this is an imperfect labeling of the data since the remaining trips might also contain some abnormal cases that was missed by FCW or LDW. For instance, swerving at night is abnormal behavior but is unlikely to set off the lane departure warning if the lane is not visible.

As shown in Fig. 2.15, abnormal cases resulted in a much higher anomaly score for most sensors than normal cases. We measured the distribution of anomaly score for both abnormal and normal cases, for both CA and PCA. The example distribution is shown in Fig. 2.15a. This example is the distribution of anomaly scores for the Engine Speed calculated using our cluster-based technique. On average, the anomaly score is 52.7% higher for abnormal cases than normal cases. At the 90-th percentile of cases, the anomaly score is 54.9% higher.

At the 50-th percentile, there is a 34.6% and 19.9% increase in anomaly score across all sensors for CA and PCA, respectively. This is shown in Figure 2.15b. Note that the Y-axis is capped at 1. Cluster-based techniques found a 357.9% increase in anomaly scores for the Gear sensor. Moreover, we found an increase of 23.4% and 42.8% for cluster-based and PCA-based techniques for the 90-th percentile cases.

Cluster-based techniques are better suited at finding anomalies in the Engine Speed, master cylinder pressure (MCP) and X-acceleration from the accelerometer. During anomalous cases, these sensors tend to deviate largely from other values. For instance, when the driver suddenly brakes, the MCP spikes up much higher than other drivers on the same road segment. Therefore, these cases are easily detected by clustering similar dense behavior together and detecting the anomaly.

On the other hand, PCA-based techniques are better suited for Vehicle Speed, Steering wheel angle, Yaw, Pitch, and Roll. Even during anomalous driving, these sensors tend to remain within bounds of what is considered normal behavior, but the pattern of the sensor will differ from traditional cases. For instance, when skidding, the driver might have trouble stopping the car and make a sudden stop. This is within the bounds of normal driving as a vehicle can come to a stop for normal reasons. However, the pattern of suddenly stopping is unusual and will be flagged using PCA-based techniques.

2.5.3 Novel Anomalous Discoveries

We apply PCA and CA to identify novel anomalous cases in the IVBSS dataset; we identified 12 additional anomalous cases using these techniques.

Using CA, we discovered and manually verified 8 additional cases of anomalies — 2 abrupt and dangerous lane changes, 2 single lane vehicle overtake maneuver, 2 sudden deviation to avoid hitting another vehicle, and 2 aggressive speeding and reckless driving. For one of the reckless driving instances, 4 out of 10 sensors were flagged as anomalous using CA (Throttle, Engine speed, Y-acceleration, and Steering angle). These are shown in Figures 2.16a, 2.16b, 2.16c, and 2.16d. In this instance, the driver merges onto the highway and speeds through the ramp. Once he merges onto the highway, he suddenly turns the steering wheel to enter the adjacent lane. At times, he speeds beyond 100 mph. This case was not previously manually identified because it did not trigger a lane departure warning or forward collision warning.

Using PCA, we discovered 6 cases of anomalous driving — 2 dangerous lane changing and swerving, 1 speeding, 1 sudden braking due to texting, and 2 sudden deviation to avoid hitting another vehicle. In one of the cases, we found that 5 out of 10 sensors were flagged as anomalous (Engine speed, pitch, Gear, Throttle, and Vehicle speed). This instance is shown in Figures 2.16e, 2.16f, 2.16g, 2.16h, and 2.16i. The driver changed two lanes in a busy highway driving very close to other vehicles. This behavior was unnatural for this part of the road and was flagged as anomalous using PCA. The sensor values were close to the normal bounds therefore CA did not assign a large anomaly score.

Two cases of anomalous driving were flagged by both PCA and CA. In both of these instances, the drivers quickly swerve out of the way to avoid hitting another vehicle. For the first instance, we found a CA-based anomaly in the Throttle position and a PCA-based anomaly in the Engine Speed. In the second instance, we found a CA and PCA-based anomaly in the MC Pressure. These are all shown in Fig-

ures 2.16j, 2.16k and 2.16l. This result demonstrates that PCA and CA may find the same trips and sensor values as anomalous. However, we discovered the remaining 10 cases using either CA or PCA independently; these two techniques can be used to complement each other in discovering anomalous driving behavior.

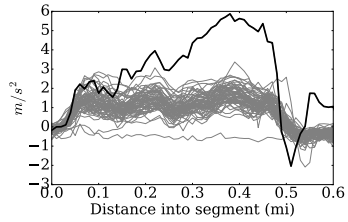
2.6 Conclusion

In our exploratory analysis of in-vehicular sensors, we found that pairwise correlation, principal component analysis and cluster analysis yielded interesting patterns and models.

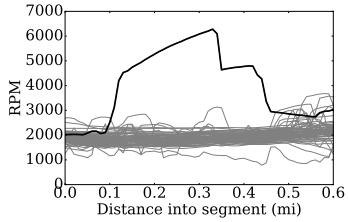
In-Vehicle Pairwise Correlation In the macroscopic scale, there is low variability of pairwise correlations across trips, drivers and vehicles. However, there is considerable variability of pairwise correlation within time windows of a single trip. We studied the cause of this variability and identified the presence of clusters which correspond to contexts of the driver or vehicle. Within a cluster, there is lower variability than compared to across clusters.

Overall we concluded that pairwise correlation may be too coarse-grained of a technique to be useful for attack detection. Even after reducing the variability through cluster analysis, the CAN-bus injection detection is very noisy and often incorrect. This exploratory analysis led us to leveraging the smartphone for CAN-bus injection, presented in Chapter III.

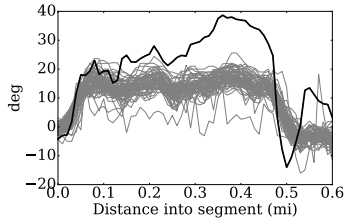
Across-Vehicle PCA and CA We used two different techniques to identify different kinds of anomalies — Principal Component Analysis (PCA) and Cluster Analysis (CA). We applied these to the Integrated Vehicle-Based Safety System (IVBSS) dataset consisting of 117 drivers and over 200,000 miles of driving behavior. On a manually labeled subset of the data, CA and PCA assigned on average a $\approx 30\%$ higher anomaly score to abnormal cases. However, although the anomaly score was higher on average for these anomolous cases, it was also lower during a subset of the



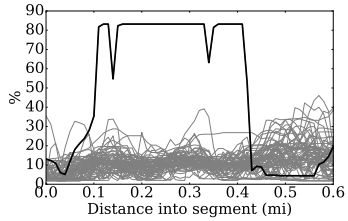
(a) Y Acceleration
CA= 7.02, PCA = 5.27



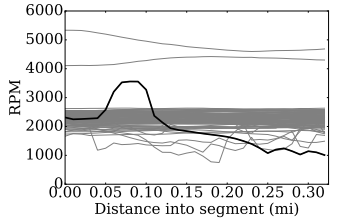
(b) Engine Speed
CA = 7.75, PCA = 0



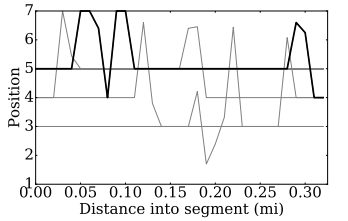
(c) Steering Wheel Angle
CA = 6.92, PCA = 0



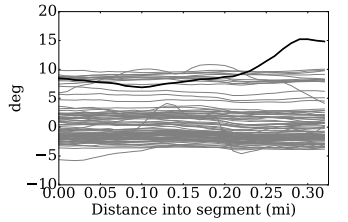
(d) Throttle
CA = 8.34, PCA = 0



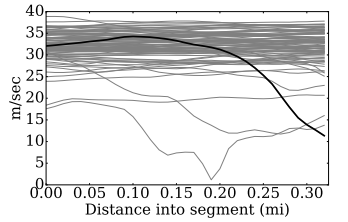
(e) Engine Speed
CA = 0, PCA = 7.18



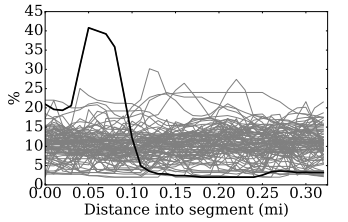
(f) Gear
CA = 0, PCA = 5.11



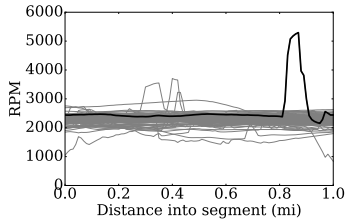
(g) Pitch
CA = 0, PCA = 5.31



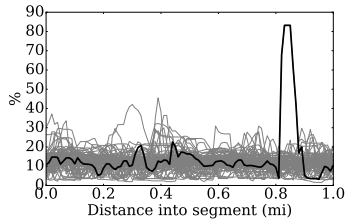
(h) Vehicle Speed
CA = 0, PCA = 6.21



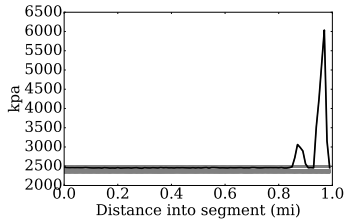
(i) Throttle
CA = 6.28, PCA = 5.7



(j) Engine Speed
CA = 0, PCA = 6.13



(k) Throttle
CA = 7.43, PCA = 0



(l) Master Cylinder Pressure
CA = 9.19, PCA = 5.87

Figure 2.16: Example anomalies for each road segment and each technique. The anomalous trip is highlighted in bold. The sensor and the anomaly scores are listed under each sub-figure.

data. This leads to false positives through this technique.

Furthermore, we used this approach identify 12 new abnormal cases which were previously missed in manual analysis. We found that PCA and CA are best suited for narrowing down the driving data which seems anomalous. Using these approaches, we found a short list of ≈ 100 anomalous cases flagged by our technique and identified the 12 cases presented here. Without our tool, it may be impossible to sift through the large amounts of data to find these unique cases.

Due to the limitations of using PCA and CA for anomaly detection, this work led us to focus on detecting a more specific kind of anomaly caused by mobile and stationary hazards on the road. We narrowed our focus to road-based anomalies specifically in *GPS trajectories* of nearby vehicles in order to find mobile hazards. That work is explained in Chapter IV.

CHAPTER III

CarSec: Using Smartphones as Car Security Assistants

3.1 Introduction

Cyber attacks on cars are now a very real and serious threat. In recent years, researchers have demonstrated numerous vulnerabilities which allow an attacker to take control of a vehicle and put drivers and passengers at peril [14, 37, 69, 68]. Most of these attacks involve injecting *falsified sensor data* into the Controller Area Network (CAN), the *de facto* communication network within a vehicle, and hence the first and foremost step in their defense is to detect these falsified sensor values.

While there are many proposals of increased security in future vehicles, we need to enhance security and safety in cars on the roads today. In particular, we need a system which can be used immediately by passengers to enhance their sense of confidence in the security and safety of their vehicle.

State-of-the-art. Existing security assistant proposals fail to meet this pressing need of increased security in today's cars. They usually detect these in-vehicle attacks by modeling packet-arrival characteristics within the vehicle [20, 73] or the relationships between in-vehicle sensors [21, 39]. If the vehicle is compromised, both classes of security assistants suffer from untrustworthy data since they both rely on

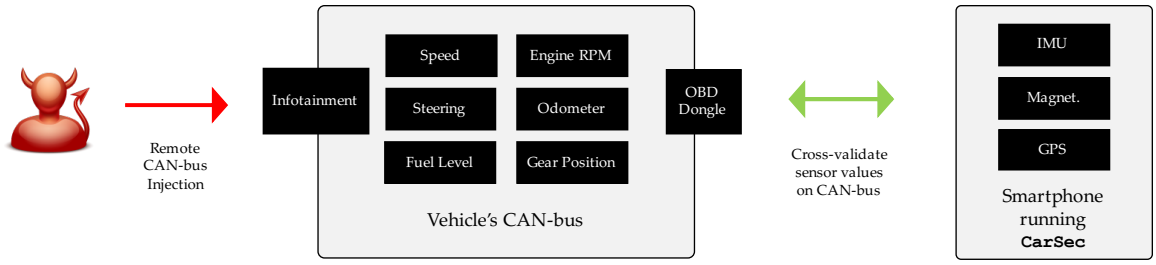


Figure 3.1: Three smartphone sensors are used to infer six different vehicular properties. `CarSec` compares these inferred values with those reported by the vehicle to detect *sensor-falsification attacks*.

in-vehicle sensor data. There has also been a strong commercial push for vehicular intrusion detection systems, but they all require deep integration with the vehicle hardware, making their integration with existing vehicles prohibitively expensive [4, 89, 98].

Proposed Solution. We propose a new security assistant called `CarSec` to fill this need. The first step of many of the vehicular attacks reported in literature involves injecting *falsified sensor data* into the CAN bus. `CarSec` independently acquires these sensor values using the *smartphone's sensors* and cross-validates them with what is seen in the CAN bus. By doing so, `CarSec` adds a second layer of assurance in the vehicle's sensor values on CAN, and can quickly identify when *sensor falsification* is taking place. `CarSec` can be used in tandem with other security measures to detect and defend against the emerging threat of vehicular cyber attacks.

Smartphones have proven successful in assisting with other vehicular tasks such as dangerous driving detection [45, 67, 60, 113, 29], road monitoring [32, 47, 92, 118, 40], and trajectory mining [23, 106, 12, 65]. We expand the use of smartphones to enhance *vehicular security* by detecting sensor-falsification attacks. To the best of our knowledge, this is the first application of smartphones for vehicular security assistance.

One of the main advantages of `CarSec` is its ease of deployment at no additional cost. A driver simply installs the `CarSec` app from the app market, and pairs it

with a Bluetooth OBD dongle which reads values from the vehicle's CAN bus. The *read-only* OBD dongle simply relays the information it sees in the CAN bus to the phone, thereby not contributing to the attack surface. `CarSec` runs automatically in the background to independently acquire vehicle sensors on the CAN bus via a dongle and cross-validate them. Since almost everyone carries a smartphone these days, we can use both driver's and passenger's phones to increase the available sensor redundancy at no additional cost.

By implementing `CarSec` on a smartphone, we provide this additional security to drivers today. The smartphone serves as an external source of knowledge about vehicular dynamics, thereby overcoming the above-mentioned limitation of sole-reliance on in-vehicle sensor data. Furthermore, `CarSec` capitalizes on the existing hardware and computing power of smartphones. This can be used in addition to commercial solutions, if any, which provide enhanced protection but require deeper and more expensive integration with the vehicle.

Key Challenges and Solutions. In order to use smartphones to detect vehicular sensor-falsification attacks, we have to estimate vehicular values and compare them with the sensor values on the CAN bus.

Challenges in Evaluating Falsification Attacks There exists a dearth of evaluation metrics for CAN-bus injection or falsification attacks, even though injection attacks are often a necessary step before achieving vehicular compromise. To solve this lack of benchmarks, we survey existing literature on vehicular cyber-attacks and identify some of the more common sensor-falsification attacks. We categorize this into different types of falsification attacks, and evaluate `CarSec` against these attacks to demonstrate its effectiveness in a realistic scenario.

Challenges in Estimating Vehicular Sensor Values Estimating the readings of vehicle sensors using smartphone sensors requires a robust method which is resilient

to common phone usage. Prior work on vehicle sensor estimation requires data from installed sensors and has not addressed the difficulties associated with common phone-based movements [64, 94]. In contrast, using the vehicle's physical dynamics models (e.g., Ackerman model [64]), publicly available vehicle specifications from the OEMs (e.g., gear ratios), and sensor fusion techniques (e.g., complementary filter, neural network), we have built a system that is robust to common phone usage/movements. We demonstrate this through extensive experiments.

Furthermore, to overcome the lack of a well-defined vehicle model for estimating the current gear position, we have developed a neural network to predict the gear position given the change in recent speed.

Key Findings. We have implemented `CarSec` in Android and evaluated it against sensor-falsification attacks. We evaluated `CarSec` by injecting three types of sensor-falsification attacks reported in literature — *sudden*, *gradual*, and *delta* injections — in real-world driving traces. `CarSec` is shown to be capable of detecting falsifications of 6 vehicular sensors with different levels of true positive rate (TPR), ranging from 97.33% for speed sensors to 78.29% for RPM falsifications, with very low false positive rate (FPR), often set to less than 1% FPR. `CarSec` can most accurately detect speed, gear position, fuel and odometer falsifications.

Furthermore, `CarSec` only consumes ≈8% of the CPU on average, even while performing expensive neural network-based sensor estimation algorithms.

This chapter makes the following main contributions:

- *Thorough characterization of CAN-bus sensor falsification attacks.* We survey the literature on vehicular cybersecurity to create a taxonomy of different attacks on vehicular CAN bus. This taxonomy of attacks will guide future research into defenses against such attacks.

- *Extensive evaluation of vehicle sensor estimation algorithms under diverse realistic situations.* We show the robustness of vehicle sensor estimation algorithms

on 6 different vehicles, 912 miles of driving, and under 4 different common uses of the phone by 7 different passengers.

- *Expansion of vehicle estimation research.* We extend prior work on vehicle estimation by (a) developing a novel neural-network based gear-estimation algorithm and (b) investigating the failure modes of engine RPM estimation using prior methods.

- *Development and evaluation of* `CarSec` *a novel system to detect CAN-bus injection attacks.* We use phone-based vehicle sensor estimation to detect CAN-bus injection attacks. To the best of our knowledge, this is the first to apply phone-based vehicle estimation algorithms for the detection of CAN-bus sensor falsification attacks. We also demonstrate its computational feasibility with an efficient Android implementation and provide source code facilitating future follow-up research in this area.

Paper Organization. The chapter is structured as follows. Sec. 4.2 reviews other related IDS solutions while Sec. 3.3 presents the necessary background and attack model. Next we describe `CarSec` in Sec. 4.3 and evaluate it against simulated and realistic attacks in Sec. 3.5. Finally, we discuss limitations of current `CarSec` and future work in Sec. 4.5, and conclude the chapter in Sec. 4.6.

3.2 Related Work

3.2.1 Phone-based Estimation of Vehicular Sensors

There are numerous studies which estimate vehicular sensors using phone sensors for dangerous driving detection [45, 67, 60, 113, 29], road monitoring [32, 47, 92, 118, 40], and trajectory mining [23, 106, 12, 65]. In contrast to these studies, we explore if phone sensing can be used to enhance security. Thus, the evaluation for

this exploration is very different from existing studies.

3.2.2 Vehicular Intrusion Detection Systems (IDS)

These fall under two main categories, CAN-bus traffic characterization and vehicular sensor modeling, both of which essentially rely on vehicular sensors and are thus vulnerable to compromise.

3.2.2.1 CAN-bus Traffic Characterization

Müter *et al.* [72, 73] developed an IDS which models information-theoretic and structural patterns of the CAN bus under normal behavior. During an attack, they observed that these information-theoretic properties, such as entropy, are likely to change. However, this only holds true for CAN-bus injection attacks that deviate from the normal behavior of the ECU. If the attacker is able to mount the attack without changing the behavior of the CAN bus — such as through a bus-off attack [19] or Bootrom attack [68] — it may not result in a significant change in the entropy score and thereby evade detection. In contrast, `CarSec` does not model the CAN-bus traffic. It externally validates the sensor values using their estimation based on smartphone sensor readings. This makes it possible to detect sensor falsification attacks even in the presence of no abnormal CAN-bus traffic.

Cho and Shin [20] used clock-based fingerprinting of ECUs to identify misbehaving ECUs, which is orthogonal to `CarSec`. Once `CarSec` is used to determine that an attack is taking place, we can use the system presented in [20] to pinpoint the culprit ECU.

3.2.2.2 In-Vehicular Intrusion Detection Systems

Other IDSes model the normal behavior of the vehicle by comparing with other vehicle sensors on the CAN bus. For example, Ganesan *et al.* [39] use cross-correlation

to constrain the possible values of different ECUs within the CAN bus. Wasicek *et al.* [107] use neural networks to model the relationship between engine RPM, torque and speed. Cho *et al.* [21] used a random forest to model the brake behavior in different road and weather conditions. All these approaches rely on sensors within the CAN bus and may be susceptible to the same adversary who can inject data into the CAN bus. In contrast, `CarSec` uses the smartphone sensors, which are an external source of knowledge, to cross-validate the internal sensors within the vehicle. Even if the vehicle has been compromised, this additional source of redundancy can provide a measurement of true sensor values.

In addition to providing an external source of knowledge, smartphones are also freely and readily available to enhance car security. Furthermore, they are personal devices which form a natural interface to the user. If there is a problem with your car, your phone can warn you and help you take preventative and diagnostic measures.

3.3 Background and Threat Model

Vehicles are built with many ECUs which are responsible for different vehicular functions and usually communicate with each other using one or more Controller Area Network (CAN) buses. These buses are broadcast-based and use a decentralized protocol for access arbitration and real-time communication.

Sensor-falsification attacks use these communication networks to broadcast sensor data masquerading as one of the ECUs. Since the CAN protocol does not identify/authenticate senders/receivers, an attacker capable of writing to CAN bus can achieve this [57, 69, 68]. For example, an attacker can gain write access using an Internet-facing Infotainment system, forced Bluetooth pairing or even a corrupted audio file with a firmware update [14]. Once successful, the attacker can mount an attack which involves writing falsified messages to the CAN bus. The intention of this attack may be to confuse the user (e.g., [69] p.32, [57] p.456), or to control the

Ref.	ID	Page #	Target ECU	Sensor spoofed	Purpose of attack
[69]	1	32	Instrument Cluster (IC)	Speed	Confuse user
	2	32	IC	Engine RPM	Confuse user
	3	34	IC	Odometer	Confuse user
	4	38	Pre-Collision System (PCS)	RADAR-state	Sudden brake, disable gas pedal
	5	43	Intel. Park. Assistant System (IPAS)	Gear, Speed	Control steering
	6	45	Lane Keeping Assistant (LKA)	Camera-state	Control steering
	7	59	IC	Fuel gauge	Confuse user
[57]	8	455	IC	Fuel gauge	Confuse user
	9	456	IC	Speed	Confuse user
[70]	10	84	IC	Engine RPM	Confuse user
	11	85	Anti-lock Brake System (ABS)	Imminent collision state	Cause ABS to apply brakes
[68]	12	7	IC	Speed	Confuse user
	13	23	Power Steer. Ctrl. Module (PSCM)	Engine RPM	Put ECU in diagnostic mode
	14	27	Park. Assistance Module (PAM)	Speed	Control steering

Table 3.1: Example CAN bus injection attacks which require falsifying vehicular sensors.

vehicle (e.g., [68] p.27), as enumerated in Table 3.1 and categorized in Fig. 3.2.

3.3.1 Why Smartphones?

`CarSec` uses smartphone sensors for cross-validation of vehicle sensors. Smartphones are attractive to play this role for several reasons.

- They are powerful and capable computers with multiple ways to sense the outside environment. For example, the latest flagship phone from Samsung has a 2.7/1.8 Ghz 8-core processor and 11 sensors [58].

- They are ubiquitous and always-on-and-with-person. One study found that 77% of U.S. adults have smartphones [13]. Given their prevalence, smartphones can act as a free source of data redundancy in our vehicular IDS.

- Smartphones undergo a frequent refresh rate and on average, users buy a new smartphone every ≈2 years [56]. In a span of two years, phones often undergo significant improvements in their resources including CPU, memory, OS, sensors, etc. For example, the Galaxy S series significantly improved in processing power (2.3 Ghz quad core in 2016 to 2.7/1.8 Ghz octa core in 2018) and software (Android 6 in 2016 to Android 8 in 2018).

3.3.2 Adversary Model

We model our adversary after what is normally found in related literature. At the bare minimum, the adversary has write access to the CAN bus (e.g., through the UConnect infotainment system, as demonstrated in [70]) and uses this access to broadcast falsified sensor values. S/he uses falsified sensor readings to confuse the user or force ECUs within the vehicle to misbehave. The attacks can vary in how quickly they are accomplished and must be detected.

3.3.2.1 Trusted Components

In order to run `CarSec`, the driver must carry a smartphone equipped with IMU and GPS sensors. We assume the smartphone is not compromised and is mounted on the windshield or placed in the cup holder. Smartphone security is an active area of research and there exist numerous deployed solutions to ensure that the software running on a smartphone has integrity and hasn't been compromised. Besides, securing smartphones is orthogonal to our work.

Additionally, the driver needs to connect an OBD dongle to the OBD-II port found under the steering wheel. These are cheap and widely available devices used by many people for diagnostic and telematics purposes. We assume that the OBD-dongle is not compromised. We envision an OBD dongle which has very limited functionality and only serves to dutifully relay the information it reads on the CAN bus. A simple OBD dongle which only serves the purpose of relaying information doesn't need Internet connectivity and doesn't need programmability. Although prior work has shown remote compromise through the OBD dongle [37, 70], its target was dongles that have Internet connectivity and sophisticated programmability. This sophistication makes them vulnerable to attacks from the Internet.

3.3.2.2 Attacks to Be Covered

`CarSec` detects *sensor falsification attacks* which are mounted via the CAN bus that can be externally cross-validated using the smartphone sensors. Sensor falsification on the CAN bus is often the first step in vehicular cyber-compromise reported in literature [68, 69, 70, 57]. As such, it is important to detect and thwart the attacks at this stage before they progress further.

We assume the sensor-spoofing happens through CAN-bus injection. In other words, it doesn't involve externally spoofing the sensor values via LiDAR injection or other means. There are numerous examples of CAN-bus injection attacks reported in literature. For example, [57] p.458 demonstrates a compound attack which involves falsifying a speedometer reading and [69] p.43 presents a detailed attack that controls the steering wheel angle by falsifying gear and vehicle speed readings.

3.3.2.3 Attacks Not to Be Covered

`CarSec` is not designed to detect CAN-injection attacks which do not involve sensor falsification. For instance, `CarSec` cannot detect CAN-injections used to lock/unlock the door found in [69] p.58. Additionally, `CarSec` is not designed to detect CAN-injections where the spoofed sensor value reflects the actual vehicle sensor value. This happens when the spoofed value is the same as the real value or when the vehicle reacts to the spoofed value and changes its state to resemble the input value (e.g., [68] p.22).

3.4 System Model

We estimated six in-vehicle sensors using smartphone sensors. We chose to focus on these 6 sensors for the following reasons. They are falsified in various attacks found in literature on offensive vehicle security. Furthermore, they are related to vehicle

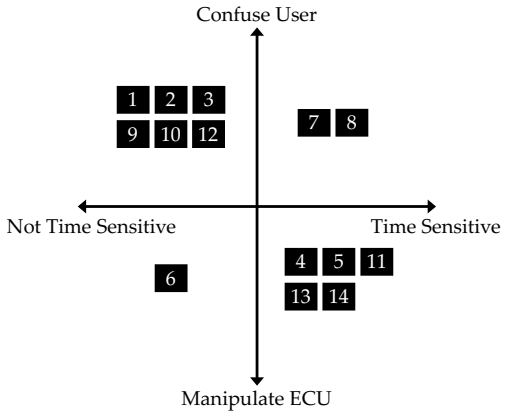


Figure 3.2: Sensor falsification attacks can be characterized in two axes:intention and time sensitivity. The attack IDs are explained in Table 3.1.

dynamics, thereby making it possible to replicate and cross-validate them using the external smartphone sensors. There are also increasing commercial interests in these sensors, making them available to consumers.

These sensors and their estimation equations are summarized in Table 3.2a. For some of them, we used estimated sensors as input to estimate additional sensors. The dependency of sensor estimation is shown in Fig. 3.12a. `CarSec` cross-validates these estimations with the sensor values reported via the CAN bus to confirm or raise a warning about the reported values. We give the details of 5 sensors below. The odometer can be accurately estimated by accumulating the Haversine distance between consecutive GPS points.[1]

Some of these sensor estimations require *per-vehicle* calibration. For example, we trained a different neural network for each vehicle in our dataset for Gear estimation, and loaded vehicle-specific parameters for engine RPM or fuel MPG. These must be done one time for each vehicle model and can be done by the OEM before release. `CarSec` doesn't require *per-smartphone* calibration. The same vehicle-model can be loaded on different smartphones.

[1] `https://en.wikipedia.org/wiki/Haversine_formula`

Sensor	Estimation Method
IMU-align	$\boldsymbol{R} = [(\vec{V} \times \vec{G})^T; \vec{V}^T; \vec{G}^T]$
Speed	$v_t = \alpha(v_{t-1} + Acc_Y * dt) + (1 - \alpha)GPS_v$
Gear	Neural-network based on vehicle speed
Steering [64]	$k * arcsin(l * yaw_{rate}/v)$
Eng. RPM [94]	$v * (F_r * G_r)/T$
Odometer	Haversine sum of consecutive GPS
Fuel level	Distance * Average MPG

(a) Estimation of 6 different sensors used in `CarSec`. When an attack is detected, `CarSec` compares the approximations with the values on CAN. For details on each sensor, see their respective sections below.

Speed Offset in Seconds
{-0.5, -0.4, -0.3, -0.2, -0.1, 0}
{-5, -0.5, -0.4, -0.3, -0.2, -0.1, 0}
{-1, 0}
{-3, -2, -1, 0}
{-4, -3, -2, -1, 0}

(b) Each feature vector represents the vehicle's speed sampled at different time offsets in seconds. For each vehicle, we searched through each of these to find the most optimal one.

Table 3.2: Summary of estimation equations.

3.4.1 Speed

We fuse the speed estimates from both the accelerometer and GPS sensors using a complementary filter. The integration of consecutive accelerometer readings is an estimate of the speed, but due to the noise in the IMU sensor readings, this results in very divergent and incorrect speed estimates. Alternatively, we use the GPS sensor for speed estimates in the order of 1Hz, but it misses more frequent changes which might be sensed by the IMU sampling at 10Hz.

To use the accelerometer, we first align the phone's IMU accelerometer readings to the vehicle's direction of travel. Given consecutive GPS points, we find the angle of the GPS bearing offset from the magnetic north. We then rotate the magnetic north vector from the magnetometer by the same angle to get the vehicle pointing vector from the phone's frame of reference, called $\vec{V}$. We do this using a change of basis transformation from the plane perpendicular to the frame of reference which has basis vectors $\vec{M}, \vec{G}, \vec{G} \times \vec{M}$ where $\vec{M}$ is the magnetic north. Using $\vec{G}$ and $\vec{V}$ we

can calculate the rotation matrix $\boldsymbol{R}$, described in Eq. (3.1).

$$\vec{C} = \vec{V} \times \vec{G}$$
$$\boldsymbol{R} = \begin{bmatrix} \vec{C_0} & \vec{C_1} & \vec{C_2} \\ \vec{V_x} & \vec{V_y} & \vec{V_z} \\ \vec{G_x} & \vec{G_y} & \vec{G_z} \end{bmatrix} \tag{3.1}$$

After calculating $\boldsymbol{R}$, we rotate all accelerometer readings in the vehicle's frame of reference by $\vec{Acc}_v = \boldsymbol{R}\vec{Acc}_p{}^T$ where $\vec{Acc}_v$ and $\vec{Acc}_p$ are the accelerometer vectors in the vehicle and phone frame of reference, respectively.

As in [64], we also found that the GPS-estimated speed is slightly delayed from the actual vehicle speed. We aligned the GPS-speed by shifting it by ≈0.5 seconds, a value we found experimentally in our data. We fuse the Y axis of the aligned Acc_v, and the delay-adjusted GPS_{speed} using a complementary filter $v_t = \alpha(v_{t-1} + Acc_v[Y]dt) + (1-\alpha)GPS_{speed}$. We set dt to 100 ms since our accelerometer sample rate is 10Hz. We search through a training set and set α to 0.33.

3.4.2 Steering Wheel Angle

We estimate the steering wheel angle (SWA) using the yaw-rate of the gyroscope on the phone. To accurately estimate the SWA, we first align the phone's coordinate system with the world coordinate system, and then convert the angular rotation in the yaw axis to SWA. We use a similar rotation matrix as described in $\boldsymbol{R}$ above. Since we are only concerned about the yaw-rate, this only uses the third row of the rotation vector — which is the vector pointing in the direction of gravity. Only using the gravity vector for world-frame alignment is more robust than using the vehicle-facing vector and is sufficient for SWA calculations.

With this new rotation matrix $\boldsymbol{R}'$, we calculate the aligned gyroscopic movement

in the world frame of reference by $\vec{g_w} = \boldsymbol{R}'\vec{g_p}^T$ where $\vec{g_w}$ and $\vec{g_p}$ are the gyroscope vectors in the world and phone frame of reference, respectively.

Once we aligned to the world frame of reference, we use a simplified Ackerman mechanism model to estimate the steering wheel angle using the smartphone's IMU sensors [64]. The steering wheel angle is estimated using:

$$\theta_{steering} = k * arcsin(l * yaw_{rate}/v), \tag{3.2}$$

where k is the steering ratio, l is the vehicle length,[2] v is the vehicle speed and yaw_{rate} is the world-aligned yaw rate calculated using the above transformation.

3.4.3 Fuel Level

We estimate the vehicle's fuel level by using the manufacturer's published average MPG. Starting with a full tank of gas, `CarSec` uses the manufacturer-published datasheet on the tank capacity of the vehicle. As the user drives his vehicle, `CarSec` matches each location of the vehicle to a road segment[3] and labels that as either highway or city-level driving, using publicly available information, e.g., [35]. We take the average MPG for that type of road and multiply it by the distance traveled to get the new fuel tank level.

3.4.4 Gear Position

We focus on automatic transmission vehicles and exclude continuous variable transmission systems. The gear position in automatic transmission vehicles is controlled by the Transmission Control Unit (TCU). The TCU uses inputs from a variety of vehicle sensors to inform its algorithm to upshift or downshift the gear. These sensors include the vehicle speed, throttle position, and many others. Using a detailed

[2]We found both of these in vehicle specifications published online by the respective manufacturers.
[3]We matched it to OpenStreetMap [75].

algorithm, the TCU adjusts the gear position to reduce load on the engine, increase safety of the driver, and reduce the long-term wear and tear of internal components.

Since the smartphone lacks access to many of these sensor values, we trained neural networks based on a vector of the vehicle's recent speed to predict the current gear position. For each vehicle, we found the most accurate feature vector from those listed in Table 3.2b. We also searched through a neural network of depths 1, 2 or 3 where each layer is densely connected with 10 neurons each. The output is a one-hot encoding of the current gear position. We used Tensorflow to train these models, and Tensorflow Lite to run them on Android [1].

3.4.5 Engine RPM

The engine RPM is related to the vehicle speed and the current gear position via:

$$\text{RPM} = v * (F_r * G_r)/T, \tag{3.3}$$

where v is vehicle speed in meters per minute, F_r is the final drive ratio, G_r is the transmission gear ratio and T is tire circumference in meters. We learned F_r, G_r and T using publicly published parameters by the vehicle manufacturer [35]. We used the gear estimate from our gear position estimation described above.

3.5 Evaluation

We first evaluated `CarSec`'s estimation accuracy for each of the 6 sensors (Sec. 3.5.2). Next we evaluated `CarSec`'s sensor-falsification detection accuracy against common falsification attacks found in the literature (Sec. 3.5.3). Finally, we implemented `CarSec` in Android and measured the computation resource usage and latency of each component (Sec. 3.5.4).

Vehicle model	Trips	Hours
Mid-size sedan 2018	21	7.08
SUV A 2017	12	4.87
Compact sedan A 2017	2	0.91
SUV B 2016	44	9.73
Hatchback 2016	2	0.73
Compact sedan B 2012	31	5.07
Total	112	28.56

Table 3.3: Driving dataset collected for evaluation of `CarSec`. We use the OpenXC [36] platform to access the CAN bus data in all test vehicles. We collected a total of 712.8 miles of data.

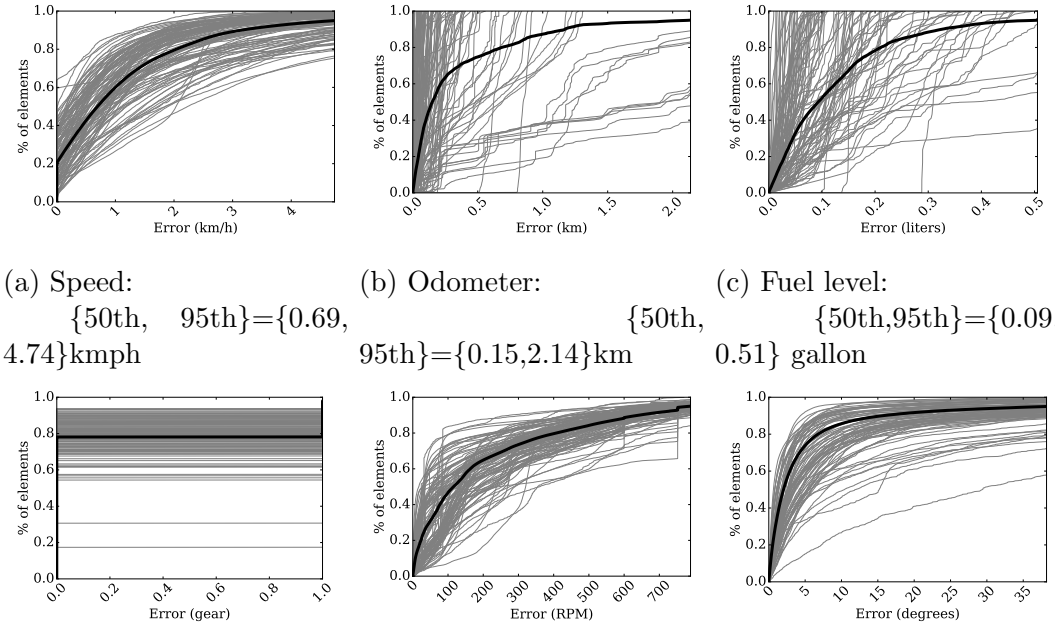
3.5.1 Evaluation Dataset

We evaluated `CarSec` by driving around and collecting ground truth data from the CAN bus and smartphone sensors. Our evaluation required in-vehicle data that is beyond the scope of the OBD-II diagnostic standard, so we used OpenXC [36] to collect data from test cars. We collected data from 112 trips for a total of 28.56 hours and 712.8 miles of driving. The trips covered highways and surface streets. We collected data from 7 different Ford vehicles and 3 drivers, summarized in Table 3.3. The drivers were asked to place the phone in a natural stationary location such as the windshield, cup holder or their pocket. More diverse phone placements while the phone is used by the *passenger* is shown in Sec. 3.5.2.7.

In the following evaluation, we estimate the in-vehicle sensors (speed, steering wheel angle, gear, engine RPM, odometer, and fuel level) using smartphone sensors (GPS, IMU, magnetometer). See Sec. 3.4 for details of the estimation algorithms.

3.5.2 Estimation Accuracy

We evaluated `CarSec`'s estimation accuracy in the absence of attacks. We compare the estimated and the ground truth values for all 112 trips. Fig. 3.3 plots the CDF of the errors. The results presented in this section corroborate the estimation accuracy reported by prior work [94, 64]. We go beyond prior work in our gear estimation



(a) Speed: {50th, 95th}={0.69, 4.74}kmph

(b) Odometer: {50th, 95th}={0.15,2.14}km

(c) Fuel level: {50th,95th}={0.09, 0.51} gallon

(d) Gear: {50th, 95th}={0, 1} gear

(e) Engine RPM: {50th, 95th}={115.6,788.5}RPM

(f) Steering wheel angle: {50th, 95th}={2.02, 38.2}°

Figure 3.3: Estimation error of vehicular sensors using `CarSec`. Each CDF shows the estimation error of each trip along with the average error in a thick black line.

evaluation and dig deeper into the common cause of estimation failure for many sensors. Moreover we evaluate the estimation algorithms under normal phone usage scenarios.

3.5.2.1 Speed

`CarSec` can very accurately estimate the speed using GPS sensors — 50th percentile has $<$ 0.69kmph error, and 95th percentile has $<$ 4.74kmph error. The low-frequency speed ($<$ 1Hz) is accurately estimated using the GPS-inferred speed and the high-frequency speed ($>$ 1Hz) is estimated using the aligned accelerometer readings.

3.5.2.2 Gear

`CarSec` also accurately estimates the gear position. Over ≈80% of the time, it can exactly estimate the current gear position, and at 95% percentile, it is wrong by 1 gear position. We observed that the errors are related to where the driver travels. `CarSec` can more accurately estimate the gear position when the trip is predominantly in the highway versus surface local roads (see Figs. 3.6a and 3.6b). We divided the data from the highway and the local road by map-matching the GPS points to the OpenStreetMap database. This effect occurs because the driver tends to stay in the same gear in the highway whereas there is much more fluctuation caused by start and stop behavior on the surface streets. This is further corroborated by the fact that gear estimation error and vehicle speed have a negative pairwise correlation of -0.1, meaning as the vehicle goes faster, there is less gear estimation error.

3.5.2.3 Odometer, Fuel-Level

`CarSec` can also accurately estimate odometer (50th% $<$ 0.15km, 95% $<$ 2.14km) and fuel level (50th% $<$ 0.09L, 95% $<$ 0.51L). Due to the accumulative nature of fuel estimation, we noticed an increasing error as the trip continues for a longer

period of time. The fuel-level estimate and drifting error are shown in Fig. 3.8. The accumulating error only affects the estimates within a single trip. In between trips, `CarSec` estimates the *change* in odometer and fuel level, and is therefore able to detect injection attacks which happen during the trip.

3.5.2.4 Steering Wheel Angle

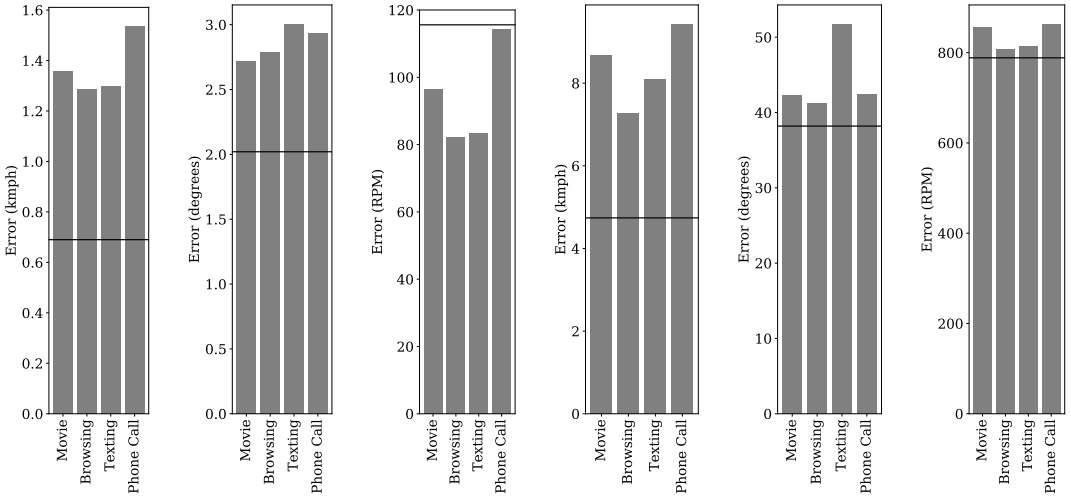
`CarSec` can estimate steering wheel angle with $< 2.02°$ error in 50% of all trips, and $< 38.2°$ in 95% of all trips. We found a strong negative correlation between steering wheel angle estimation and vehicle speed (coef=-0.22) and gear position (coef=-0.31). We found large errors in steering wheel angle when the car moves slowly. This occurs because as the car drives faster, there is a stronger relationship between the induced yaw rate in the vehicle body and the steering wheel angle.

3.5.2.5 Engine RPM

`CarSec` has the most challenge in estimating the engine RPM. The 50% error is 115.6 RPM and 95% error is 788.5 RPM. We use the estimated vehicle speed and vehicle gear position to calculate the likely RPM value (equation shown in Table 3.2a). Even if we use the ground truth vehicle speed and gear position, there is a large variability in the calculated RPM, as shown in Fig. 3.7a.

This large variation in the engine RPM estimation is caused by tire slip. The relationship between the engine RPM and tire speed is mediated by the tire slip ratio [21]. The tire slip is affected by such factors as the road conditions, wear of the tire, and the friction coefficient between the tire and the road. To uncover this relationship, we separated the regions with high acceleration of the vehicle, where there is likely to be high tire slip as the vehicle moves faster, from regions with low acceleration. This difference in the estimation error between the two scenarios is shown in Fig. 3.7b. We found +524 more RPM error (a 1790% increase) for sudden acceleration versus

normal acceleration conditions. Similarly, we found 372% increased error in up-hill versus flat roads and 178% increased error in high-precipitation areas. (Not shown in figure)



(a) 50th - Speed (b) 50th - Steering (c) 50th - RPM (d) 95th - Speed (e) 95th - Steering (f) 95th - RPM

Figure 3.4: Estimation error using the passenger's phone. The black horizontal line shows the 50th and 95th percentile errors for each of these sensors when the phone is more carefully mounted, as done in our previous experiments.

3.5.2.6 GPS blockage noise

`CarSec` relies on GPS to estimate the vehicle speed. We evaluated the effect of GPS error/noise by adding artificial noise to the dataset before estimating the vehicle speed. The results are plotted in Fig. 3.5.

We added normally-distributed noise to the GPS samples (sampled up to once every 100ms). We injected random noise for each 100ms time sample of the GPS signal. Realistic GPS noise is likely to be less jittery and noisy, but our analysis shows the effect of a more extreme noise distribution. We found that as we increase the high-frequency GPS noise, the average error also increases. The trend is that for every additional meter of high-frequency GPS noise, we see an additional error of 1 km/h speed estimation on average. Furthermore, we found that with an increased

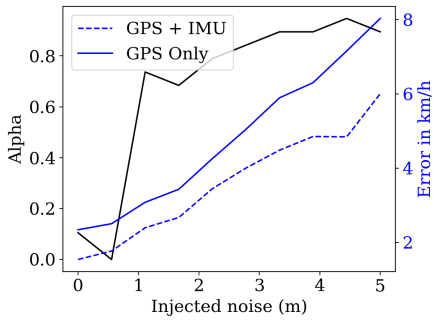


Figure 3.5: Vehicle speed estimation error for increasing high-frequency GPS noise. For higher GPS noise, `CarSec` relies more and more on the accelerometer component of the complementary filter.

GPS noise, `CarSec` also relied more on the vehicle-aligned accelerometer. This is shown as the α curve in our analysis results. α refers to the trade-off between GPS-speed and accelerometer-speed, used in the complementary filter in `CarSec`. To extend this work, we can use the GPS confidence returned by GPS chips to conditionally turn on or off `CarSec`'s estimation capabilities.

3.5.2.7 Additional Redundancy from Passengers' Phones

`CarSec` can be easily extended to make use of additional sources of redundancy from other devices. We demonstrate this possibility by running `CarSec` on passengers' smartphones while they engage in various common activities on their phone. We recruited 7 different passengers to repeat a 6.7 mile-long circuit four times. Each trip around the circuit, the passenger engaged in one of the following four activities — watching a movie, browsing a website, typing out a message, or making a phone call. For each condition we collected ≈2 hours and 50 miles of data. In total we collected 7.9 hours and 200 miles of data.

The 50th and 95th percentile estimation errors are shown in Fig. 3.4. We skipped odometer and fuel since those only depend on the GPS, which is unaffected by who is using the phone. We also skipped gear estimation results since they are the same as the 50th and 95th percentile estimation accuracy presented earlier in this section.

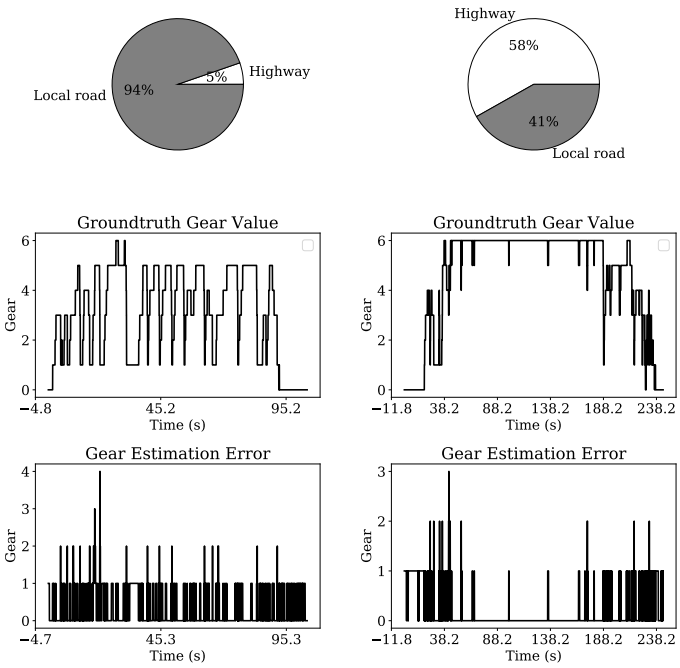
Across all sensors, the estimation accuracy is only minimally affected by the passenger using the phone. There are two main factors which affect the estimation accuracy while the passenger uses the phone.

The first factor is how much the user interacts with the phone. We found that as the user types on the phone or makes a phone call, the steering wheel estimation accuracy tends to be worse. Especially when the passenger types on the phone, the 95th percentile steering wheel estimation error is worse compared to other activities. The steering wheel estimation heavily makes use of the IMU sensors to both align to the world-frame and convert the gyroscope readings to steering wheel estimates. Increased noise in the IMU readings results in worse estimations. Speed is also affected by typing (95th percentile texting is worse than browsing) but this is overshadowed by a second factor.

The second factor is the phone orientation. When the passenger watches a movie (in landscape orientation) or places a phone call, the phone is no longer oriented in the direction of the car's travel. For both browsing and texting, the phone is in portrait orientation and faces the direction of the car's travel. Since speed estimation uses the accelerometer in the direction of the car's travel, this is more sensitive to the phone orientation.

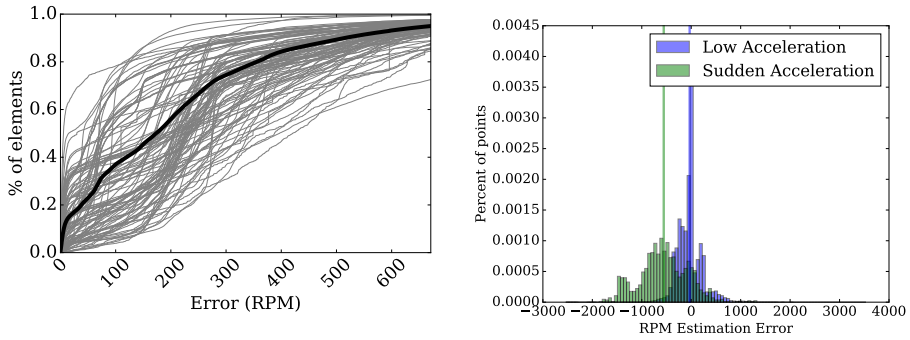
The second factor doesn't affect steering wheel estimations as badly because the phone only needs to the re-oriented to the world-plane to get the angular movement. We simply do this using the gravity vector. In contrast, the speed estimation requires that we get the oriented acceleration along the axis of the car's movement, which is a more constrained estimation of the phone's orientation.

`CarSec` can similarly be extended using other IoT devices such as the driver's smartwatch.



(a) Local road gear error (b) Highway gear error

Figure 3.6: Engine RPM and tire slip investigation



(a) Engine RPM estimation error using the ground truth vehicle speed and gear position.

(b) Distribution of RPM errors. Figure best seen in color.

Figure 3.7: Engine RPM and tire slip investigation

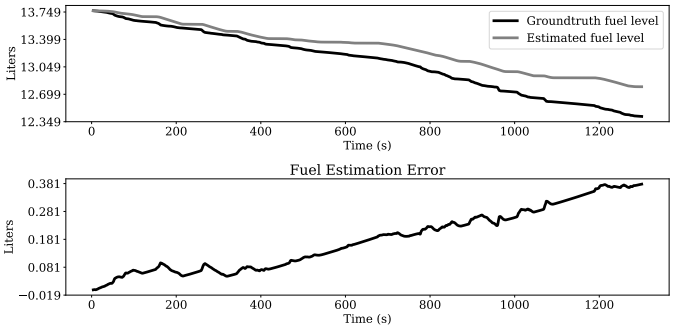


Figure 3.8: Fuel estimation error drift

3.5.2.8 Key Findings

We corroborated the errors reported in prior work in estimating vehicle sensors [64, 21]. We also made three key findings, summarized below.

- Phone-based estimation algorithms work reliably even while the phone is being used for common activities such as texting or making a phone call.
- Gear position can be accurately estimated using a neural network that accepts the recent change in vehicle speed.
- By conditioning for various factors, we discovered that engine RPM estimation is plagued by tire slip. This is especially predominant in steep road segments or regions with high-acceleration applied to the car.

3.5.3 Sensor-Falsification Detection Accuracy

Next, we use the best settings learned from the estimation accuracy evaluation (Sec. 3.5.2) to evaluate the of sensor-falsification detection accuracy. This evaluation first involves injecting data into the CAN bus to mimic realistic attacks (Sec. 3.5.3.1) and then detecting intrusion using `CarSec` (Sec. 3.5.3.3).

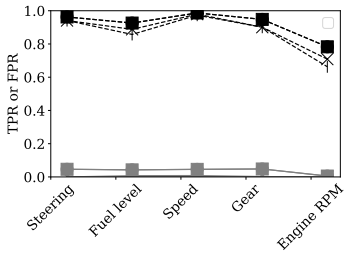
3.5.3.1 CAN-bus Injection

We injected data into the collected data traces to simulate *sudden*, *gradual*, and *delta* injections. The attacker might resort to one of these three forms of injection depending on the intended outcome. For example, in [68] p.23, an attacker spoofs false RPM values to put an ECU into diagnostic mode. In this case, the purpose of the attack is to quickly put the vehicle in diagnostic mode, so the attacker would flood the CAN bus with the target RPM value, a *sudden* injection. Alternatively, in [69] p.38 or [57] p.458, the attacker gradually changes the falsified values to control the vehicle or stealthily confuse the user, a *gradual* injection. An attacker might also falsify values that are consistently different from the current value so as to mimic normal but incorrect behavior. For example, in [57] p.458, the attacker falsified the speedometer value to be 10mph below the actual speed of the car, a *delta* injection. Table 3.4 details the injection types, sensors, and values.

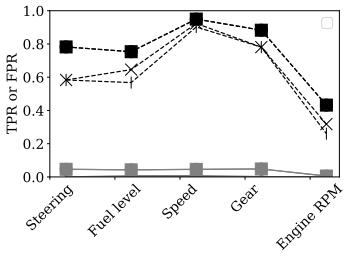
	Sensor	Values	Rationale
Sudden, Grad.	Speed	{ 4, 10, 25, 50, 100 } kmph	Used in manipulating other ECUs E.g. [69].
	Steer. Wheel Angle	{ -100, -50, -10, 10, 50, 100 } degrees	Trick adaptive headlight into shining in the wrong location.
	Gear	{ -1, 1, 4, 6 } gear	Used in manipulating other ECUs, e.g., [69].
	Engine RPM	{ 100, 1000, 2000, 5000 } RPM	Low engine RPM used to put ECUs into diag. mode [68].
	Fuel level	{ 15.4 } gallons	Trick driver into emptying gas tank, e.g., [57].
Delta	Speed	{ -50, -10, +10 } kmph	Trick driver into going faster. E.g., [57].
	Steer. Wheel Angle	{ +10, +50 } degrees	
	Odometer	{ -1000 } km	Trick driver into missing oil change dates.
	Fuel Level	{ +0.5 } gallons	Trick driver into driving without sufficient fuel level, e.g., [57]

Table 3.4: Specific injection values used in our analysis. The first five attacks were injected as a sudden or gradual injection. The last four were injected immediately as a delta of the actual value. See Sec. 3.5.3.1 for more details. These injection values were inspired by existing literature and extend beyond them.

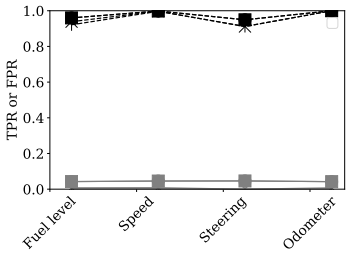
In all three cases of injection, we measured the difference between smartphone-estimated value and CAN-bus reported value, and counted the instances of true positive (TP), true negative (TN), false positive (FP) and false negative (FN). We used the normal case without injection to count FP and TN, and the injected case to count TP and FN. In what follows, we compare the True Positive Rate (TPR) and False Positive Rates (FPR) of various conditions. The TPR (also known as *recall*) is



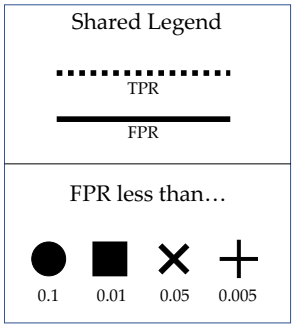
(a) *Sudden attack* where the attacker immediately sets the sensor value to the target value.



(b) *Gradual attack* where the sensor value gradually changes to the target over a span of 10 seconds.



(c) *Delta attack* where the attacker sets the injected value to a offset of the actual value.



(d) All graphs share the above legend. Legend consolidated to save space.

Figure 3.9: The true positive rate (TPR) and false positive rate (FPR) of different conditions. We considered gradual, sudden, and delta injections. See Sec. 3.5.3.1 for more details. For each condition, we restricted the maximum FPR and adjusted the parameters to yield the highest TPR. As we reduced the FPR requirement, the TPR also suffered.

defined as $TP/(TP+FN)$ and the FPR is defined as $FP/(FP+TN)$. Configuring `CarSec` results in a tradeoff between TPR and FPR. A cautious driver would prefer to increase TPR at the risk of more alarms. However, if the FPR is too high, it will raise too many false alarms and might lead to the driver ignoring the alarms during a real attack.

3.5.3.2 Warning Threshold

`CarSec` uses two parameters before flagging an on-going attack — the attack **magnitude** and **duration**. By adjusting these parameters, `CarSec` can be configured to trade off false positive rates (FPR) with true positive rates (TPR). The first parameter specifies a threshold difference between the estimated `CarSec` and OBD values before flagging it as anomalous. This can be caused by an attack, by a faulty sensor, or by estimation inaccuracy due to inaccurate smartphone sensors. The second parameter disambiguates this by looking for a sustained deviation from expected values. The first parameter is in a different unit for each sensor. For example, the steering wheel angle sensor uses "degrees." The second parameter is in seconds.

We search through 100 combinations of both parameters and calculate the receiver operating characteristic (ROC) curve. We set the **magnitude** threshold to one of 10 different values, defined independently for each sensor, and set the **duration** threshold to one of 10 different values equally ranging from 100ms to 5s. Fig. 3.10 is the ROC curve for speed-falsification detection. With the ROC curve, we search for the configuration which yields the maximum TPR for bounded FPR values, where FPR is bounded to $\{0.1, 0.5, 0.01, 0.001\}$

3.5.3.3 Detection Accuracy

Fig. 3.9 shows the FPR and TPR for different attacks, and detection thresholds. `CarSec` is able to most accurately detect attacks which falsify speed, fuel level or

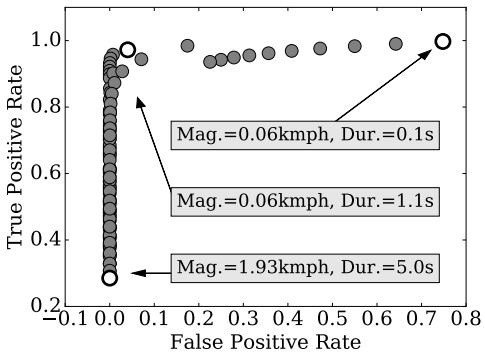


Figure 3.10: ROC curve of 100 different combinations of the two parameters — attack magnitude and duration. For each combination, we calculate the FPR and TPR. A similar ROC curve was computed for each of the 6 sensor estimations.

odometer values.

Speed For *sudden* falsification attack, `CarSec` can detect speed injection attacks with TPR=97.33%, FPR=0.2% and for a *delta* attack, `CarSec` can detect the injection with TPR=99.67% and FPR=0.2%. However, if the attacker *gradually* spoofs speed values to the desired target value, `CarSec`'s accuracy drops to TPR=90.25% at FPR=0.2%. If the user is willing to tolerate more false alarms, `CarSec` can detect gradual speed spoofing attacks at TPR=94.98% at FPR=4.56%.

Fuel-level We found a similar pattern in the fuel-level attacks. `CarSec` can detect a fuel-level delta attack at TPR=93.86% at FPR=0.77%, and a sudden attack at TPR=88.8% at FPR=0.77%. However, it only detects the gradual attack at TPR=64.57% at FPR=0.77%. In the subsequent analysis, we uncovered the reason for poorer performance in detecting gradual-level attacks. In a gradual attack, injected values closely resembles the actual values during initial stages of the attack. Only after a few seconds of the 10-second gradual attack does the value start to differ. We decouple this factor and study the detection accuracy by the percentage of injection in Sec. 3.5.3.4.

Steering Wheel Angle `CarSec` is able to detect delta steering wheel injection attacks at TPR=91.98% for FPR=0%, sudden-injection attacks at TPR=94.12% for FPR=0%, and gradual-injection attacks at TPR=58.33% for FPR=0%. The steering-wheel detection is less accurate for gradual injection attacks due to the inherent difficulty of estimating the steering wheel angle using IMU sensors. This is especially a problem when the vehicle is traveling very slowly or stopped at a stop sign. The driver may move the steering wheel significantly but the IMU sensor will be unable to estimate these values. Therefore, in order to improve the TPR, we have to accommodate a much higher FPR, which may not be acceptable for drivers.

Gear Position `CarSec` can accurately detect any gear injection attacks — TPR=90.08%, FPR=0.22% for sudden injection and TPR=88.3%, FPR=4.79% for gradual injection. Contrary to other sensors, even a gradual gear injection can be detected with high accuracy because of the discrete nature of gear values. The smallest injection can change the gear value by 1 gear position. As seen in Fig. 3.3, we can estimate the gear position correctly almost 80% of the time.

Engine RPM Mirroring what we saw in the estimation accuracy, `CarSec` has the most difficulty in detecting RPM injection attacks. Sudden attacks have TPR=78.29%, FPR=0.54% and gradual attacks have TPR=43.22%, FPR=0.54%. This resembles our estimation error which is caused by tire slip, as shown in Fig. 3.7b. In the ideal case, the RPM estimation without considering tire slip is still incorrect by just over 1000 RPM values in the 99th percentile. If we only look at injection between 1000–1200 RPM values, we can detect them with 87.23% TPR. We further investigate the relationship between attack magnitude and detection rate in the next section.

3.5.3.4 Smallest Detectable Attack

We measured the *minimum injection* that `CarSec` can detect. This analysis helps us disentangle the type of injection from our detection method. Fig. 3.11 shows the result of this analysis.

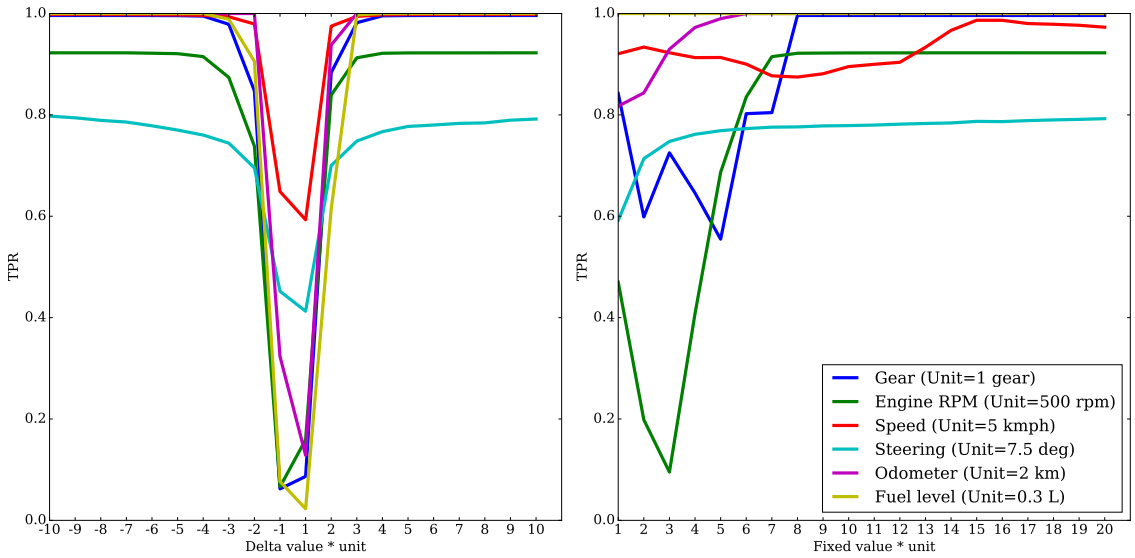
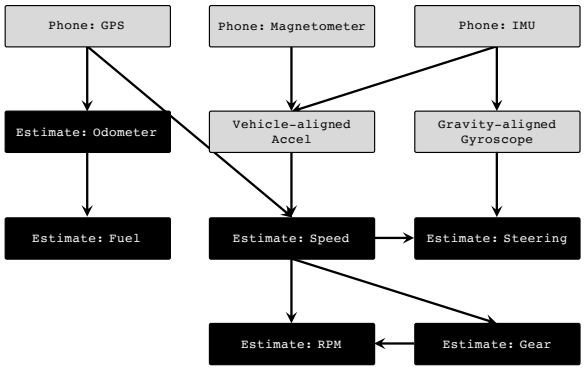


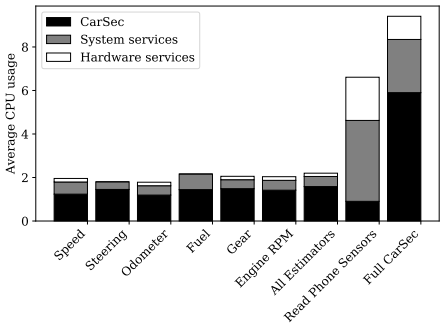
Figure 3.11: The left figure shows TPR for varying amounts of injection magnitude. The right figure shows varying *fixed* values of injection. At the moment of the injection the vehicle sensor varies depending on the scenario. For instance, in some cases the car was traveling at 10 kmph when there was a 4kmph injection. For all cases, FPR was less than 5% and therefore omitted. Figure best seen in color.

As shown in Fig. 3.11, we injected different magnitude values for each sensor deviating from the actual value. The X-axis is the magnitude of the injection where each unit is multiplied by the scale specific to that sensor, shown in the figure legend. We can detect speed injection once it exceeds 10kmph. Similarly, the minimum bound threshold for other sensors are: engine RPM=1000 RPM, Gear=2, steering=22.5°, odometer=4km and fuel level=0.9L.

On the left figure, we show varying magnitudes of a *delta* injection. As the injection is close to 0 delta (i.e. the true value), the TPR drops to 0. On the right figure, we show the TPR for fixed injection values. If the fixed value is close to the actual value (e.g., setting engine RPM to ≈1500) then the TPR becomes very low.



(a) The architecture of our `CarSec` implementation. The gray modules contribute to the estimated outputs. The black modules are the six estimators.



(b) Average % CPU utilized for each individual component of `CarSec` and the combined operation of `CarSec`.

Figure 3.12: Implementation details and measurements

3.5.3.5 Key Findings

The key results of intrusion detection accuracy evaluation can be summarized as follows. These findings are key contributions of our work, which is the first to investigate using phone sensors to detect CAN-bus injection attacks.

- Mirroring what we found in estimation evaluation (Section 3.5.2), `CarSec` can detect injection to the vehicle speed, fuel level, steering wheel angle, gear and odometer with high true positive rate (TPR), but has a lower TPR for engine RPM.
- The TPR depends on the nature of the attack. It is lowest during a gradual attack - this is due to injections of very low magnitude at the onset of the attack.
- The TPR increases as the attack magnitude increases and starts to starts to exceed the estimation error.

3.5.4 Android Implementation and Evaluation

We implemented `CarSec` as an Android app for all devices running Android 7.0 or higher. `CarSec` reads vehicular data via a Bluetooth-based dongle attached to

the vehicle's OBD port. It uses the internal phone-based sensors to estimate the same sensors from the OBD dongle, and reports any suspicious looking discrepancies to the user. `CarSec` infrequently polls nearby Bluetooth devices to detect if it is within proximity of the vehicle. Once it detects that the user is near their vehicle, it establishes a connection, and starts security-assistance operations.

The Bluetooth latency incurred by `CarSec` communicating the information from the OBD dongle to the phone is negligible (due to short distance and low traffic) and all data are aligned to the same time using timestamps. Furthermore, as we show below, the time to run each estimation algorithm is also within the order of milliseconds, making `CarSec` a very time-efficient implementation.

We implemented each of the six sensor estimators into `CarSec` in a modular fashion. Each module has a set of inputs which it uses to calculate the estimated value. The inputs can be raw phone sensors or outputs from other module estimators. Fig. 3.12a shows the relationship between the input and output of each module. Each module's outputs are routed to other modules which depend on them. This modular implementation pattern avoids replicated code and operations within the resource-constrained phone. We built each module on top of a vehicular data collection library developed in our lab. The library handles interfacing with Android sensors, with Bluetooth-based OBD dongle, routing data from one module to another, and many other basic requirements of `CarSec`. Each sensor estimation logic took less than 100 lines of Java code to implement. The vehicle alignment and gear estimation modules took the most amount of development effort due to their increased complexity.

We measured the CPU usage of each of the estimators within `CarSec` and the combined operation of `CarSec`. The results are shown in Fig. 3.12b. We separated out the six estimation algorithms within `CarSec` and evaluated their CPU usage in isolation, shown in the first six bars of the figure. These algorithms were tested

using trace of the sensor data, in order to measure CPU usage isolated from data collection. We also measured the CPU usage of data collection as a separate trial shown in the bar titled "Read Phone Sensors", and the full `CarSec` implementation with all estimators and data collection, shown in the bar titled "Full `CarSec`". All our measurements were made on a Google Pixel 2, which has an Octa-core ARM-based Kyro CPU.

For each test case, we ran that subset of `CarSec` for 1 minute and sampled the percent CPU utilized using '`top`'. We divided this value by the number of cores — 8 cores in our test device. We also measured the CPU utilized by two system-level hardware access services (grouped under "Hardware services") and two system services responsible for other operation within Android (grouped under "System services").

As shown in Fig. 3.12b, `CarSec` only uses approximately 2% of the total CPU availability for each of the sensor estimators. Furthermore, when we ran all sensors simultaneously (shown under "All Estimators"), the CPU utilization is still near 2%. The primary overhead of running `CarSec` is not the sensor estimation algorithms; their implementation is very light-weight and make use of the output from each other to reduce total CPU consumption. The primary overhead comes from other Android-related requirements of launching an app.

Next we measured the cost of reading from the smartphone sensors (i.e., GPS, IMU and magnetometer) without doing any sensor estimation. That is shown in the bar labeled "Read Phone Sensors". The `CarSec` process only takes approximately 1% of the CPU, but the system-wide services which are used to read from the sensors take approximately 6%, putting the total at around 7% CPU utilization. Finally, we ran the full `CarSec` implementation with a user interface and it takes approximately 8% of the total CPU utilization.

We expect the driver to connect their phone to the car power source, as is commonly the case. A security-conscious driver may be willing to make this tradeoff to

avoid any battery consumption by `CarSec`.

Each of the six modules which estimate six sensors run in negligible time. We measured this by running `CarSec` and measuring the time required to process and output each of the estimations. We averaged the runtime over approximately 100 runs of each estimator. Speed, steering wheel, fuel, and engine RPM modules all ran in less than 1 ms on average. Odometer estimation ran in 2 ms on average and gear estimator took 5.6 ms on average. The gear module is the most complex as it uses a neural network to predict the gear position, and hence it takes the longest amount of time. Since each estimation is very fast, the real bottleneck is the rate at which the phone provides us with the low-level sensor values. For example, the odometer estimation module depends on the GPS, which is often only available at 1Hz.

3.6 Discussion

`CarSec` brings car-security to everyday drivers. Due to ease of deployability and the widespread, ubiquitous nature of smartphones, our system has the potential for a wide reach.. Discussed below are a couple of remaining issues worth further investigation beyond our current approach.

- **Compromised Phone Sensors.** As described in our adversary model (Sec. 3.3.2), we assume that the phone isn't compromised. Mobile security is an active research area on its own and is orthogonal to our work. We refer the reader to recent surveys on mobile security [34].

- **Additional Redundancy.** We demonstrated that `CarSec` can run on passengers' phones while they engage in common activities on their phones. Similarly, we can extend this work to other devices which have the required three sensors. `CarSec` only needs IMU, magnetometer, and GPS sensors to estimate vehicular sensors. Many existing IoT devices could be used in this way including

smartwatches, fitness trackers, or some mounted after-market devices such as Amazon Echo Auto, [4] as long as they expose access to these three sensors. We leave the evaluation of this extension as future work.

- **Response After Detection.** `CarSec` serves as a two-factor source of knowledge to detect sensor falsification attacks. If an attack is detected, `CarSec` merely notifies the driver of the suspicious activity and they can choose to take the car to a mechanic for further diagnostics. We do not consider automated response based on this information. Automatically responding is risky since the attacker may target the smartphone in order to trigger this response mechanism and immobilize the car. Therefore, we restrict `CarSec` to ***detection*** and leave ***response*** up to the driver.

3.7 Conclusion

In this chapter, we presented `CarSec`, a car-security assistant which uses smartphone sensors to cross-validate and detect vehicular sensor falsification attacks. We used smartphone sensors to estimate six sensors inside the vehicle: speed, fuel level, odometer, engine RPM, gear position and steering wheel angle. Using driving traces and simulated injections, we have demonstrated `CarSec`'s ability to detect injections which have sufficient magnitude of attack (e.g., enough to actually impact the car or confuse driver) very quickly after the attack onset.

We focused on these six sensors, but `CarSec` can easily be extended to many more sensors within the vehicle. Furthermore, through the combination of multiple phones, other IoT devices, and rich media sensors on the phone (e.g., cameras and microphones), we can extend the space of sensors which can be cross-validated using `CarSec`. This work lays the groundwork for future researchers to expand on the idea

[4] `https://www.amazon.com/Introducing-Echo-Auto-first-your/dp/B0753K4CWG`

of using smartphones for car-security assistance.

CHAPTER IV

Ubi: Using GPS Trajectories to Detect Driving Hazards

4.1 Introduction

According to the World Health Organization (WHO), road injuries are one of the leading causes of deaths worldwide in 2018 [76]. The National Highway Traffic Safety Administration (NHTSA) reported that in 2017, dangerous driving alone claimed over 3000 lives [74]. To reduce this fatality, dangerous driving hazards — including potholes, debris, pedestrians or other reckless drivers — must be detected and dealt with in a timely manner.

4.1.1 State-of-the-Art

Automatic detection of road hazards is reserved for high-end vehicles with sophisticated sensors and significant computing power. The majority of drivers without this capability rely on phone-based solutions to be warned of upcoming hazards. Stationary hazards such as potholes or speed traps are collected and shared through popular navigation applications such as Google Maps [66], but these solutions fail to track *moving* hazards such as pedestrians or bicyclists. Furthermore, none of these approaches support tracking highly-mobile hazards such as a fast-approaching reckless

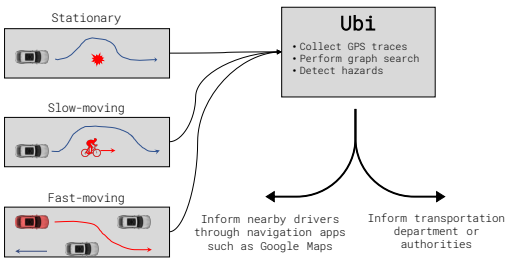


Figure 4.1: `Ubi` detects different driving hazards (shown in red) by collecting GPS trajectories of vehicles around the hazard (shown in blue). This operation is done in a cloud service and broadcast back to the drivers, or to the proper authorities

driver. The state-of-the-art in detecting reckless driving behavior requires collecting GPS or sensor data from the driver in question [117, 111]. This approach is insufficient for an uncooperative reckless driver who wishes to hide his driving behavior from the authorities. Thus, we need a solution to detect and track stationary and mobile driving hazards without their explicit cooperation.

In this chapter, we provide a unified solution for detecting stationary (e.g., pothole), slow (e.g., pedestrian) and fast (e.g., driver) moving hazards. Our solution, `Ubi`, does not require data from the hazard in question and infers the presence through GPS trajectories of, and sightings reported by well-behaving drivers surrounding the hazard.

4.1.2 Proposed Solution

We propose a system called `Ubi` that detects stationary (e.g., potholes, broken-down cars, or fallen tree branches), slow-moving (e.g., pedestrians) and fast-moving (e.g., bicyclists, reckless driver) hazards. Our system, summarized in Fig. 4.1, accepts the GPS trajectories of drivers on the road and any reported sightings of the hazard. Using an internal graph representation of the road segment, `Ubi` identifies the most likely hazard on the road segment, if any, and predicts their location into the future. As new cars approach the hazard's predicted location, `Ubi` sends a warning to the

driver ahead of time.

4.1.3 Ubi Operation

Drivers install `Ubi` on their smartphone along with their regular navigation app. During their daily commute, as they approach a hazard `Ubi` sends them an alert with the type of hazard (e.g., pedestrian) and the current distance from the car. A driver may also press a button on their smartphone to instruct `Ubi` about the presence of a new hazard nearby. On the server side, the cloud operator collects GPS samples from multiple drivers at the same location and tracks the reported sightings of hazards. The server cycles through a library of known hazard types and identifies the most likely hazard that fits the observed sightings. As a driver queries `Ubi` at time t, the cloud operator predicts the location of the hazard at t and warns the driver if they are nearby.

4.1.4 Key Technical Details

`Ubi` detects and tracks mobile hazards using two key ideas.

1. **Mobility model** `Ubi` uses an internal representation of the mobility patterns of each hazard type. A mobility model is the probability distribution of movement over each time step. For example, a pedestrian may only travel at a maximum of $10m$ per second whereas a driver can travel much faster.

2. **Graph representation** `Ubi` combines the GPS trajectories, sightings of hazards, and the mobility model of each hazard using a 3-dimensional occupancy graph (Fig. 4.6). As hazards are sighted, `Ubi` marks individual nodes at that time and attempts to find the best hazard that matches the sightings. For each time step, `Ubi` applies the mobility model for each hazard type to predict how the hazard could have moved from one time step to the next. This prediction is then used to warn future drivers as they approach the hazard.

4.1.5 Results

We use `Ubi` to detect three different hazards in simulation: a stationary pothole, a slow-moving pedestrian, and a fast-moving bicyclist. We demonstrate that in all three cases, `Ubi` is able to accurately classify the hazard type ($\approx 95\%$ accuracy) and provide accurate warnings about the hazard's distance from the driver ($1.75m$ for pothole, $1.5m$ for pedestrian and $3.5m$ for bicyclist). We demonstrated that `Ubi` is resilient to GPS noise as high as $25m$, which is the worst-case scenario in highly-dense neighborhoods [52]. Furthermore, we show that the distance error decreases as the density of cars increases.

4.1.6 Contributions

This chapter makes the following contributions:

1. Development of `Ubi`, a novel hazard detection and warning system that uses GPS trajectories of nearby cars to detect stationary and mobile hazards;

2. A novel graph-based algorithm to track and predict the future locations of mobile hazards;

3. Demonstration of `Ubi`'s effectiveness in detecting stationary potholes, slow-moving pedestrians, and fast-moving bicyclists.

4.1.7 Outline

This chapter is organized as follows. We give `Ubi`'s algorithm and implementation details in Sec. 4.3. In Sec. 4.4 we present the results of evaluating `Ubi`. Finally, we consider future work in Sec. 4.5, survey related work in Sec. 4.2, and conclude in Sec. 4.6.

4.2 Related Work

There have been numerous methods proposed thus far to detect stationary, slow- and fast-moving hazards. However, most of them require direct sensing of the hazard (Sec. 4.2.1). A few of them also detect stationary hazards through indirect sensing of GPS trajectories (Sec. 4.2.2).

4.2.1 Direct: Hazard Detection

Using cameras Most driving hazard detection systems use cameras and other sensors to directly sense the hazard. For example, numerous systems use camera to detect pedestrians [26] or other vehicles [93]. Autonomous vehicles use even more advanced vision sensors to detect hazards in their surroundings [59]. Vision-based systems require advanced processing and installation, which can be expensive in existing vehicles. Our solution simply uses GPS trajectories which is already widely used in navigation systems.

Road anomaly detection Some systems use GPS and IMU sensors to detect stationary hazards, such as potholes or speed bumps [92, 91, 32, 7, 50]. They collect IMU/GPS data from multiple vehicles driving by the stationary hazard and use machine learning techniques to identify the location of the hazard. With a notable exception explored later [91], these systems only work if the vehicle drives *over* the hazardous region. If the vehicle swerves out of the way, it will not register as anomalous IMU sensor data. Furthermore, these systems are only able to detect stationary hazards. In contrast, `Ubi` uses only GPS trajectories to detect stationary, slow- or fast-moving hazards.

Detection of reckless drivers There have been numerous proposals for dangerous driving detection. Many of them rely on data from inside the vehicle [45, 67, 117,

104, 116, 107], smartphone data [113, 60, 61, 45, 18], camera data [112, 96, 53, 104, 61], or GPS trajectories [115, 111, 47, 2, 78]. All of these assume that the collected dataset also includes data from the dangerous driver. In reality, however, this may be missing due to lack of coverage or the active evasion by the dangerous driver. `Ubi` assumes no data from the dangerous driver and uses the behavior of nearby vehicles to reconstruct the likely trajectory and presence of the dangerous driver.

4.2.2 Indirect: Detection based on GPS trajectories

There are some solutions which infer the presence of other objects based on GPS trajectory data. The authors of [90] use swerving behavior to detect the presence of potholes. They observe that drivers tend to avoid potholes, and by aggregating the driving behavior from multiple drivers, it is possible to reconstruct the likely location of potholes. The authors of [47, 12] use GPS trajectories to infer the presence of stop signs and stop lights. They observe that near stop signs, drivers will have a sudden de-acceleration behavior, which can be aggregated in a large scale. [106] uses GPS traces and map matching algorithms to find new road segments which are not cataloged in a digital map. [23] uses common parking patterns in a busy street to automatically infer parallel parking spots. The occupancy of a parking spot changes over time, and during dense times, they use the fact that drivers drive past illegal parking spots to infer that it must be illegal to park there.

All of these approaches use the GPS trajectories to find *stationary* hazards and landmarks, such as potholes or stop signs. In contrast, `Ubi` goes beyond to *slow-* and *fast-moving* hazards as well.

4.2.3 Graph-based anomaly detection

A few approaches use graph theory to detect abnormal driving behavior. [78] collects GPS trajectory and builds graphs to represent common paths between source

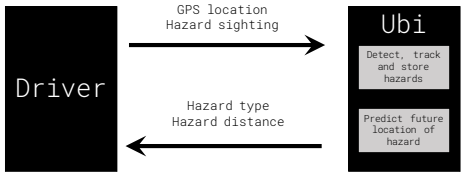


Figure 4.2: `Ubi` system components. Individual drivers upload their current location and optionally a sighting of the hazard if they are nearby. `Ubi` uses subsequent sightings to track the location of the hazard, and returns the distance from the upcoming hazard.

and destination. If the flow of traffic through the graph changes significantly one day, they investigate more closely to find the source of the change and find traffic obstructions. Unlike this, we model the *micro*-traffic using graphs. The nodes in our graph model is the open space and edges represent feasible movement of hazards through the nodes.

[116] uses graphs to represent the state of the vehicle using multiple vehicular sensors such as engine RPM, speed, gear and swerving behavior. Our work uses graph models for an entirely different purpose of finding open spaces and feasible trajectories.

4.2.4 Crowd-sourced detection

Existing navigation applications like Google Maps or Waze use crowd-sourced reports to detect *stationary* obstacles or events on the road, such as a speed trap or stopped car. These rely on manual user inputs and only work for stationary hazards. Our system has the ability to expand the detection repertoire of these navigation systems by also tracking and detecting *mobile* hazards, such as a pedestrian or a reckless driver.

```
def warn_driver (time, gps, sighting):
  all_gps.append(gps)

  # Sighting reported, update hazard model
  if sighting is not None:
    all_sightings.append(sighting)
        likelihood = {}
        for hazard in all_hazard_types:
          likelihood[hazard] = graph_search(
            all_gps, all_sightings, hazard, time)
        likely_hazard = most_likely_hazard(likelihood)

  # No sighting reported
  # Simulate past sighting of hazard and warn driver
  else:
    network = graph_search(
      all_gps, all_sightings, likely_hazard, time)
    location = average_location(network, time)
    return likely_hazard, location
```

Figure 4.3: `Ubi` pseudocode. `Ubi` accepts a timestamped GPS location and whether the hazard/obstacle was sighted at this location. If there is a sighting, it updates the likelihood model. Otherwise, it predicts the current location of the obstacle and warns the driver. The graph search algorithm is described in more detail in Alg. 4.4.

4.3 System Design

`Ubi` is a real-time detection and warning service for upcoming stationary or mobile hazards. As shown in Fig. 4.2, `Ubi` accepts GPS trajectories from multiple drivers near the same location and internally tracks and simulates the location of hazards on the road. `Ubi` uses a three-dimensional graph to represent the trajectories of vehicles and the potential trajectories of hazards. As more drivers drive around the hazard, `Ubi` gains increasing confidence about the type and projected trajectory of hazard. As the driver approaches the hazard, `Ubi` sends a warning with the type of hazard and the estimated distance from the car. The main system components behind `Ubi` are shown in Fig. 4.2 and the pseudocode is shown in Fig. 4.3.

Input:
$\mathcal{T}$ – set of trajectories for all vehicles
$\mathcal{S}$ – set of sightings of hazard
$\mathcal{M}$ – set of obstacles in the map
$\mathcal{H}$ – set of movements for hazard from (x, y)
$\mathcal{E}$ – set of valid regions where hazard can enter

Output: List of possible hazard paths

```
1  Nodes ← ∅;
2  for t ∈ all times in T do
3      Nodes_t ← Nodes_t ∪ E_t;
4      Nodes_t ← Nodes_t ∩ ¬(T_t ∪ M_t);
5      if t ∈ S then
6          Nodes_t ← Nodes_t ∩ S_t;
7      for (x0, y0) ∈ Nodes_t do
8          for (x1, y1) ∈ H_{x0,y0} do
9              collision ←False;
10             Hazard transition = (x0, y0) → (x1, y1);
11             for car ∈ T do
12                 if collides(car_t, Hazard transition) then
13                     collision ←True;
14             for car ∈ T do
15                 Car transition = car_t → car_{t+1};
16                 if collides((x1, y1), Car transition) then
17                     collision ←True;
18             if collision == False then
19                 add node (x1, y1, t + 1);
20                 add edge (x0, y0, t) → (x1, y1, t + 1);
21 return List of possible hazard paths
```

Figure 4.4: Graph search used in `Ubi`

4.3.1 `Ubi` System Input Output

Drivers running `Ubi` share GPS samples of their location and whether a hazard was sighted in their vicinity. The cloud operator pools data from multiple drivers near the same location and uses a graph data structure in two different ways. Suppose `Ubi` has received data from time $t0$ until $t1$. First, `Ubi` uses a graph to simulate an obstacle traveling through the sighted regions between $t0$ and $t1$. `Ubi` attempts to place multiple hazards in this time range amidst the GPS samples and reported sightings. Second, when a new car approaches at $t2 > t1$, `Ubi` simulates the location of the most likely hazard (learned from the first use of graph search above) until $t2$. Next we describe this graph search, which constitutes the core algorithm used in `Ubi`.

4.3.2 Graph Search

`Ubi` uses a graph representation to efficiently find hazards amidst the known vehicle GPS trajectories and through reported sightings. For mobile hazards, `Ubi` also finds the possible likely paths the hazard could have taken through the trajectories. A graph is a natural representation of physical locations and the relationship between them. The nodes in the graph encode the available open space and edges represent feasible movements between nodes. The graph is initialized at the starting time and with a definition of the map including road boundaries and obstacles on the road. The graph search create nodes only where there is free open space that is not occupied by other known cars. Depending on the mobility of the unknown hazard (e.g., a pedestrian, a driver who is turning right at the next intersection) we create edges with different weights and connectivity.

A step-by-step visual walk-through of the graph construction and graph search is shown in Fig. 4.6 and the algorithm is shown in Alg. 4.4.

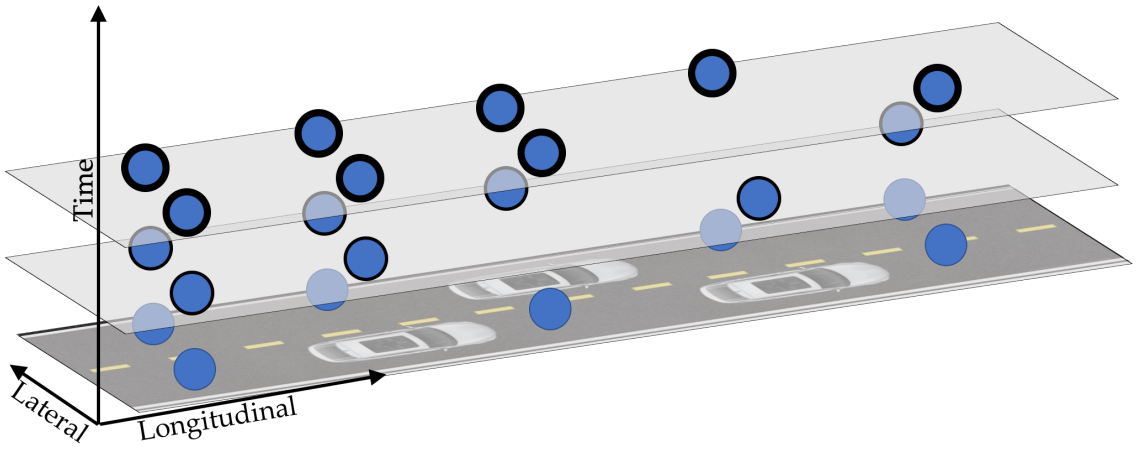


Figure 4.5: The open space around the GPS trajectories are represented using a network. Each node represents a potential location for the hazard.

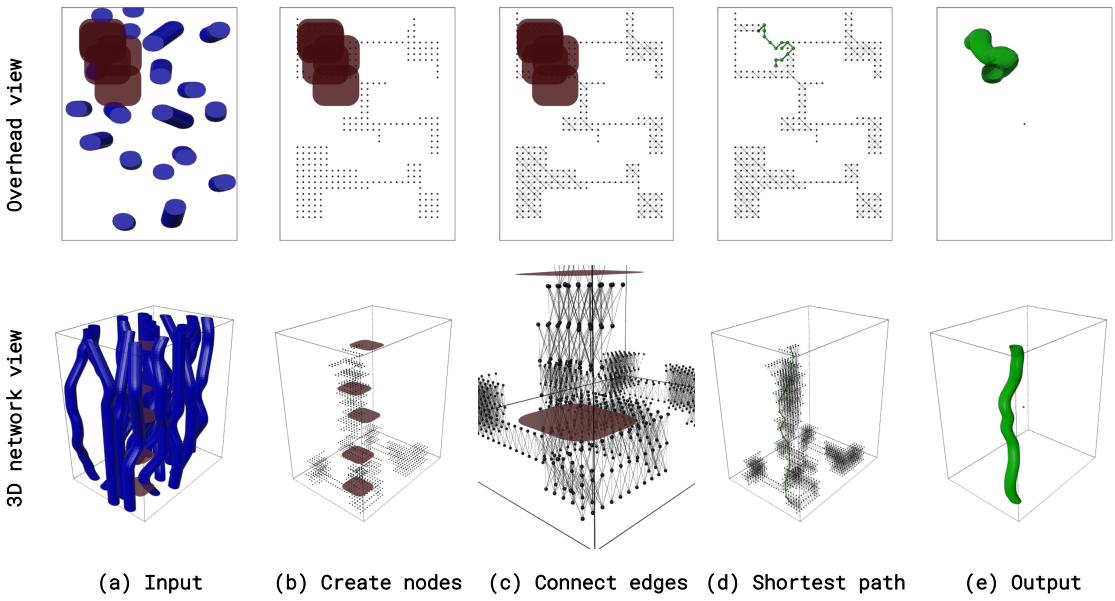


Figure 4.6: An overview of the steps involved in detecting hazards in `Ubi`. This example uses trajectories of simulated bumper cars where the hazard is also a missing bumper car. This is chosen for ease of visualization. The top row shows the overhead view during one time step. The bottom row shows the full 3D graph where the Z axis is used to represent time. The blue lines in the input represent the trajectory of the input. The sightings are shown in red where the missing bumper car was last seen. The third step (c) shows how the edges are connected such that the hazard goes through the sightings. All outside nodes and edges are excluded (Alg. 4.4 #6).

4.3.2.1 Discretization

In order to use graphs to find possible vehicle trajectories, we first discretize the continuous information into a grid with a grid size of X, Y, T for each cell. For each trajectory, we discretize them using $D(x, y, t) = (\lfloor x(i)/X \rfloor, \lfloor y(i)/Y \rfloor, \lfloor t(i)/T \rfloor) = ([x]_X, [y]_Y, [t]_T) = [\mathbf{X}]$. We have to ensure that the discrete resolution is fine enough to capture the motion and shape of the hazard but not too fine making it expensive. For instance, if we are searching for a pedestrian, we need a finer resolution than if we are searching for a car. In many of our evaluation, we found that $X = 1, Y = 1, T = 1$ works well. We investigate the trade-off of resolution, accuracy, and computation in Sec. 4.4.

4.3.2.2 Encode Open Space into Nodes

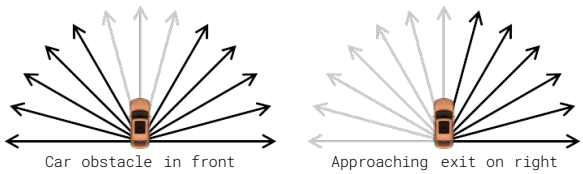
After discretizing the known trajectories, we create nodes in our graph that roughly corresponds to the open space around the vehicles. Each node specifically represents the proposition *if the unknown hazard was placed at this location, it will not collide with the car trajectories*. Therefore, given the shape of the unknown hazard (e.g., cars are roughly 4.3m x 1.3m [1]), we place them at each location with a specific orientation that is fixed for that portion in the road segment. This yields a 3D network where each node encodes the (x, y, t) of vehicle. An overhead view of an example 3D network is shown in Fig. 4.5.

4.3.2.3 Encode Mobility into Edges

Given the open space over time represented as a graph, we encode possible mobility of hazards through the open space as directed edges.

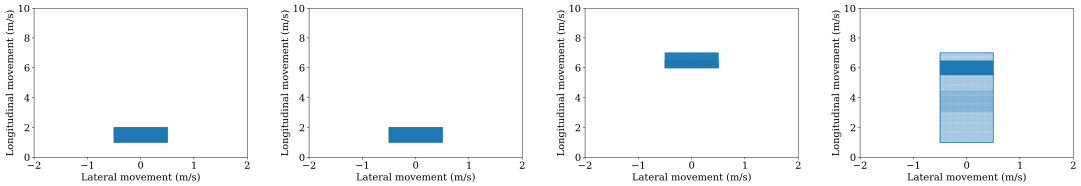
`Ubi` searches for hazards of varying mobility — stationary, slow- and fast-moving. For each type of hazard, we define a mobility model which specifies how much the

[1] `https://sumo.dlr.de/wiki/Vehicle_Type_Parameter_Defaults`



(a) Edge weights encoding the likelihood of transition of a reckless driver hazard. If the hazard is next to the edge of the road they are likely to continue straight.

hazard could have moved in one time step. The mobility models of a slow-moving pedestrian and a fast-moving bicyclist are shown in Fig. 4.8.



(a) Pedestrian fixed speed

(b) Pedestrian accelerating

(c) Biker fixed speed

(d) Biker accelerating

Figure 4.8: Mobility models for 4 different hazard types. Each scatter plot represents the full range of possible movement in one time step with the hazard starting at $(0, 0)$. We also compared these with a stationary obstacle that isn't shown here.

Possible Movement: Physical Constraints Starting at $T = 0$, we connect nodes to subsequent time steps based on the possible mobility shown in Fig. 4.8. The edges represent how the hazard could have moved through the time steps. In the case of stationary hazards all edges are simply connected to the same node in the next time step, i.e. all edges point directly up in our 3D directed graph. When connected nodes based on the mobility model, we have to ensure there is no collision with other known trajectories. This is enforced in two different conditions.

First, moving the unknown hazard from X to Y from $T0$ to $T1$ must not jump over any known trajectories of other vehicles in between the two time steps. Due to the discretization of time, this overlap of trajectories and hazards may occur in between the two time steps. We resolve this by checking the continuous-space location

of the cars in between $T0$ and $T1$ to ensure no collision happens *between* $T0$ and $T1$. (Alg. 4.4 #10-13) The second requirement is that no other known trajectories overlap with the unknown hazard between the two time steps due to their own movement. This occurs even if the unknown hazard doesn't move between the two time steps. A lack of motion may still be in violation of other vehicle motions (Alg. 4.4 #14-17). Resolving these two constraints gives us a full list of *possible* movements of the unknown hazard.

Probable Movement: Behavioral Constraints After connecting all possible transitions between open-space nodes, we are still left with an over-estimation of where the unknown hazard may be in our graph. This is due to the behavioral probabilities of how different entities behave in the real world. For example, a car is likely to stay within the lane or go in the right direction on a one-way road. Although it is physically possible to go in the opposite direction, this is highly unlikely. Fig. 4.7a shows example likelihoods of a reckless driver hazard given the position on the road. We set the edge weight to reflect the transition likelihood.

Specifically, we assign a higher transition likelihood based on two factors: (1) the prior probability distribution of hazard mobility (e.g., bicyclists are more likely to go straight) and (2) physical constraints on the road such as collision with other vehicles or obstacles on the road.

4.3.2.4 Find Shortest Path

After creating the graph and connecting the edges, we use Dijkstra's algorithm to find the shortest weighted path through the network. We further add the constraint that the path has to pass through the known sightings of the unknown vehicle. For each run of `Ubi` we have a set of possible paths through the network. The shortest weighted path is the path with the smallest sum of weights of its edges. This path

gives us the most likely path through the graph since the edge weights are inversely proportional to the likelihood of the transition. Choosing the shortest path gives a more accurate and more likely trajectory of the hazard.

4.3.3 Using graph search to warn drivers

`Ubi` uses the graph search described above in two different steps of its operation. The pseudocode of `Ubi` is shown in Fig. 4.3. In the first application of the graph search, `Ubi` uses multiple sightings of the hazard to determine the most likely hazard. In the extreme cases, if the hazards have very different mobility models, it may be possible to uniquely identify the hazard simply using the sighting. If that isn't possible, `Ubi` uses the edge weights from the graph search to determine the most likely hazard.

Next it uses the same graph search tool to forward simulate the most likely hazard. From the last known sighting, `Ubi` incrementally applies the hazard mobility model to the graph to get a set of possible location of the hazard. `Ubi` then averages the set of possible locations and uses that to determine the distance from the upcoming vehicle.

4.4 Evaluation

We first evaluate the accuracy of the warning detection system, and then the requirements and runtime of the graph search algorithm in isolation.

4.4.1 Evaluation of the Warning System

`Ubi` receives GPS locations from each driver and returns a warning if there is a hazard up ahead and the distance to the hazard. As each driver gets near the hazard, `Ubi` creates a new sighting and run the graph search to detect the type of hazard. For all GPS samples of cars before they encounter the hazard, `Ubi` uses the graph search

to simulate the location of the hazard from the last sighting and reports the average location from the approaching car.

We compare `Ubi`'s warnings against the correct expected behavior. Before the first driver encounters the hazard, the expected behavior is that there is no warning returned from `Ubi`, as it hasn't seen the hazard yet. After the first driver reports a sighting, the correct behavior is to warn upcoming drivers that there is a stationary obstacle detected at that fixed location. As more drivers report sightings of the hazard, the correct behavior is to report the exact type of hazard and the accurate distance to the hazard.

These experiments were done in simulation using three different hazard models — a stationary obstacle, a pedestrian (traveling up to $1.5m/s$) and biker (traveling up to $5.5m/s$) [99]. The speed and acceleration parameters of the models were chosen from the SUMO vehicle type parameters.[2] We simulated 2 different types of mobility for each hazard with all three hazards moving on the side of the road (1) going straight close to their max speed, and (2) accelerating up to their max speed. The mobility template for all types of hazards is shown in Fig. 4.8.

4.4.1.1 Warning Accuracy

Fig. 4.9 shows an example output of `Ubi` to warn a driver of an upcoming pedestrian on the side of the road. Initially, when `Ubi` collects only a few sightings of the hazard, it tends to incorrectly classify the hazard type. As it collects more sightings, it can more accurately detect the hazard type. The distance to the hazard also depends on the correctness of the classification. If it is estimated to be the wrong hazard type, the estimated distance suffers due to incorrect forward simulation of the hazard type from the last sighting.

We measured the accuracy of the hazard warning for fixed speed hazards. The

[2] `https://sumo.dlr.de/docs/Vehicle_Type_Parameter_Defaults.html`

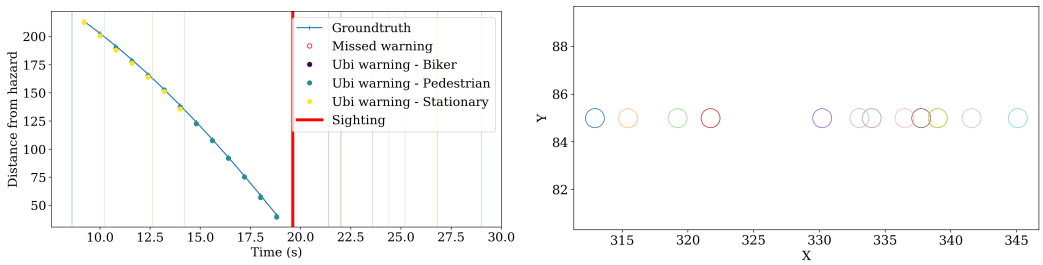
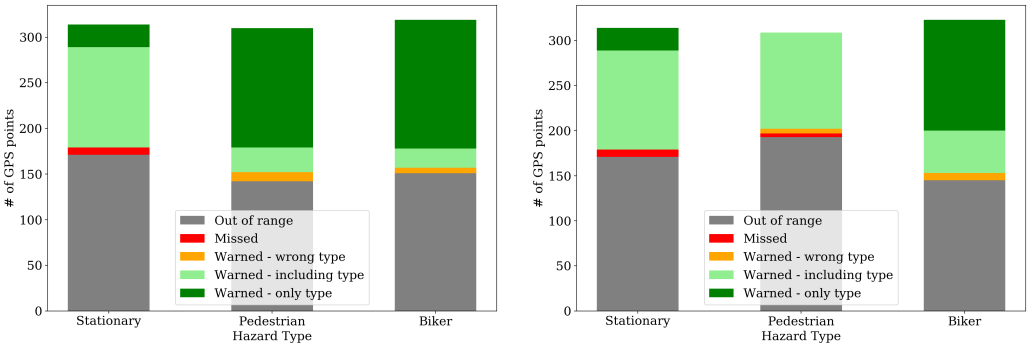


Figure 4.9: The left figure shows the distance from the hazard. Vertical lines show the time of the sightings of the hazard by all cars in the simulation, which is also reflected in the right figure to show the *location* where the sighting took place. The colored scatter points on the left figure represent `Ubi`'s response to the driver, showing both the classification and the distance to the hazard.



(a) `Ubi` warning accuracy for fixed-speed hazards.

(b) `Ubi` warning accuracy for accelerating hazards

results are shown in Fig. 4.10a. The hazard was placed near the middle of the track in the simulation, and therefore once vehicles went past the hazard, they were no longer notified, therefore around 50% of the GPS points receive no response from `Ubi`. For the remaining, we measured the number of missed (shown in red) warnings where `Ubi` didn't give any warning of an upcoming hazard, the number of warnings of a the wrong type (shown in orange), number of warnings that include the actual hazard and other hazards (light green) and warnings that only specify the current hazard (dark green).

In most instances, we found that `Ubi` rarely misses giving any sort of warning. For stationary hazards, `Ubi` failed to give a warning only 8 out of 143 requests (94.4% correct). For pedestrian and biker hazards, `Ubi` gave the wrong warning 10 out of

168 requests (94% correct) and 7 out of 168 requests (95.8% correct), respectively.

For the rest of the requests, `Ubi` either warned the driver of the correct hazard type (dark green) or returned a set of hazards which included the correct hazard type (light green). `Ubi` may return a set containing multiple possible hazards if the sightings are compatible with multiple hazard types. The pedestrian and biker hazard mobility models (shown in Fig. 4.8) are very distinct when they move at fixed speeds, and hence those two hazard types are not confused for another type. However, the stationary obstacle is often confused for a pedestrian obstacle since a pedestrian may move slowly near where the stationary obstacle was sighted. This results in a large number (110 out of 143) of warnings which include both pedestrian and stationary hazard warnings.

We further explored the relationship between overlapping hazard types and warnings by using an accelerating hazard model (second two figures in Fig. 4.8). The accuracy results are shown in Fig. 4.10b. We found that `Ubi` still rarely failed to give a warning of an upcoming mobile hazard. However, there is a much higher ambiguity about the hazard type. For an accelerating pedestrian hazard, `Ubi` often warned the driver that the upcoming hazard may be a pedestrian or another hazard type (107 out of 116 warnings). This ambiguity is also increased for the accelerating biker hazard but is not a pronounced because after collecting a few sightings, `Ubi` is able to distinguish the biker from the slow-moving pedestrian.

4.4.1.2 Density and GPS Noise

Next we studied the impact of car density and GPS noise in `Ubi` hazard warnings. First, we varied the density of cars on the road with the stationary, pedestrian or biker hazard. The impact on detection accuracy and distance error is shown in Fig. 4.11.

For very low-density traffic, `Ubi` receives fewer sightings of the hazard. This results in more misclassifications of the hazard. We found that for very low density, only

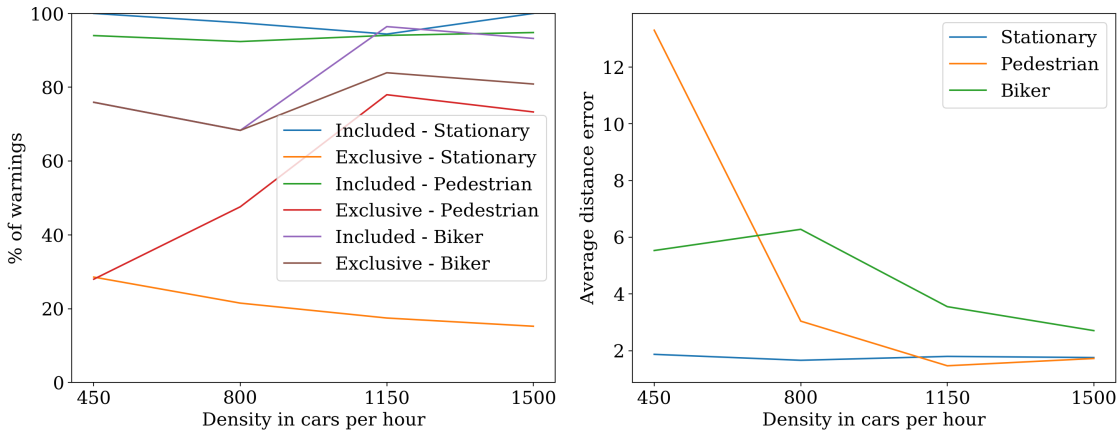


Figure 4.11: Accuracy and distance errors for varying density of cars. "Included" results show the responses which included multiple hazards, including the correct hazard. "Exclusive" results show the responses which only warn the driver about the correct hazard.

$\approx 30\%$ of warnings for a pedestrian hazard classify the hazard as a pedestrian, but $\approx 95\%$ of warnings report a set of hazards including the pedestrian and others. This is also reflected in the distance reported by the `Ubi` warning. For very low density, the pedestrian hazard average error is $\approx 13m$, which drops to $\approx 2m$ for higher density traffic. This occurs because `Ubi` determines that with few sightings, the hazard could be multiple different options. Forward simulation with the wrong hazard type results in a wrong estimate of the actual position of the hazard.

Next we studied the effect of GPS noise in `Ubi` classification and distance reports. To measure the impact of GPS noise, we down-sampled the simulated trajectory to $1Hz$, and added random distributed noise scaled up to four different values. The results are shown in Fig. 4.12.

In general, we found that `Ubi` warning accuracy and distance is unaffected by GPS noise. However, if we have high GPS noise, `Ubi` is unable to correctly warn drivers of stationary obstacles. This occurs because the graph search algorithm connects nodes from $(t0, x0, y0)$ to $(t1, x1, t1)$ only if it is possible for the hazard to travel from $(x0, y0)$ to $(x1, y1)$ in one time step without colliding with any known trajectories.

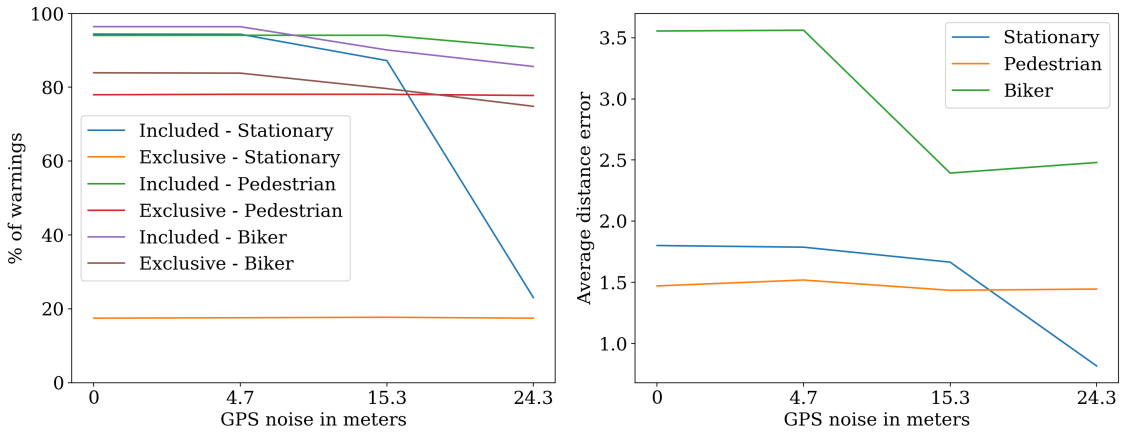


Figure 4.12: Accuracy and distance errors for varying GPS noise. The different noise values were chosen from [52].

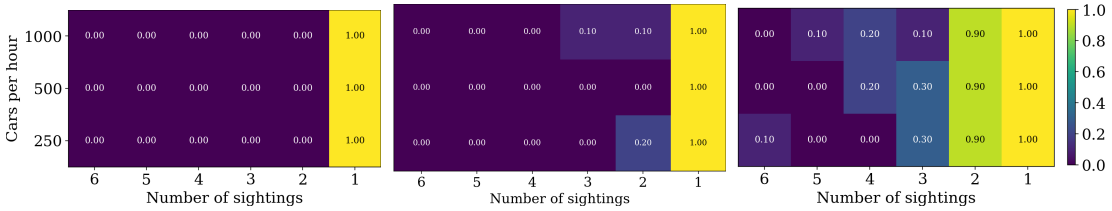
As we increase the GPS noise, the "known trajectories" appear more noisy and cause more collisions with the hazard paths. For mobile hazards, such as a pedestrian or a bicyclist, this algorithm is able to find a path through the noisy trajectories and still find a path which connects the hazard sightings, but for stationary obstacles it is unable to do this. This makes the stationary obstacle detection brittle to high GPS noise. If high GPS noise is detected, we can switch to simpler methods for tracking stationary obstacles such as those used by existing navigation apps (e.g., Google Maps).

4.4.2 Evaluation of Graph Search

Next we evaluated the accuracy of the graph search in detecting stationary, slow- and fast-moving hazards. The graph search algorithm is used twice inside the `Ubi` algorithm. First, it is used to detect the most likely hazard given the sightings. Next, it is used to predict the location of the hazard and warn future drivers.

We measured the number of sightings required before we can confidently label each hazard type. We evaluated this using Movsim [99] to create trajectories from vehicles driving around the hazard. We drew sightings around the hazard and ran applied the

search to evaluate whether the hazard was detected under different conditions.



(a) Stationary obstacle hazard FPR (b) Slow moving hazard FPR (c) Reckless driver hazard FPR

Figure 4.13: Heatmap of false positive rate for each hazard type. The number of sightings varied from 1–6 and the density of cars changes from light (250 cars per hour), medium (500 cars per hour) and dense (1000 cars per hour).

If the dataset contains a hazard, this is considered a positive test case. If `Ubi` finds a hazard in this dataset, that is a *true positive* and if it fails to find a hazard, that is a *false negative*. We also drew random sightings where there is no hazard to create a negative test case. If `Ubi` finds a hazard in this dataset, that is a *false positive* and if it doesn't find a hazard, that is a *true negative*.

4.4.2.1 Simulation Dataset

Using Movsim [99], we simulated 3 different hazards — a stationary stopped car, a slow-moving bicyclist, and a fast-moving reckless driver. The vehicle parameters are defined in `https://sumo.dlr.de/docs/Vehicle_Type_Parameter_Defaults.html`. For each hazard, we varied the density of the other vehicles on the road and the number of sightings where the hazard was spotted by neighboring cars.

The density is defined in terms of number of cars per lane per hour. We simulated for 250 cars per hour ('light'), 500 ('medium') and 1000 ('dense'). The road is 500m long and cars were randomly generated to travel through the road segment avoiding the hazard. The cars use the MOBIL lane-changing model to avoid hazards and change lanes among other vehicles [55].

We created 10 different random scenarios and ran the simulation for 30 seconds. We first evaluated the effect of density and number of sightings in the false positive

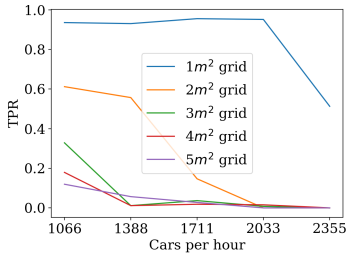
rate (Sec. 4.4.2.2). In order to keep the FPR low, `Ubi` requires a certain density of cars and reported sightings. In all cases in our simulation, `Ubi` was able to correctly detect the hazard if it exists in the scenario (i.e., TPR = 1 in all cases). However, if we decrease the distance between neighboring cars or if the discretization of the graph is very coarse-grained, the TPR suffers. We explore this in Sec. 4.4.2.3. Changing the discretization resolution affects the run-time of the algorithm. The run-time is measured in Sec. 4.4.2.4

4.4.2.2 False Positive Rate

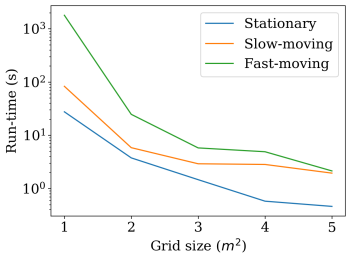
The false positive rate depends on three factors — the mobility of the hazard, the number of sightings, and the density of the known cars. Fig. 4.13 shows three heatmaps of the FPR while varying the number of sightings and the density of the cars. We can see the pattern for different types of hazards with different mobility.

If there is low mobility (e.g., a stationary hazard), then even two false sighting reports are enough to rule out the hazard. As seen in the heatmap, we have no false positives for stationary obstacles if we require 2 or more sightings. However, if the hazard has higher mobility, then multiple false sightings might still be a valid way for the hazard to move, and therefore a false positive is reported. However, if we require multiple sightings (e.g., 3 sightings for slow-moving and 5 for fast-moving) the false positives drop.

We can use this as a guideline to adjust the false positive rate. If we are using `Ubi` to find a stationary hazard, we require 2 or more sightings. If it is a highly mobile hazard, then we must record multiple sightings before we use `Ubi` to find the hazard in order to avoid false positives. The exact number of sightings depends on the density of the cars in that region and the type of hazard. The trade-off here is that to require more sightings means it takes longer to detect the hazard after it appears on the road.



(a) True positive rate. The density varies from 1000–2000 cars per hour and the discrete grid size changes from $1m^2$–$5m^2$.



(b) Runtime of `Ubi`. We measured the run time for different hazards and different resolutions $1m^2$–$5m^2$.

4.4.2.3 Effect of discrete grid resolution

In the simulations above, the true positive rate (TPR) was always 100%. The TPR (defined as $\frac{TP}{TP+FN}$) is affected by the density of the cars and the resolution for the graph. We varied these two parameters and measured the true positive rate in simulation, shown in Fig. 4.14a. As the resolution becomes more coarse-grained, the TPR tends to be worse. This is further exacerbated with a high density of cars on the road.

This pattern occurs because a coarse-grained resolution is unable to represent cars that are close to each other without collision. `Ubi`'s graph search models the physical possible movement of the hazard in the discrete realm (details in Sec. 4.3.2.3). With a coarser resolution, there are no possible movements to match the sightings of the vehicle because even small movements result in collision with nearby vehicles.

With a grid size of $1m^2$, the TPR is very high throughout our analysis and only starts to drop as we reach 2000+ cars per hour. To accommodate higher density road segments, we can increase the resolution of the graph, but this results in a higher run time. The run time is further explored in the next section.

4.4.2.4 Run-time measurement

The run time of the graph search depends on multiple factors. Mainly, it is affected by the resolution and the number of steps the hazard can take at each time step (Variable $\mathcal{H}$ in Alg. 4.4 #8). For a stationary hazard, it only has one possible movement from one time step to the next. For a mobile hazard, the faster it can move, the more possible steps it can take from one time step to the next.

The number of nodes in the graph is upper-bounded by $M \times N \times T$, where MN is the number of cells needed to represent the discrete grid and T is the number of discrete time steps. The number of edges depends on how many possible steps each hazard can take. The upper bound for the number of steps is $M \times N \times T \times H$ where H is the number of possible movements the hazard can take at each time step.

Once we find the possible set of movements, we use Dijkstra's algorithm to find the shortest path for each pair of possible nodes from the start to finish. The run time of Dijkstra's algorithm is upper bounded by $O(|E| + |V|log|V|)$. If there are En possible entry nodes and Ex possible exit nodes, we repeat the graph search $En \times Ex$ times.

The measured run time for each type of hazard and different resolution is shown in Fig. 4.14b. As we increased the grid size from 1 to 5 m^2, there was a sharp decrease in run time for each type of hazard. For a stationary hazard, the line of best fit has a -5.8 negative slope, and the slow-moving hazard has -16.6 and the fast-moving hazard has -284. The fast-moving hazard is better fit with an exponential curve with a -4 exponent. The biggest factor influencing the run time is the number of possible movements by each of the hazard. The fast-moving hazard has the most number of possible movements at each time step. This results in many more edges in the network, which requires more computation to search through the network to find paths which contain the hazard.

These results were obtained on a 2.1 Ghz Intel Xeon E5 processor. The current

implementation only uses one core of the CPU and can be improved using multiple cores. The algorithm is inherently parallelizable since we operate on multiple independent nodes at the same time from one time step to the next. We can use existing parallel processing frameworks such as MapReduce[3] to process the nodes in a distributed manner.

The output of the graph search can be used in a real-time or a semi-real-time way depending on the underlying application. If it is a dangerous hazard (e.g., a reckless driver) or is in a highly populated area, then it is important to report the results to upcoming drivers in real-time. If it is a non-time-critical hazard such as a pothole, it is sufficient to detect and report this at a later time.

4.5 Discussion and Future Work

Possible deployment strategies Existing navigation apps such as Google Maps or Waze already notify drivers of stationary obstacles in the path of traffic. For example, through crowd-sourced reports, Waze can report if there is a stopped car on the road. Using `Ubi` they can extend their coverage to include mobile hazards as well. If a few drivers report the presence of a pedestrian on the side of the road, using `Ubi` these navigation applications can warn drivers ahead of time. Alternatively, we can also build `Ubi` as a standalone application which accepts reports of hazard sightings. It can collect this on a central cloud and distribute it back to drivers with `Ubi` installed on their phones if there is a hazard nearby.

Security and Privacy implications The downside of distributing `Ubi` as a standalone application is that it shares the user's location across multiple service providers, thereby putting their location privacy at risk of exposure. It is, therefore, advisable that existing widely-deployed navigation service providers adopt `Ubi` into their sys-

[3] `https://en.wikipedia.org/wiki/MapReduce`

tem. If it must be deployed as a standalone application, we can further explore adding differentially-private noise into the location acquired from each driver in order to protect the user's privacy but at the same time accurately determine the location of the hazard. This is part of our future work.

Another direction of future research is whether it is possible for a malicious actor to poison the data that feeds into `Ubi` and falsely report a hazard where there is none. An attacker who poisons the system can then influence routes suggested by navigation applications and create a free route for their own travel. This is partly mitigated by design because we require that multiple users report sightings before we flag a hazard (see Sec. 4.4.2.2). In future, we will test the robustness of `Ubi` to such data poisoning attacks.

4.6 Conclusion

Traditional methods to detect dangerous driving hazards requires sophisticated vision-based sensors or are restricted to stationary obstacles. We have proposed a novel method and system, called `Ubi`, which only uses GPS trajectories from *nearby vehicles* and reported sightings to detect and track the presence of stationary and mobile hazardous object. Our system doesn't require direct GPS traces from the hazard itself, and works for detecting stationary as well as mobile hazards.

`Ubi` is shown to be able to detect hazards with very high accuracy ($> 94\%$) and with low distance error ($< 3.5m$). `Ubi` is also resilient to GPS noise up to $25m$ and can be implemented efficiently using a directed graph and therefore leverages existing graph search algorithms such as Dijkstra's shortest path algorithm.

`Ubi` can be integrated into existing navigation applications. By integrating stationary and mobile hazard detection into their systems, navigation applications can consider the safety of the route to the destination in addition to the traditional considerations of shortest travel time.

CHAPTER V

CAB: On-demand Vehicular Data Collection Builder

5.1 Introduction

Vehicular research and app development spans a wide range of interests and requirements such as driver monitoring [87, 54, 96], road hazards detection [32, 3, 103], and vehicular security [72, 49, 68]. There are also numerous consumer-focused vehicular apps such as driver status monitoring or driving score evaluation [101, 6].

However, there is a high barrier of entry to engage in automotive research or app development. A researcher or an app developer needs to understand how to collect, process and interpret driving behavior from low-level sensor data, set up all the communication channels to collect data and interpret them for research or consumer-facing app development purposes. This expensive startup barrier deters researchers or consumers from venturing into this field. This is a detriment to rapid prototyping of vehicular apps, and therefore slows down consumer-facing vehicular app development.

To remove or mitigate this barrier, we need an easily configurable vehicular app builder which allows users to build and launch data-collection campaigns without in-depth technical knowledge, and facilitate rapid prototyping of new vehicular data-collection ideas/apps.

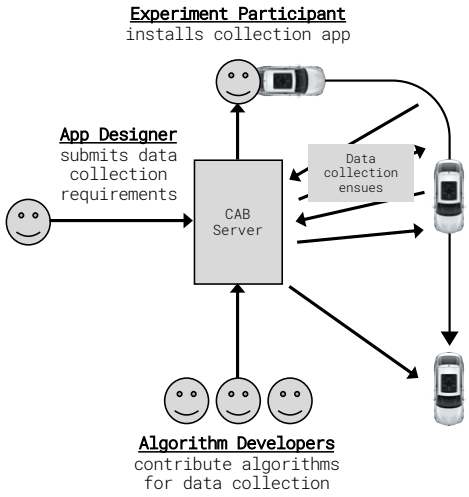


Figure 5.1: Three parties involved in CAB. *Algorithm developers* contribute code to the CAB repository. *App designers*, which could be researchers or app developers, submit high-level requirements. The CAB server takes the available algorithms and automatically compiles a data-collection app. The *Experiment participant* joins the data collection and installs the data-collection app

5.1.1 State of the Art

Consumer-facing data-collection apps either provide high-level statistics which cannot be re-configured for different purposes [6, 101] or provide raw low-level information which isn't useful for consumers without sophisticated technical knowledge [36, 44]. Furthermore, academic researchers often build a new data-collection platform for *each* investigation, which results in time-consuming and often duplicated effort [42, 17, 95]. Existing data-collection platforms lack the flexibility to be used across multiple studies [95, 64, 8, 82, 48]. They often collect excessive and potentially irrelevant data or are built for a specific purpose and are not customizable for different data-collection needs.

Due to the lack of re-configurable data-collection platforms, consumers and researchers are deterred from entering into this field of automotive data collection and research.

5.1.2 Proposed System: CAB

We propose a novel system called `CAB` (Car App Builder) to meet the above-mentioned need. `CAB` is a configurable data-collection app builder that is useful for non-technical users as well as app developers who want to rapidly prototype their vehicular app. The primary goal of `CAB` is to reduce the effort needed to build data-collection apps and engage in vehicular data collection and research.

`CAB` uses a simple user interface to accept the requirements of the data-collection app and automatically compiles an app to perform the data collection, creates a sandboxed server to accept and communicate data, and a website where users can sign up and configure their data-collection app. Collectively, we call the app, server and website as the "data-collection platform." An app developer can further take the auto-generated data-collection platform and prototype their own ideas on top of it, reducing the overhead needed to test their ideas.

Fig. 5.1 shows the system components of `CAB`. There are three main parties who interact with `CAB`. A *data-collection app designer* submits the high-level data-collection requirements to the `CAB` server. The server proposes a data-collection strategy which includes a set of algorithms running on the user's phone, server-side components and website components which meets these requirements. The second party, the *algorithm developer*, contributes individual algorithms to the `CAB` repository. They provide the source code for each algorithm and specify the required information and the produced output information. The submitted algorithms are compiled as needed for each data-collection platform. Finally, the third party, the *data-collection participant*, installs the compiled app created by `CAB`.

5.1.3 Key Technical Details

`CAB` has two main technical features which enable its use for diverse data-collection needs. Fig. 5.2 shows the system architecture of `CAB`.

'Information' ⇔ 'Algorithm' Interface. At the heart of `CAB` is the language-agnostic definition of *information* and *algorithm*. An algorithm is a module in the `CAB` repository which defines a set of information types it requires and the information type that it outputs. This interface between algorithms and information enables the following features:

- *Automatic dependency resolution.* `CAB` automatically crawls the graph of information and algorithms to find the set of dependencies needed to meet the data-collection requirements.

- *Distributed development.* Multiple developers can define algorithms and contribute to the shared `CAB` repository. The algorithms are interconnected using this well-defined algorithm-information interface.

- *Redundant algorithms.* Multiple algorithms may produce the same information using different means. For example, if a user doesn't have an OBD dongle, a different algorithm may produce the same information by estimating the required information using only phone sensors. This increases the reach and deployability of the apps compiled using `CAB`.

Automatic Compilation into Sandboxed Environment. Once an app designer submits a new data-collection requirement `CAB` automatically identifies the set of dependencies and compiles a custom data-collection platform.

It creates an isolated environment for this data-collection campaign and links the relevant data-collection algorithms from the repository. This includes Android-based components, server-side scripts, and website user-interface components which are necessary for this data collection platform. `CAB` launches a separate *linking server* for each data collection campaign. The linking server allows different algorithms to work together seamlessly to receive and upload data that is required for their operation.

5.1.4 Results

Our evaluation shows that CAB can be used to build diverse data-collection apps, with diverse requirements. We implemented 3 different data-collection apps in CAB to show its versatility — fuel consumption data collection, vehicle sensor estimation, and crowd-sourced obstacle detection. In each scenario, CAB reduced the effort needed to program and build the data collection platform. We demonstrate that with 18 different modules which we implemented in this work, we can create 8,100 unique data collection applications, which is a 450× return in developer effort. Next, we did a usability study with domain-expert and non-expert researchers. The participants were likely to use CAB for their own data collection needs and find that it reduces the effort needed to launch a data collection campaign as well as develop novel algorithms.

5.1.5 Contributions

This chapter makes the following contributions.

- An open-source data-collection builder, CAB, which accepts high-level data collection requirements and automatically builds custom data collection platforms.
- A collection of 18 algorithms developed using CAB, which can be assembled to create 8,100 unique data collection platforms.
- A user interface for researchers to specify and launch their own data collection platforms. A demo of the interface is found on YouTube[1]
- Use of CAB to build 3 vehicular data-collection platforms from related literature.

[1] https://www.youtube.com/playlist?list=PLuEbkT_dQmRIn7JOoDf2JHKQT0n_xfcFc

5.2 Data-Collection Requirements

Vehicular data collection research spans multiple areas, a subset of which are shown in Table 5.1.

Most of the studies involving data collection transform low-level sensor data (accessed via a smartphone or an OBD dongle) into higher-level information. For example, Hong *et al.* [45] transformed smartphone sensors into a dangerous driving classifier. Chen *et al.* [17] transformed IMU data into lane-changing and turning behavior. Many other studies in dangerous driving detection also transformed low-level sensor data in a similar way [113, 17, 60, 61, 45, 18].

Environment-modeling studies require a server-side component to collect data from multiple vehicles. For instance, Jiang *et al.* [52] collected pothole information across multiple drivers in a central cloud. Wang *et al.* [106] developed CrowdAtlas, a system which combines GPS traces multiple drivers to automatically update maps with driving data. Many of these systems also require training machine learning models which is better suited to a resource-heavy server. For instance, Zheng *et al.* [118] trained a neural network to predict air quality in cities, as task which is best suited to a resource-rich machine rather than directly on the smartphone.

Finally, many applications require user input to collect information which cannot be learned through sensors. For example, Hong *et al.* [45] required questionnaires to learn about the driver's history of accidents and build a dangerous driving profile model. Chen and Shin [16] used crowd-sourced reports to collect ground truth information about unprotected left turns. All these methods require a versatile user interface to collect information from users. Many consumer applications such as Google Maps [41] and Waze [108] require user input to collect the presence of speed traps of obstacles on the road.

Driver-modeling	Environment-modeling	Vehicle-modeling
Drowsiness detection [87, 112]	Pothole detection [32, 92, 91]	CO2 emission analysis [22]
Driver fingerprinting [28, 15, 60, 117]	Digital maps [16, 23, 3, 47]	Vehicle turning detection [17, 105, 64]
Dangerous driving [45, 67]	City-wide monitoring [40, 118, 95]	Speed estimation [114, 51]

Table 5.1: Selection of studies which require data collection categorized by whether the focus of the study is to model the driver, the environment or the vehicle.

5.2.1 Design Goals

Based on the requirements of vehicular studies involving data collection, we consolidated a list of design goals to be met by `CAB`. First, we have a list of *functional* design goals that must be supported by `CAB` as follows:

1. **Low-level sensor access translated into high-level information.** `CAB` must be able to transform low-level sensor data to high-level information. Sensor data must be accessed from the phone, OBD or other supporting devices. `CAB` must be easily extensible to support new high-level information as they are discovered in the research community.

2. **Server-side processing.** `CAB` must provide seamless communication of data between the smartphone and server-side scripts. Server-side processing is required for heavy-duty computation when data must be shared across multiple drivers.

3. **User interaction.** `CAB` must provide the ability to interact with the user to acquire user-supplied information such as their reports about their mood. The user interaction platform should support many forms of intuitive interaction such as text-based forms and buttons.

In addition to the functional requirements, `CAB`'s main goal is to reduce the effort required by app designers and algorithm developers. To this end, we have two additional design goals.

(4) **Effort reduction for app designers.** `CAB` must provide a simple interface where app designers can submit their requirements and build data-collection

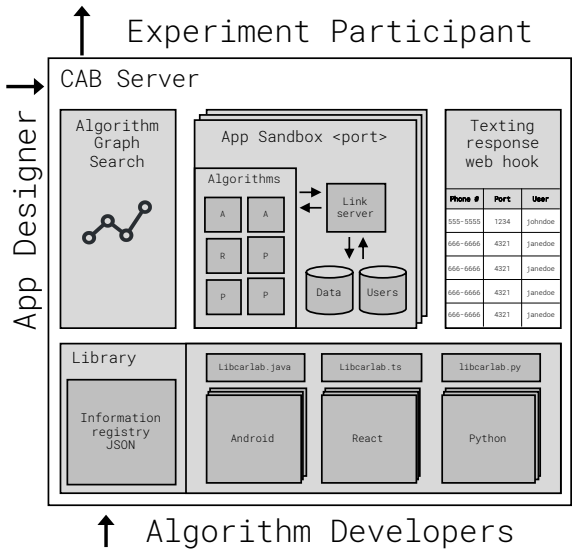


Figure 5.2: CAB system architecture.

platforms. CAB must simplify the data-collection process by presenting the collected data and interfacing with participants in data-collection experiments.

(5) **Effort reduction for algorithm developers.** CAB must provide a simple programmatic API and development environment where developers can implement their own algorithm and integrate with the CAB ecosystem.

5.3 System

CAB is a reconfigurable on-demand data-collection builder useful for data-collection campaigns and rapid prototyping of data-collection apps. It accepts high-level requirements from users and automatically assembles a data-collection platform with all the necessary infrastructure to carry out complex data-collection campaigns across multiple devices. CAB can also be easily extended to add more functionality through a shared central code repository.

In what follows, we describe the system components of CAB which make it possible to automatically build data-collection campaigns to meet different requirements. The discussion is organized around the three main parties involved in CAB, shown

Platform	Language	User interaction	% phones
Android	Java	Low-level phone sensors	39.8% [2]
React	TypeScript	Web interface	77% [3]
Python scripts	Python3.7	Texts or phone calls	95% [4]

Table 5.2: Algorithms can be developed for 3 different platforms. All algorithms share the common interface so data can be communicated across each other seamlessly. Each algorithm is implemented in a language that has typing support to enforce the proper formatting of information.

in Fig. 5.1. We describe the system architecture pertaining to each party and their interaction with CAB. Fig. 5.2 shows the overall system architecture. The high-level specification files and their relationships are summarized in Fig. 5.3 and detailed in their respective sections below.

5.3.1 Algorithm Developer

CAB builds data collection apps by sampling from a library of algorithms contributed to the shared repository. An "algorithm" is a self-contained module defined in one of the three supported languages shown in Table 5.2. The three languages meet the three functional design requirements. Algorithms implemented in Android are best suited for low-level sensor access (design goal #1). Python-based algorithms are best suited for server-side processing (design goal #2) and React-based algorithms can define complex user interfaces (design goal #3).

Each algorithm defines an interface in a JSON file which defines all required *input* information and the *output* information. An example interface file is shown in Fig. 5.10a. All algorithms share a common language of *information*, including the information name, a shared understanding of its meaning, and a data type, defined in a language-independent JSON file shown in Fig. 5.10a.

CAB provides a tool to auto-generate the algorithm stubs given an algorithm definition of input and output. The auto generation tool ensures each algorithm conforms to the interface required to communicate with other algorithms and helps jumpstart the development process without any tedious setup. Auto-generation expedites de-

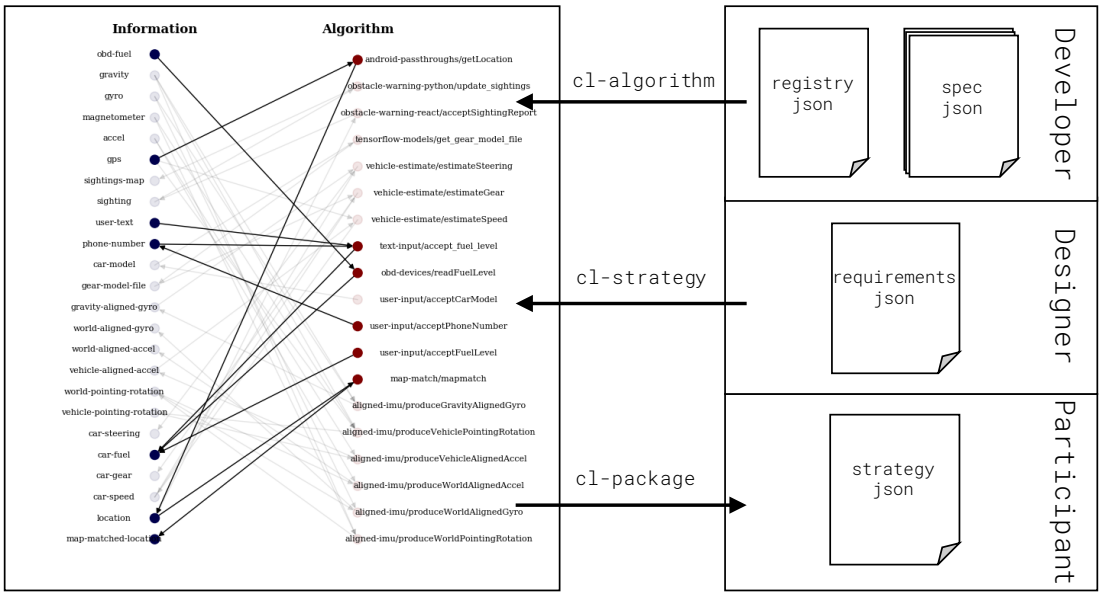


Figure 5.3: `CAB` uses several high-level configuration files to automatically generate data-collection platforms. Examples of all configuration files are shown in the Appendix. It uses the script `cl-algorithm` to convert the `spec.json` file into algorithm stubs to be filled in by the algorithm developer. Using `cl-strategy`, it converts the high-level requirements input by the app designer (`requirements.json`) to get a detailed strategy file which lists all algorithms and dependencies (`strategy.json`) to be included in the data collection app. Finally, it uses `cl-package` to compile the data-collection app, build the data-collection website, and initialize a virtual machine for each data-collection platform.

velopment of new algorithms for the CAB repository, thereby meeting design goal #5.

The auto-generation tool can be invoked using `cl-algorithm <specfile.json>`. This command creates all the necessary classes to interface with the CAB library and includes the required library files, which we implemented separately for each platform. It also creates function headers which accept input information as function parameters and return the output information (abstract classes in Java, Components with props in React, and Classes with functions in Python). The data type for the function header is loaded from the registry JSON file. Once created, the algorithm developer can simply fill in the function stub to transform the input information and return the output information. The CAB library ensures that the required input information is collected and fed into the function and the output return value is properly sent to the CAB server and multiplexed to the appropriate algorithm which requires that information. An example auto-generated function stub for the above algorithm spec is shown in Fig. 5.4. The algorithm class also explicitly defines the list of functions and their input and output information types. The explicit definition is used later when automatically compiling algorithms into a data-collection platform.

5.3.2 App Designer

An app designer submits a high-level requirement to CAB and decides on a set of algorithms which may span multiple platforms to meet the data-collection requirements. CAB uses the requirement file to assemble all required algorithms into a package for each platform and runs them simultaneously during the data-collection process. Each aspect of this process is described in detail below.

```java
package cab.aligned_imu;
import android.renderscript.Float3;

public class Algorithm extends AlgorithmBase {

  @Override
  public Float[] produceVehiclePointingRotation (
    Float3 m, Float3 gps, Float3 g) {
      // Write code here
    }

  @Override
  public Float3 produceVehicleAlignedAccel (
    Float3 accel, Float [] rotation) {
      // Write code here
    }
}
```

Figure 5.4: Java algorithm implementation stub auto generated using `CAB`. A similar function is auto generated for Python and React-based algorithms.

5.3.2.1 Refining the data-collection strategy

The app designer submits the basic requirements, via a `requirements.json` file — an example shown in Appendix Fig. 5.10c — to `CAB`. The requirements file specifies the list of information that is required, the list of forbidden sensors (e.g., GPS), and the set of platforms for this data-collection requirement. `CAB` uses the internal representation of the available algorithms and their information to find the set of algorithms which output the required information. `CAB` furthermore crawls all the input requirements of the algorithms to resolve their dependencies. The user can further specify which algorithm to use if there is a choice. If no choice is given, we choose all algorithms for added redundancy. `CAB` outputs the final set of algorithms needed to satisfy the input requirements and saves this to a `strategy.json` file, example shown in Appendix Fig. 5.10d.

The graph used to resolve algorithmic dependencies is shown in the middle in Fig. 5.3. This graph is automatically generated with all the information types, and the algorithm specification files submitted by developers. As more developers contribute more algorithms and refine `registry.json`, the graph is automatically up-

dated thereby providing more sophisticated algorithms to app designers.

We have built a simple web UI to generate the requirements file so that a non-technical users can also generate a valid file without writing JSON specifications. We will give an example this web UI in the evaluation section. By only requiring the high-level specification file and a user-friendly web UI, we meet design goal #4. This significantly reduces the effort of an app designer to create and launch their own data-collection platform.

5.3.2.2 Compiling required algorithms

Once the app designer confirms the data-collection strategy, `CAB` uses that to compile the relevant algorithms together. We created a script (`cl-package strategy.json`) which translates `strategy.json` into a standalone project for each platform. All Android algorithms are linked into an Android project which is compiled into an APK. All Python algorithms are loaded as separate modules and invoked by a Python script running on the `CAB` server. The React algorithms are loaded as components into a website also running on the `CAB` server.

The linking process is different for each platform. For Android algorithms, algorithms are included as separate Gradle modules. Their paths and module names are defined in the `settings.gradle` file. They are also included in the projects `build.gradle` file and are therefore compiled during run-time. All further dependencies of the algorithms are defined in their own project and recursively resolved during compile time. Similarly, Python modules are defined as Python libraries and are symbolically linked into the project folder for this data-collection platform. React algorithms are compiled as separate Node modules and are linked together using the node package manager.

Once the algorithm dependencies are loaded into the project folder, the `cl-package` script statically creates the code needed to load the algorithms during run-time and

```
package cab.packaged;
import java.util.Arrays;
import edu.carlab.Registry;
import edu.carlab.Strategy;

public class PackageStrategy extends Strategy {
  public PackageStrategy () {
    // List of algorithm objects
    List<Algorithm> loadedAlgorithms =
    Arrays.asList(
    carlab.android_passthroughs.Algorithm.class,
    carlab.obd_devices.Algorithm.class);

    // An "AlgorithmFunction" is a custom class that
    // defines all input and output Information
    List<AlgorithmFunction> loadedFunctions =
    Arrays.asList(
    carlab.android_passthroughs.Algorithm.getLocation,
    carlab.obd_devices.Algorithm.readFuelLevel);

    // Information to be uploaded to CL server
    List<Information> saveInformation =
    Arrays.asList(
    Registry.Location,
    Registry.CarFuel);
  }
}
```

Figure 5.5: Auto-generated code connecting different algorithms loaded for Android. This file is auto-generated and doesn't need to be edited by the app designer.

invoke the callback functions if necessary. The example statically generated code for Android is shown in Fig. 5.5. A similar strategy file is generated for Python and React (TypeScript). A technical developer can modify the auto-generated code to add their own functionality to the packaged data-collection application. This enables rapid prototyping, contributing towards design goal #5.

5.3.2.3 Routing information across algorithms

One of the primary services provided by `CAB` is the seamless communication of data across different algorithms. This communication happens both inside the platform and across platforms. For Android algorithms which require low-level sensor data (such as GPS or IMU sensors), `CAB` creates the necessary listeners to read that sensor

Endpoints	Parameters	Return
`POST /createuser`	username, password	Success message
`GET /login`	username, password	Session ID used for all future transactions
`POST /add`	session, information, file or value	Success message
`GET /list`	session, information, sincetime	List of data points
`GET /latest`	session, information	Latest data point

Table 5.3: API endpoints exposed by the linking server. A new linking server is started for each data collection application. All platforms (Android, React and Python scripts) make HTTP calls to the linking server to communicate information to other platforms.

information. The data is then passed into the algorithm's callback function. If the information is required by other algorithms running on a different platform, then it is packaged and uploaded to a linking server. Each data-collection campaign contains a separate linking server which is responsible for marshaling between algorithms, for saving data and user management. The linking server exposes RESTful API endpoints for uploading and downloading relevant data, shown in Table 5.3.

5.3.2.4 Isolating application in a virtual machine

`CAB` spawns a new virtual machine instance to hold platform-specific files. The virtual machine contains the linking server responsible for sharing information between algorithms within each platform. The linking server runs on a fixed port inside the virtual machine and is forwarded to a different uniquely chosen port in the host machine. The host machine's port is exposed to the outside so that Android-based algorithms can interface with the appropriate linking server. This also allows us to run multiple platforms on the same server. The virtual machine also contains all the data and maintains a database of users for that data-collection app.

5.3.3 Experiment Participant

A participant registers for the data-collection campaign through the linking server with a username and password. This adds a new entry to the users database for that data-collection campaign. Once they create an account, they are presented with a URL to access the React-based algorithms and a link to download the APK containing

the Android-based algorithms. They log in to the website and on their Android app using the same login information from before.

If the data-collection campaign includes an algorithm which accepts text messages from the user, the linking server also registers the user's phone number with the *texting server* database. The texting server is a public-facing server that runs on on a fixed port on the host machine. It records the port number of the linking server, the session ID and the phone number of the user. We use an external texting service called *Twilio*[5] to send and receive texts. Whenever the user responds to a text, it is received by the Twilio service which then invokes a web hook running on the texting server with the message information. The texting server looks up the appropriate linking server which registered this phone number and forwards the message.

5.4 Implementation

The `CAB` server, texting server, and linking servers were implemented in 385 lines of Ruby on Rails. The Android `CAB` library was written in 3101 lines of Java and 246 lines of XML. The Python library was written in 139 lines of Python3.7 and the React library was written in 109 lines of TypeScript. Additionally, each project is wrapped in a platform-specific code which includes all developer-specific algorithms (output from `cl-package`), example shown in Fig. 5.5. The Android-package is written in 475 lines of Java, Python-package is 83 lines and the React-package is 313 lines of TypeScript. Finally, `CAB` auto-generates the base code for each algorithm to simplify the development process (output from `cl-algorithm`), example shown in Fig. 5.4. `CAB` creates 47 lines of Java, 29 lines of Python and 20 lines of TypeScript for each algorithm. In total, all aspects of our system (not including the algorithm libraries) amounts to 385 lines of Ruby, 3,623 lines of Java, 251 lines of Python and 442 lines of TypeScript. The Android libraries are responsible for interfacing with the low-level

[5] `https://www.twilio.com/`

Android sensors and is therefore the most complex part of our system.

We used Android SDK 28 in Java and Python 3.7. The server-side components for each data collection platform is run inside a Docker container. Node modules are maintained and packaged using the Node Package Manager version 6.13. We implemented 19 algorithms and defined 18 new information types.

5.5 Demonstrative Applications

We implemented three different apps using `CAB`, to demonstrate its expressiveness and ability to meet diverse requirements. Each app highlights a different aspect of `CAB`. For all apps we highlight the reduced effort required by the app designer or the algorithm developer.

5.5.1 Case Study 1 – GreenGPS

GreenGPS [40] is a study into the fuel efficiency of various road segments. We implemented a data-collection app which collects the fuel information and the road name and location of the experiment participants. Fig. 5.6 shows the workflow from the app designer perspective. First, they specify the information types they wish to collect from the web interface. This is converted into a `requirements.json` file and `CAB` prepares a data-collection strategy using the repository of available algorithms. This is shown in the graph in the bottom right of the figure.

Note that `CAB` automatically compiles all possible algorithms which can produce the necessary information. In this example, there are three possible algorithms which output the *car-fuel* information. The user can either (1) input the fuel level through a web interface (opened on their phone or on a laptop), (2) read the fuel level through an OBD dongle or (3) they can text the fuel level to the `CAB` service. For all data-collection paths, `CAB` includes the necessary dependencies into the platform. By automatically compiling all possible methods into the data-collection platform, `CAB` allows

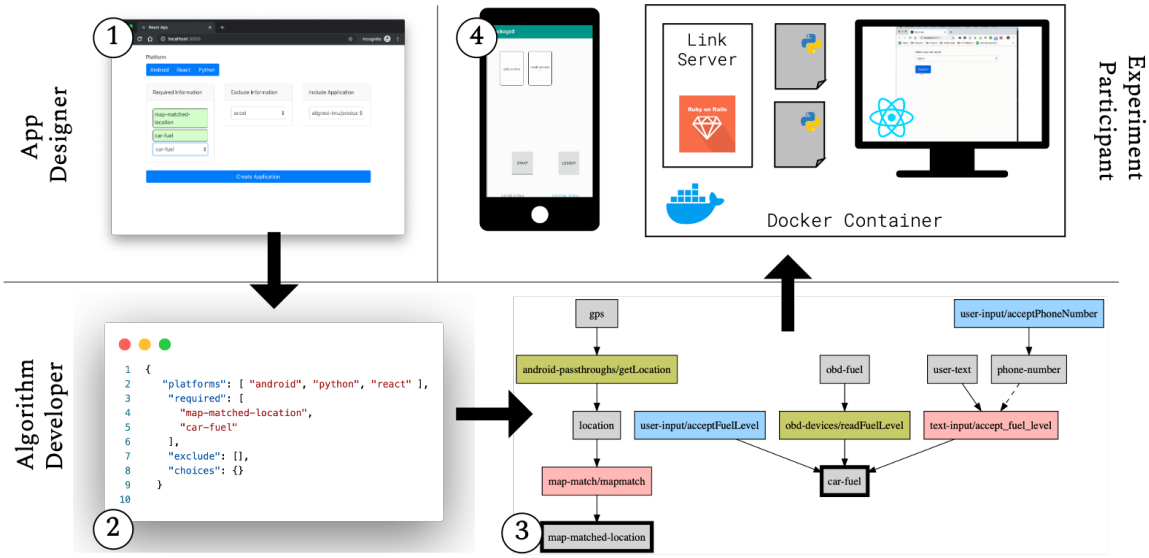


Figure 5.6: The app designer inputs the high-level requirements using our web interface (#1). This is translated to a JSON specification file (#2) which is used by `CAB` to assemble the necessary algorithmic modules. The dependency graph (#3) lists the compiled algorithmic modules to meet this requirement. The user-input information is shown in the gray boxes with a thick black border. The remaining information (gray boxes), Android algorithms (green), React algorithms (blue) and Python algorithms (pink) are automatically determined. `CAB` uses the dependency graph to generate the individual components for the data-collection platform (#4).

the app designer to take advantage of the redundancy present in the repository without additional effort.

Finally, CAB materializes the strategy by compiling an Android APK with the required Android algorithms and launches a Docker container with the Python scripts needed for the server-side algorithms and a React website with all necessary React-based algorithms. The sign-up URL is shared with the app designer to distribute to experiment participants.

The app designer's effort to build this data-collection platform with multiple redundant paths is significantly reduced to simply specifying two information types on a web interface drop down, and distributing the sign-up link to experiment participants.

5.5.2 Case Study 2 – Car Sensor Estimation

Next, we demonstrate an app which estimates the vehicle speed, steering wheel angle and gear position using smartphone sensors. Estimating vehicular sensors is a useful concept in numerous studies. Several studies use the gyroscope to estimate the vehicle steering wheel angle [64, 17, 91] and others estimate the vehicle speed for various purposes such as road pavement monitoring [92] or instantaneous vehicle speed estimation [114].

In order to use CAB to achieve this, an app designer uses our web interface to select three information types: *car-speed, car-steering, and car-gear*. CAB searches through the algorithms in the repository to determine the required dependencies to produce the three necessary information types. Fig. 5.7 shows the list of algorithms and information included in this app. The effort from the app designer perspective is significantly reduced to just using this web interface to select the required sensors, and CAB produces the Android app to achieve this goal.

The individual blocks in the compiled application are supplied by algorithm developers and contributed to the repository. Due to the automatic code generation

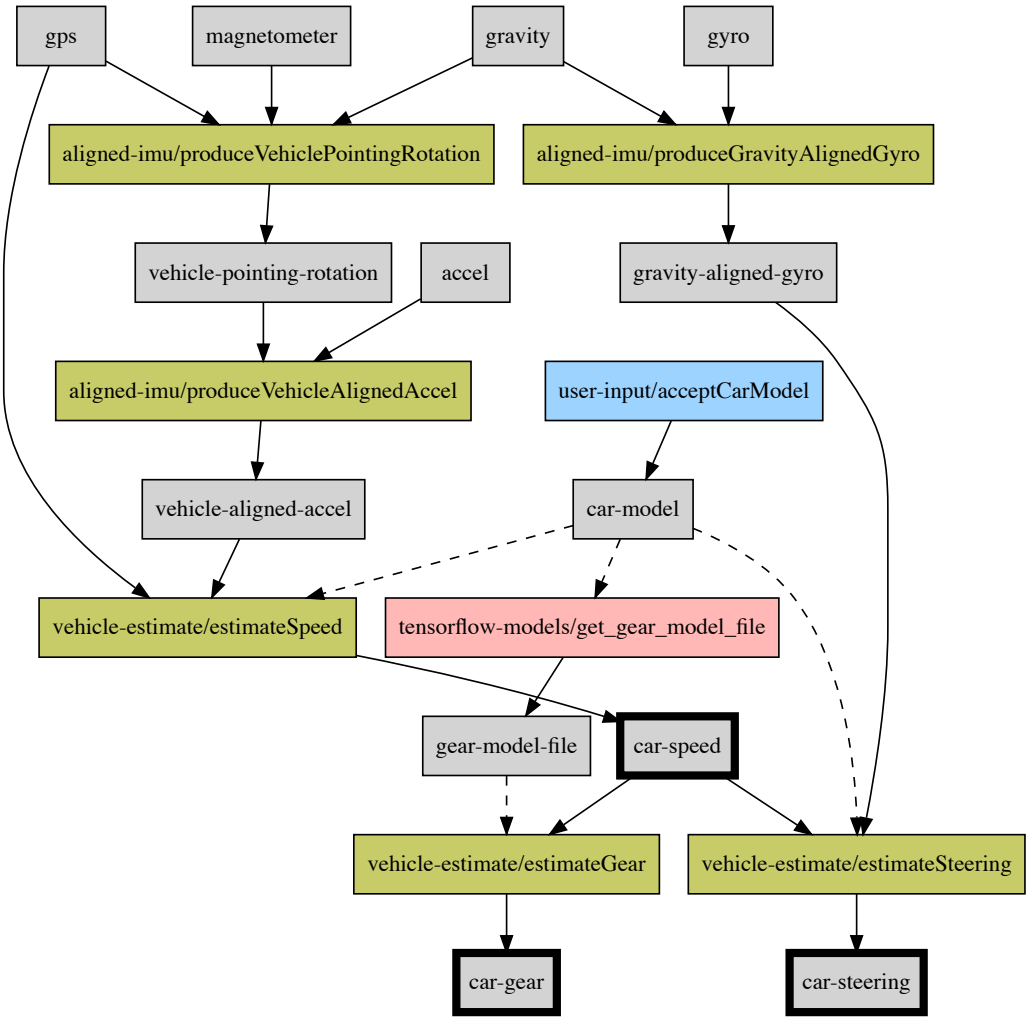


Figure 5.7: All information and algorithms compiled together for a vehicle sensor estimation case study. All information blocks are in gray. The input required information are the three gray blocks with a thick border. The remaining dependencies and all algorithms were determined by CAB's graph search.

Platform	#	**Dev LoC**
Android	6	156 / 667 (19%)
Python	1	11 / 116 (8.7%)
React	1	79 / 189 (29.5%)

Table 5.4: Developer-supplied code for each algorithm for vehicle estimation demo app

within CAB, algorithm developers can focus on just the estimation logic. CAB generates function stubs where the required input is passed in as function arguments, as shown in Fig. 5.4. In this example app, the developer-supplied code only makes up a fraction of the auto-generated code, as shown in Fig. 5.4.

5.5.3 Case Study 3 – Obstacle/Hazard Warning

We created an obstacle/hazard avoidance app using CAB. Upcoming obstacle warning is part of navigation apps like Google Maps [41] or Waze [108]. An obstacle/hazard warning app needs to collect reports from multiple users and warn users of upcoming hazards based on the vehicle's current location. CAB compiled a data-collection platform to achieve this containing of 1 Android component to get the user's location, 1 React component to show the map and accept reports of obstacles and 1 Python component to collect reports from multiple users and disseminate the information.

The Python algorithm collects the individually reported sightings and shares a combined sightings report to all drivers. The Python algorithm runs in the same sandbox for all users using this app, and hence it is easy to pool data across drivers. This Python script only contains 13 lines of developer-supplied code and 33 lines of auto-generated code. This highlights the minimal effort required by the algorithm developer to collect data from multiple drivers. The CAB linking server automatically routes all information within the data collection campaign.

The React algorithm, shown in Fig. 5.8, consists of a map to display upcoming obstacles and several buttons using which the user can flag new obstacles observed near their location. React can be used to create complex user interfaces such as this interactive map application. The required information (e.g. upcoming obstacles) is automatically routed into the React application through the properties of the React components.

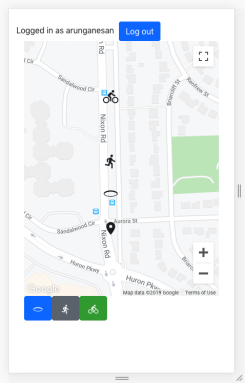


Figure 5.8: Obstacle avoidance/warning app built using CAB. The React algorithm has a maps interface here it displays upcoming obstacles *sightings-map* and outputs a new sighting if the user presses one of the three buttons.

5.6 Evaluation

Here we evaluated the effort reduction for app designers and algorithm developers. **App designers** who wish to launch a data-collection platform without the support of CAB need to engage in multiple steps in addition to programming the data-collection platform. They need to (1) list the required information for the data-collection platform, (2) develop the tools needed to collect the information, (3) set up a server to receive the data from multiple users, (4) create a website to distribute the app to participants, and (5) optionally set up a web hook to receive texts from participants if that is required for their data-collection platform. CAB significantly shortens this process by only requiring step #1 — the list of required information. It automatically completes the remaining steps of the data collection.

In our evaluation we built 18 different functions contained in 10 algorithms spanning 3 platforms. A data-collection platform is created using a combination of these 18 functions including any dependencies that are needed for each function. Given the current state of our repository, CAB can create a total of *8,100 unique applications* by assembling different combinations of the 18 functions. This is a 450× growth in possible data- collection platforms given only 18 functions contributed by developers. We foresee this number drastically increasing with contributions by other developers.

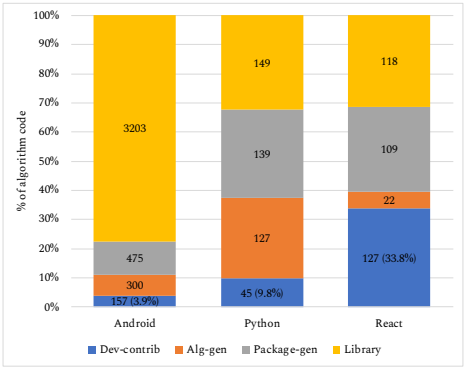


Figure 5.9: Lines of code contributed by developer compared to auto-generated and library components.

We also quantified the effort reduction for **algorithm developers**. Due to the automatic code generated by CAB, algorithm developers need only be concerned with the logic of their algorithm and not with any other aspect of data collection such as accessing low-level sensors or communicating information with the server. Fig. 5.9 shows the lines of code needed to implement algorithms in each platform compared to auto-generated code and libraries. Developers only need to contribute 3.9% of Android code, 9.8% of Python and 33.8% of React code with the logic for each component. The Android libraries are especially large because they have to interface with various low-level sensors.

5.7 User Study

We recruited 5 researchers to gauge the usability of CAB. We explained the concept of CAB and showed them a video demo of CAB[6]. Three of the participants are vehicular domain experts and the other two are non-experts. After the presentation of CAB, we asked them to rate the app designer interface and the algorithm developer workflow. For both components, the participants answered (Q1) *"Would you use CAB for your future research?"* (rated 1 - 7 where 7 is "definitely would use") and (Q2) *"Do you perceive CAB will reduce your development and deployment effort?"* (rated 1 - 7 where

[6] https://www.youtube.com/playlist?list=PLuEbkT_dQmRIn7JOoDf2JHKQT0n_xfcFc

7 is "definitely would reduce effort").

Below we summarize the survey results and some recurring comments raised by participants in free-form discussion. We uncovered several practical challenges that must be addressed when deploying `CAB`.

App Designer. Participants found the app designer interface to be very useful and would consider using it in their own research (score=6.7, std=0.4). One domain-expert researcher said

> *"[App development is] really frustrating – if I can use [`CAB`] automatically to do that, it would reduce the learning curve for many novices. Even for researchers it would reduce effort significantly"*

This sentiment was shared by other participants as well. They largely found that the app designer interface would reduce the effort needed to develop such data collection apps (score=6.7, std=0.49).

In free-form discussion multiple participants commented that the definition of information types needs to be well-documented. One participant said the documentation would help answer *"How can I know if `CAB` can help with my purpose?"*.

Algorithm Developer. Participants were largely interested in using the algorithm development workflow for their future research (score=6.4, std=0.8). The participant who gave the lowest score (5) suggested that a graphical user-interface would be better for defining the algorithm specifications. Participants suggested that the specification files and script invocation doesn't need to be exposed to developers. With such a user interface, they would be more willing to use the system (score=6) with the following quote on its simplicity:

> *"I already have the algorithm. So I already write that. So I just copy it. That's very convenient, it will not take much time."*

The participants agreed that the algorithm generation would reduce the effort needed by algorithm developers (score=6.8, std=0.4).

General Comments. Multiple participants commented on the best practices for contributing to the `CAB` repository. As multiple developers create algorithms to be used with `CAB`, we need a standard code-review procedure to ensure the algorithms meet quality and accuracy standards. One participant suggested offline testing of the algorithms by feeding in trace files for input and verifying their output values. However, the details of the open source collaboration of `CAB` is outside the scope of this work. We foresee answering these questions as we deploy `CAB` and grow a community of contributors and users.

5.8 Related Work

Vehicular research spans a diverse set of areas including driver monitoring [87, 54, 96], road anomaly detection [32, 3, 103], and vehicular security [72, 49, 68]. Due to the lack of a very flexible reconfigurable data collection builder, most of these researchers build their own data collection tools. This is a high barrier of entry for non-technical researchers who would like to enter this field and investigate vehicle-related research questions. Furthermore, since platforms are built for a specific purpose, they often lack the flexibility to take advantage of redundant ways to measure the same information.

Most data collection apps have similar requirements. They must all first access and process low level sensor data from the vehicle or the phone. This requires understanding how to interface with different hardware devices and consolidating all the information in one place. Next they often need to upload the data to a remote server for later processing, and handle user management of the uploaded data. In many cases, the data needs to be communicated between multiple aspects of the application that is deployed on the user's phone. A general data collection platform needs

to address these requirements.

We present an overview of the current state of vehicular data collection platforms below.

5.8.1 Specialized Data Collection

Certain use cases require custom-built data collection tools and therefore cannot be automated with a general configurable tool builder. For instance, IVBSS [42] and collects data from modified Honda Accords and Bender *et al.* [8] require integration with the vehicle's LiDAR and other sensors. Other data collection platforms require adding sensors to vehicles. The Safety Pilot Model Deployment [10] outfits cars with a DSRC antenna, an aftermarket safety device and sometimes with a MobileEye camera [100].CANOPNR [95] is an OBD-II data logger built using Arduino which can run local processing and offload the data to the cloud. This platform was used to study slippery road conditions [30]. BigRoad [64] uses an easy-to-deploy data collection platform [63] consisting of an IMU sensor attached to steering wheel angle and a smartphone app. These research undertakings require heavy engineering effort and must require custom platforms to suit their special needs.

5.8.2 General Data Collection

Other vehicular research efforts can benefit from a general data collection builder tool. For instance, SenseMyCity [84] is a crowdsourcing mobile platform that collects data from the smartphone and the vehicle through the OBD-II port. This has been used to study city-wide fuel consumption [83] and the mental state of bus drivers [85]. Chen *et al.* [17] built V-Sense, which uses smartphone-based sensing to detect steering maneuvers. Walhstörm *et al.* [103] include many such examples in their survey. These investigations can be expedited by the existence of a simple data collection tool builder which can be configured to meet their specific needs. This would enable non-technical

researchers to undertake similar research problems.

CARLOG [52] defines a programming abstraction on top of vehicular events. Their programming abstraction can be composed to find driving events such as turning or reckless driving. In contrast, `CAB` defines functionality as callback functions which can perform procedural computation and store internal state. Information and events is composed in `CAB` by routing information from one algorithmic module to another. In addition, we also provide other convenient functionality such as automatic code generation and a web-UI to generate the data collection campaign.

5.8.3 Reusable Data-Collection Platforms

There are a few notable platforms which have been re-used across multiple investigations. The CarTel hardware data collection platform [48] was customized with additional sensors and used in several follow up work [31, 71, 32, 97]. However, this platform wasn't designed to be easily extended to additional use cases and must be manually modified for each investigation. In contrast, `CAB` can be easily extended for future required functionality.

Sensibility Testbed [82] has a web interface through which researchers can submit their data collection tasks. It automatically deploys the task to users who have installed the Sensibility Testbed app. This tool makes it very easy to do data collection, however it does not allow for developers to prototype any real-time custom functionality on top of the data collection platform. `CAB` generates the data collection platform which allows customization and extension to meet each data collection need.

HealthSense [24] provides a user interface to define clinical trials, distribute to participants and collect data. In contrast, `CAB` also allows for rapid prototyping of new algorithmic ideas. Using the auto-generated code, algorithm developers can extend the functionality of `CAB` to add novel user interfaces or engage in real-time processing of user data.

5.9 Discussion & Future Work

Communicating App Requirements `CAB` accepts the requirements as a JSON specification file (Fig. 5.10c) and assembles the data-collection platform. We created a simple website to create this requirements file, but there is much room for exploration and investigation in this area. Ultimately, the goal of a good interface is to learn the user's intent and requirements and automatically translate that to a `CAB` input file. This may be done using natural language processing akin to many general voice assistants such as Amazon's Alexa or domain-specific voice assistants such as Clinc.[7] An investigation into the most natural form of input is outside the scope of this work but worth further investigation.

Application to Personal Informatics The core concepts of `CAB` are applicable more broadly than vehicular data collection. We foresee the application of the same tools to personal health informatics, as seen in related work [24, 9]. Personal informatics is a rich area of research as we can infer the user's mental state or emotional level based on numerous low-level sensors, such as their heart rate or daily movement. Such informatics can help raise self-awareness of the user's own state of mind and well-being.

Expansion to Additional Sensors Future work can expand `CAB` to additional hardware sensors such as wearables or rich-vehicle sensors such as camera, radar or LiDAR. `CAB` naturally has support for acquiring data from one platform and sending it to another platform for processing. Expanding to other sensors will leverage this functionality to allow easy access and rapid prototyping of novel applications. For example, Augmented Vehicle Reality [81] shares LiDAR point cloud information across vehicles to detect obstacles in blind spots. Such technology can be quickly

[7] `https://clinc.com/`

prototyped using CAB.

5.10 Conclusion

We have presented CAB, an on-demand vehicular data-collection app builder. CAB accepts high-level requirements from researchers and developers, and creates a custom data-collection app to meet their needs. The app is easily distributed through the CAB server and doesn't require any programming expertise to launch a custom data-collection campaign.

CAB compiles each on-demand data-collection platform using algorithmic modules contributed to the shared repository by developers. Using a shared understanding of *information types* that are requested by the user, CAB automatically resolves algorithmic dependencies and builds an app to meet the requirements. By assembling different sets of algorithms, CAB can create numerous unique data-collection apps using just a small number of algorithms. In this work, we created 18 algorithms which can be composed to create 8,100 unique applications.

We have demonstrated CAB's versatility by building three complex data-collection apps. CAB significantly reduces the programming effort needed to achieve complex app functionality, requiring as little as 4% of the Android code that would otherwise be needed. Through a usability study, we found that researchers find that CAB reduces the effort for both data collection and for algorithm development, and are likely to use CAB for their research or app development needs.

5.11 Appendix: Specification Files

```
{
  "location": {"type": "float[2]"},
  "car-speed": {"type": "float"},
  "car-fuel": {"type": "float"},
  "car-steering": {"type": "float"},

  "vehicle-pointing-rotation":
      {"type": "float[9]" },
  "vehicle-aligned-accel":
      {"type": "float[3]" },
  "accel":
      {"type": "float[3]",
      "sensor": true},
  "gps":
      {"type": "float[3]",
      "sensor": true},

  "phone-number": {"type": "string"},
}
```

(a) Information registry specification

```
{
"aligned-imu" : {
  "platform": "android",
  "functions": {
    "produceVehicleAlignedAccel": {
      "output": "vehicle-aligned-accel",
      "input": [
          "accel",
          "vehicle-pointing-rotation"]
    },
    "produceVehiclePointingRotation": {
      "output":
        "vehicle-pointing-rotation",
      "input": ["magnetometer", "gps",
                "gravity"]
    },
  }
}
}
```

(b) Algorithm function specification

```
{
  "required": [
      "location",
      "map-matched-location",
      "car-fuel"
  ],
  "platforms": [
      "android",
      "python",
      "react"
  ],
  "exclude": [
      "accel"
  ],
  "choices": {
      "car-fuel":
        "obd-devices/readFuelLevel"
  }
}
```

(c) App designer requirements

```
[
{ "algorithm": "android-passthroughs",
  "function": "getLocation" },

{ "algorithm": "map-match",
  "function": "mapmatch" },

{ "algorithm": "user-input",
  "function": "acceptFuelLevel" },

{ "algorithm": "obd-devices",
  "function": "readFuelLevel" },

{ "algorithm": "text-input",
  "function": "accept_fuel_level" },

{ "algorithm": "user-input",
  "function": "acceptPhoneNumber" }
]
```

(d) App designer strategy

Figure 5.10: Specification files

CHAPTER VI

Thesis Contributions and Conclusion

This thesis made three main contributions towards unifying diverse sensor streams. By unifying sensor streams, we can take full advantage of the information learned by any individual sensor or vehicle. We built systems which utilized sensor redundancy and solved problems in two different domains.

CAN-bus Injection Detection. First, we developed a system called `CarSec` which uses smartphone sensors to detect if the vehicular sensors are deviating from their expected value. We estimated six (6) different vehicular sensors and measured their accuracy under many realistic driving scenarios. We characterized CAN-bus injection attacks found in literature and demonstrated `CarSec`'s ability to exploit the sensor redundancy found in phones for their detection.

Driving Hazard Detection. Next, we developed a system called `Ubi` which uses GPS trajectories of nearby cars to detect dangerous driving hazards. `Ubi` detects stationary and mobile driving hazards using GPS trajectories of nearby vehicles. We modeled the mobility of these hazards to predict their location and warn drivers early on as they approach the hazard. By leveraging the GPS sensing redundancy from multiple nearby vehicles, we are able to make the roads safer and provide earlier warning about dangerous hazards ahead.

Data-Collection App Building. Finally, we consolidated the main requirements for building vehicular data collection applications which leverage redundant sensors into a data-collection app builder called `CAB`. In contrast to existing tools, `CAB` is designed so that non-technical users can easily create and launch a data-collection campaign for their research purposes. Moreover, `CAB` is designed to be easily extended by developers and supports development in three different platforms – Android (Java), React (TypeScript) or Python scripts.

CHAPTER VII

Interesting Future Direction

There are several interesting directions to follow up this thesis.

Extensions to `CarSec`. `CarSec` uses three sensors, GPS, IMU and magnetometer, to replicate six vehicular sensors. As a possible extension of this work, one may consider using the camera and microphone sensors found in smartphones. Every year smartphone cameras improve. Modern smartphones even have depth cameras and multiple cameras for a wide field of view. This could serve as an additional source of redundancy for detecting attacks. By replicating the vision of in-vehicle LiDAR or camera, we can leverage smartphone cameras to the detect adversarial machine learning attacks [79], or sensor blinding attacks [80, 110]. If we can share the extracted camera information across vehicles, we can extend sensor redundancy and explore a wide range of attacks. Similar work on vision sharing between vehicles was done by Qiu *et al.* [81].

Extensions to `Ubi`. The graph search algorithm used in `Ubi` is a general representation of hazard mobility. It can represent diverse hazard mobility and used to represent how cars interact with the hazard, such as how they slow down or speed up as they approach the hazard. We can explore the full limits of such a tool by extending `Ubi` to other types of hazards not explored in this thesis. This could be

extended to detect stray animals on the road which have a distinct mobility pattern. Being able to detect animals and report in real-time could potentially can save lives by early warnings.

Extensions to `CAB`. We developed a flexible and extensible data-collection platform for vehicular data-collection research. A possible extension of this work involves expanding the development surface to different platforms. For example, currently in order to access low-level sensors on the phone, a developer has to use Android-based algorithms available within `CAB`. If we add support for React-Native or Swift, a user can also access low-level sensors on iPhones or other mobile operating systems, thereby increasing the coverage and impact of `CAB`.

Bibliography

[1] Martín Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Gabriel Agamennoni et al. "Anomaly detection in driving behaviour by road profiling". In: *2013 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2013, pp. 25–30. ISBN: 1-4673-2755-7.

[3] H. Aly, A. Basalamah, and M. Youssef. "Map++: A crowd-sensing system for automatic map semantics identification". In: *2014 Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. June 2014, pp. 546–554. DOI: 10.1109/SAHCN.2014.6990394.

[4] ARILOU. *Automotive Cyber Security, Part of NNG Group* . https://ariloutech.com/. 2018.

[5] David Arthur and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. ISBN: 0-89871-624-1.

[6] automatic. *Automatic*. https://automatic.com/. 2019. URL: https://automatic.com/ (visited on 11/14/2019).

[7] S. Barnwal. "Vehicle Behavior Analysis for Uneven Road Surface Detection". In: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. Sept. 2015, pp. 1719–1722. DOI: 10.1109/ITSC.2015.279.

[8] Asher Bender et al. "A flexible system architecture for acquisition and storage of naturalistic driving data". In: *IEEE Transactions on Intelligent Transportation Systems* 17.6 (2016), pp. 1748–1761.

[9] Frank Bentley et al. "Health Mashups: Presenting Statistical Patterns Between Wellbeing Data and Context in Natural Language to Promote Behavior Change". In: *ACM Trans. Comput.-Hum. Interact.* 20.5 (Nov. 2013), 30:1–30:27. ISSN: 1073-0516. DOI: 10.1145/2503823. URL: http://doi.acm.org/10.1145/2503823 (visited on 12/13/2019).

[10] D. Bezzina and J. Sayer. "Safety pilot model deployment: Test conductor team report". In: *Report No. DOT HS* 812 (2014), p. 171.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2007.

[12] R. Carisi et al. "Enhancing in vehicle digital maps via GPS crowdsourcing". In: *2011 Eighth International Conference on Wireless On-Demand Network Systems and Services*. Jan. 2011, pp. 27–34. DOI: 10.1109/WONS.2011.5720196.

[13] Pew Research Center. *Demographics of Mobile Device Ownership and Adoption in the United States*. http://www.pewinternet.org/fact-sheet/mobile/. Feb. 2018.

[14] Stephen Checkoway et al. "Comprehensive Experimental Analyses of Automotive Attack Surfaces." In: *USENIX Security Symposium*. San Francisco, 2011.

[15] Dongyao Chen, Kyong-Tak Cho, and Kang G. Shin. "Mobile IMUs Reveal Driver's Identity From Vehicle Turns". In: *arXiv:1710.04578 [cs]* (Oct. 2017). arXiv: 1710.04578. URL: http://arxiv.org/abs/1710.04578 (visited on 10/25/2019).

[16] Dongyao Chen and Kang G. Shin. "TurnsMap: Enhancing Driving Safety at Intersections with Mobile Crowdsensing and Deep Learning". In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3.3 (Sept. 2019), 78:1–78:22. ISSN: 2474-9567. DOI: 10.1145/3351236. URL: http://doi.acm.org/10.1145/3351236 (visited on 10/25/2019).

[17] Dongyao Chen et al. "Invisible Sensing of Vehicle Steering with Smartphones". In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '15. event-place: Florence, Italy. New York, NY, USA: ACM, 2015, pp. 1–13. ISBN: 978-1-4503-3494-5. DOI: 10.1145/2742647.2742659. URL: http://doi.acm.org/10.1145/2742647.2742659 (visited on 10/25/2019).

[18] Z. Chen et al. "D3: Abnormal driving behaviors detection and identification using smartphone sensors". In: *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. June 2015, pp. 524–532. DOI: 10.1109/SAHCN.2015.7338354.

[19] Kyong-Tak Cho and Kang G. Shin. "Error handling of in-vehicle networks makes them vulnerable". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1044–1055. ISBN: 1-4503-4139-X.

[20] Kyong-Tak Cho and Kang G. Shin. "Fingerprinting Electronic Control Units for Vehicle Intrusion Detection." In: *USENIX Security Symposium*. 2016, pp. 911–927.

[21] Kyong-Tak Cho, Kang G. Shin, and Taejoon Park. "CPS approach to checking norm operation of a brake-by-wire system". In: *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*. ACM, 2015, pp. 41–50. ISBN: 1-4503-3455-5.

[22] Moritz Contag et al. "How They Did It: An Analysis of Emission Defeat Devices in Modern Automobiles". In: *2017 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2017, pp. 231–250. DOI: 10.1109/SP.2017.66.

[23] V. Coric and M. Gruteser. "Crowdsensing Maps of On-street Parking Spaces". In: *2013 IEEE International Conference on Distributed Computing in Sensor Systems*. May 2013, pp. 115–122. DOI: 10.1109/DCOSS.2013.15.

[24] Aidan Curtis et al. "HealthSense: Software-defined Mobile-based Clinical Trials". In: *The 25th Annual International Conference on Mobile Computing and Networking*. MobiCom '19. event-place: Los Cabos, Mexico. New York, NY, USA: ACM, 2019, 32:1–32:15. ISBN: 978-1-4503-6169-9. DOI: 10.1145/3300061.3345433. URL: http://doi.acm.org/10.1145/3300061.3345433 (visited on 12/12/2019).

[25] Bernard Desgraupes. "Clustering indices". In: *University of Paris Ouest-Lab Modal'X* 1 (2013). https://cran.biodisk.org/web/packages/clusterCrit/vignettes/clusterCrit.pdf, p. 34.

[26] Piotr Dollar et al. "Pedestrian detection: An evaluation of the state of the art". In: *IEEE transactions on pattern analysis and machine intelligence* 34.4 (2011), pp. 743–761.

[27] NHTSA DoT. *Traffic Safety Facts: 2017 Data. Pedestrians*. Tech. rep. 2017. URL: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812681.

[28] Miro Enev et al. "Automobile Driver Fingerprinting". In: *Proceedings on Privacy Enhancing Technologies* 2016.1 (2015), pp. 34–50. ISSN: 2299-0984. DOI: 10.1515/popets-2015-0029. URL: https://www.degruyter.com/dg/viewarticle/j$002fpopets.2016.2016.issue-1$002fpopets-2015-0029$002fpopets-2015-0029.xml (visited on 10/25/2019).

[29] D. J. Enriquez et al. "CANOPNR: CAN-OBD programmable-expandable network-enabled reader for real-time tracking of slippery road conditions using vehicular parameters". In: *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*. IEEE, 2012, pp. 260–264. ISBN: 1-4673-3063-9.

[30] Denrie Enriquez et al. "On software-based remote vehicle monitoring for detection and mapping of slippery road sections". In: *International journal of intelligent transportation systems research* 15.3 (2017), pp. 141–154.

[31] Jakob Eriksson, Hari Balakrishnan, and Samuel Madden. "Cabernet: vehicular content delivery using WiFi". In: *Proceedings of the 14th ACM international conference on Mobile computing and networking*. ACM, 2008, pp. 199–210. ISBN: 1-60558-096-1.

[32] Jakob Eriksson et al. "The pothole patrol: using a mobile sensor network for road surface monitoring". en. In: ACM Press, 2008, p. 29. ISBN: 978-1-60558-139-2. DOI: 10.1145/1378600.1378605. URL: http://portal.acm.org/citation.cfm?doid=1378600.1378605 (visited on 03/13/2018).

[33] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 1996, pp. 226–231.

[34] P. Faruki et al. "Android Security: A Survey of Issues, Malware Penetration, and Defenses". In: *IEEE Communications Surveys Tutorials* 17.2 (2015), pp. 998–1022. ISSN: 1553-877X. DOI: 10.1109/COMST.2014.2386139.

[35] Ford. *2017 Escape Tech Specs*. https://media.ford.com/content/\dam/fordmedia/NorthAmerica/US/Events/17-LAAS/2017-ford-escape-tech-specs.pdf. 2018.

[36] Ford. *OpenXC*. http://openxcplatform.com/. 2018.

[37] Ian D. Foster et al. "Fast and Vulnerable: A Story of Telematic Failures." In: *WOOT*. 2015.

[38] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data clustering: theory, algorithms, and applications*. Vol. 20. Siam, 2007. ISBN: 0-89871-623-3.

[39] Arun Ganesan, Jayanthi Rao, and Kang Shin. *Exploiting Consistency Among Heterogeneous Sensors for Vehicle Anomaly Detection*. Tech. rep. SAE Technical Paper, 2017.

[40] Raghu K. Ganti et al. "GreenGPS: a participatory sensing fuel-efficient maps application". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 151–164. ISBN: 1-60558-985-3.

[41] Google Google. *Google Maps*. en. Dec. 2019. URL: https://www.google.com/maps (visited on 12/09/2019).

[42] Paul Green. "Integrated vehicle-based safety systems (IVBSS): Human factors and driver-vehicle interface (DVI) summary report". In: (2008). http://deepblue.lib.umich.edu/bitstream/2027.42/58189/1/100874.pdf.

[43] Kyusuk Han, Swapna Divya Potluri, and Kang G. Shin. "On authentication in a connected vehicle: Secure integration of mobile devices with vehicular networks". In: *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 160–169. ISBN: 1-4503-1996-3.

[44] Ian Hawkins. *Torque Pro (OBD 2 & Car) - Apps on Google Play*. en. Dec. 2019. URL: https://play.google.com/store/apps/details?id=org.prowl.torque&hl=en_US (visited on 12/09/2019).

[45] Jin-Hyuk Hong, Ben Margines, and Anind K. Dey. "A smartphone-based sensing platform to model aggressive driving behaviors". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Toronto, Ontario, Canada: ACM, 2014, pp. 4047–4056. ISBN: 978-1-4503-2473-1.

[46] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. "Security threats to automotive CAN networks–practical examples and selected short-term countermeasures". In: *International Conference on Computer Safety, Reliability, and Security*. Springer, 2008, pp. 235–248.

[47] Shaohan Hu et al. "SmartRoad: Smartphone-Based Crowd Sensing for Traffic Regulator Detection and Identification". In: *ACM Trans. Sen. Netw.* 11.4 (2015), pp. 1–27.

[48] Bret Hull et al. "CarTel: a distributed mobile sensor computing system". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 125–138. ISBN: 1-59593-343-3.

[49] Rob Millerb Ishtiaq Roufa et al. "Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study". In: *19th USENIX Security Symposium, Washington DC*. 2010, pp. 11–13.

[50] Mohit Jain et al. "Speed-Breaker Early Warning System." In: *NSDR*. 2012.

[51] B. Jiang and Y. Fei. "Traffic and vehicle speed prediction with neural network and Hidden Markov model in vehicular networks". In: *2015 IEEE Intelligent Vehicles Symposium (IV)*. June 2015, pp. 1082–1087. DOI: 10.1109/IVS.2015.7225828.

[52] Yurong Jiang et al. "Carloc: Precise positioning of automobiles". In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2015, pp. 253–265. ISBN: 1-4503-3631-0.

[53] D. A. Johnson and M. M. Trivedi. "Driving style recognition using a smartphone as a sensor platform". In: *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. Oct. 2011, pp. 1609–1615. DOI: 10.1109/ITSC.2011.6083078.

[54] Nidhi Kalra and Divya Bansal. "Analyzing driver behavior using smartphone sensors: a survey". In: *Int. J. Electron. Electr. Eng* 7.7 (2014), pp. 697–702.

[55] Arne Kesting, Martin Treiber, and Dirk Helbing. "General lane-changing model MOBIL for car-following models". In: *Transportation Research Record* 1999.1 (2007), pp. 86–94.

[56] Julia Kollewe. *Dixons Carphone warns on profits as customers keep phones for longer*. https://www.theguardian.com/business/2017/aug/24/dixons-carphone-profits-phones-weak-pound-upgrades. 2017.

[57] Karl Koscher et al. "Experimental security analysis of a modern automobile". In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 447–462. ISBN: 1-4244-6895-7.

[58] Tyler Lee. *Samsung Galaxy S9 Rumored To Pack 3,200mAh Battery*. https://www.ubergizmo.com/2017/12/galaxy-s9-3200mah-battery-rumor/. 2017.

[59] John Leonard et al. "A perception-driven autonomous urban vehicle". In: *Journal of Field Robotics* 25.10 (2008), pp. 727–774.

[60] Fu Li et al. "Dangerous driving behavior detection using smartphone sensors". In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2016, pp. 1902–1907. ISBN: 1-5090-1889-1.

[61] Hongyu Li et al. "Automatic Unusual Driving Event Identification for Dependable Self-Driving". In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. Shenzhen, China: ACM, 2018, pp. 15–27. ISBN: 978-1-4503-5952-8.

[62] Chao Liu et al. “An unsupervised spatiotemporal graphical modeling approach to anomaly detection in distributed CPS”. In: *Cyber-Physical Systems (ICCPS), 2016 ACM/IEEE 7th International Conference on*. IEEE, 2016, pp. 1–10. ISBN: 1-5090-1772-0.

[63] Luyang Liu. “Enabling Large-Scale Road Data Acquisition with an Easy-to-Install and Sustainable Setup”. In: *Proceedings of the 2017 Workshop on MobiSys 2017 Ph. D. Forum*. ACM, 2017, pp. 3–4. ISBN: 1-4503-4957-9.

[64] Luyang Liu et al. “Bigroad: Scaling road data acquisition for dependable self-driving”. In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 371–384. ISBN: 1-4503-4928-5.

[65] Xuemei Liu et al. “Mining Large-scale, Sparse GPS Traces for Map Inference: Comparison of Approaches”. In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’12. event-place: Beijing, China. New York, NY, USA: ACM, 2012, pp. 669–677. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339637. URL: http://doi.acm.org/10.1145/2339530.2339637 (visited on 08/05/2019).

[66] The Manifest. *The Popularity of Google Maps: Trends in Navigation Apps in 2018 — The Manifest*. 2018. URL: https://themanifest.com/app-development/popularity-google-maps-trends-navigation-apps-2018 (visited on 11/25/2019).

[67] Javier E. Meseguer et al. “Drivingstyles: A smartphone application to assess driver behavior”. In: *2013 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2013, pp. 000535–000540. ISBN: 1-4799-3755-X.

[68] C. Miller and C. Valasek. “Advanced CAN injection techniques for vehicle networks”. In: *Presentation at BlackHat USA* (2016). http://illmatics.com/can%20message%20injection.pdf.

[69] Charlie Miller and Chris Valasek. “Adventures in automotive networks and control units”. In: *Def-Con* 21 (2013). https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf, pp. 260–264.

[70] Charlie Miller and Chris Valasek. “Remote exploitation of an unaltered passenger vehicle”. In: *Black Hat USA* 2015 (2015). http://illmatics.com/Remote%20Car%20Hacking.pdf.

[71] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. “Nericell: rich monitoring of road and traffic conditions using mobile smartphones”. In: *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 2008, pp. 323–336. ISBN: 1-59593-990-3.

[72] Michael Müter and Naim Asaj. “Entropy-based anomaly detection for in-vehicle networks”. In: *Intelligent Vehicles Symposium (IV), 2011 IEEE*. IEEE, 2011, pp. 1110–1115. ISBN: 1-4577-0891-4.

[73] 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*. IEEE, 2010, pp. 92–98. ISBN: 1-4244-7409-4.

[74] NHTSA. *Distracted Driving — NHTSA*. https://www.nhtsa.gov/risky-driving/distracted-driving. Apr. 2019. (Visited on 04/21/2019).

[75] OpenStreetMap. *OpenStreetMap Foundation Wiki* . https://wiki.osmfoundation.org/. 2018.

[76] World Health Organization. *Top 10 causes of death*. https://www.who.int/en/news-room/fact-sheets/detail/the-top-10-causes-of-death. Apr. 2019. (Visited on 04/21/2019).

[77] Miroslav Pajic et al. "Robustness of attack-resilient state estimators". In: *ICCPS'14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014)*. IEEE Computer Society, 2014, pp. 163–174. ISBN: 1-4799-4930-2.

[78] Bei Pan et al. "Crowd Sensing of Traffic Anomalies Based on Human Mobility and Social Media". In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL'13. event-place: Orlando, Florida. New York, NY, USA: ACM, 2013, pp. 344–353. ISBN: 978-1-4503-2521-9. DOI: 10.1145/2525314.2525343. URL: http://doi.acm.org/10.1145/2525314.2525343 (visited on 04/26/2019).

[79] Nicolas Papernot et al. "Practical black-box attacks against machine learning". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 506–519. ISBN: 1-4503-4944-7.

[80] Jonathan Petit et al. "Remote attacks on automated vehicles sensors: Experiments on camera and lidar". In: *Black Hat Europe* 11 (2015), p. 2015.

[81] Hang Qiu et al. "AVR: Augmented Vehicular Reality". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '18. event-place: Munich, Germany. New York, NY, USA: ACM, 2018, pp. 81–95. ISBN: 978-1-4503-5720-3. DOI: 10.1145/3210240.3210319. URL: http://doi.acm.org/10.1145/3210240.3210319 (visited on 11/24/2019).

[82] Michael Reininger et al. "A first look at vehicle data collection via smartphone sensors". In: *Sensors Applications Symposium (SAS), 2015 IEEE*. IEEE, 2015, pp. 1–6. ISBN: 1-4799-6117-5.

[83] Vitor Ribeiro, Joao Rodrigues, and Ana Aguiar. "Mining geographic data for fuel consumption estimation". In: *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*. IEEE, 2013, pp. 124–129. ISBN: 1-4799-2914-X.

[84] Joao GP Rodrigues et al. "A mobile sensing architecture for massive urban scanning". In: *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*. IEEE, 2011, pp. 1132–1137. ISBN: 1-4577-2197-X.

[85] João GP Rodrigues et al. "A mobile sensing approach to stress detection and memory activation for public bus drivers". In: *IEEE Transactions on Intelligent Transportation Systems* 16.6 (2015), pp. 3294–3303.

[86] Peter J. Rousseeuw. "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis". In: *Journal of computational and applied mathematics* 20 (1987), pp. 53–65.

[87] Arun Sahayadhas, Kenneth Sundaraj, and Murugappan Murugappan. "Detecting driver drowsiness based on sensors: a review". In: *Sensors* 12.12 (2012), pp. 16937–16953.

[88] J. Sayer et al. "Integrated vehicle-based safety systems field operational test final program report". In: (2011). `https://deepblue.lib.umich.edu/handle/2027.42/84378`.

[89] ARGUS Cyber Security. *Argus Cyber Security - Automotive Cyber Security*. `https://argus-sec.com/`. 2018.

[90] Fatjon Seraj et al. "A Smartphone Based Method to Enhance Road Pavement Anomaly Detection by Analyzing the Driver Behavior". In: *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. UbiComp/ISWC'15 Adjunct. event-place: Osaka, Japan. New York, NY, USA: ACM, 2015, pp. 1169–1177. ISBN: 978-1-4503-3575-1. DOI: `10.1145/2800835.2800981`. URL: `http://doi.acm.org/10.1145/2800835.2800981` (visited on 04/21/2019).

[91] Fatjon Seraj et al. "A smartphone based method to enhance road pavement anomaly detection by analyzing the driver behavior". In: *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. Osaka, Japan: ACM, 2015, pp. 1169–1177. ISBN: 978-1-4503-3575-1.

[92] Fatjon Seraj et al. "RoADS: A Road Pavement Monitoring System for Anomaly Detection Using Smart Phones". en. In: *Big Data Analytics in the Social and Ubiquitous Context*. Ed. by Martin Atzmueller et al. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 128–146. ISBN: 978-3-319-29009-6.

[93] Sayanan Sivaraman and Mohan Manubhai Trivedi. "Looking at vehicles on the road: A survey of vision-based vehicle detection, tracking, and behavior analysis". In: *IEEE Transactions on Intelligent Transportation Systems* 14.4 (2013), pp. 1773–1795.

[94] Isaac Skog and Peter Händel. "Indirect instantaneous car-fuel consumption measurements". In: *IEEE Transactions on Instrumentation and Measurement* 63.12 (2014), pp. 3190–3198.

[95] Kristian Smith and Jeffrey Miller. "OBDII data logger design for large-scale deployments". In: *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*. IEEE, 2013, pp. 670–674. ISBN: 1-4799-2914-X.

[96] Ines Teyeb et al. "A Drowsy Driver Detection System Based on a New Method of Head Posture Estimation". en. In: *Intelligent Data Engineering and Automated Learning – IDEAL 2014*. Ed. by Emilio Corchado et al. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 362–369. ISBN: 978-3-319-10840-7.

[97] Arvind Thiagarajan et al. "VTrack: accurate, energy-aware road traffic delay estimation using mobile phones". In: *Proceedings of the 7th ACM conference on embedded networked sensor systems*. ACM, 2009, pp. 85–98. ISBN: 1-60558-519-X.

[98] TowerSec. *Harman Automotive Cyber Security - TowerSec - Automotive Cyber Security (a HARMAN company)*. `http://tower-sec.com/`. 2018.

[99] Martin Treiber and Ame Kesting. "An open-source microscopic traffic simulator". In: *IEEE Intelligent Transportation Systems Magazine* 2.3 (2010), pp. 6–13.

[100] UMTRI. *Safety Pilot: Model Deployment*. `http://safetypilot.umtri.umich.edu/index.php?content=technology_overview`. 2018.

[101] Verizon. *Drive Smarter With Connected Car Technology — Hum by Verizon*. `https://www.hum.com/`. 2018.

[102] Bimal Viswanath et al. "Towards Detecting Anomalous User Behavior in Online Social Networks." In: *USENIX Security Symposium*. 2014, pp. 223–238.

[103] Johan Wahlström, Isaac Skog, and Peter Händel. "Smartphone-based vehicle telematics: A ten-year anniversary". In: *IEEE Transactions on Intelligent Transportation Systems* 18.10 (2017), pp. 2802–2825.

[104] Wenshuo Wang, Junqiang Xi, and Ding Zhao. "Driving style analysis using primitive driving patterns with bayesian nonparametric approaches". In: *IEEE Transactions on Intelligent Transportation Systems* (2018).

[105] Wenshuo Wang and Ding Zhao. "Extracting Traffic Primitives Directly From Naturalistically Logged Data for Self-Driving Applications". In: *IEEE Robotics and Automation Letters* 3.2 (Apr. 2018), pp. 1223–1229. ISSN: 2377-3774. DOI: 10.1109/LRA.2018.2794604.

[106] Yin Wang et al. "CrowdAtlas: Self-updating Maps for Cloud and Personal Use". In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '13. event-place: Taipei, Taiwan. New York, NY, USA: ACM, 2013, pp. 27–40. ISBN: 978-1-4503-1672-9. DOI: 10.1145/2462456.2464441. URL: http://doi.acm.org/10.1145/2462456.2464441 (visited on 04/21/2019).

[107] Armin Wasicek and Andre Weimerskirch. *Recognizing Manipulated Electronic Control Units*. English. SAE Technical Paper 2015-01-0202. Warrendale, PA: SAE International, Apr. 2015. DOI: 10.4271/2015-01-0202. URL: https://www.sae.org/publications/technical-papers/content/2015-01-0202/ (visited on 04/21/2019).

[108] Waze Waze. *Driving Directions, Traffic Reports, and Carpool Rideshares by Waze*. Dec. 2019. URL: https://www.waze.com/ (visited on 12/09/2019).

[109] Rui Xu and Donald Wunsch. "Survey of clustering algorithms". In: *IEEE Transactions on neural networks* 16.3 (2005), pp. 645–678.

[110] Chen Yan, Wenyuan Xu, and Jianhao Liu. "Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle". In: *Def-Con* 24 (2016).

[111] L. Yin et al. "Detecting illegal pickups of intercity buses from their GPS traces". In: *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. Oct. 2014, pp. 2162–2167. DOI: 10.1109/ITSC.2014.6958023.

[112] Chuang-Wen You et al. "CarSafe App: Alerting Drowsy and Distracted Drivers Using Dual Cameras on Smartphones". In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '13. event-place: Taipei, Taiwan. New York, NY, USA: ACM, 2013, pp. 13–26. ISBN: 978-1-4503-1672-9. DOI: 10.1145/2462456.2465428. URL: %5Curl%7Bhttp://doi.acm.org/10.1145/2462456.2465428%7D (visited on 04/21/2019).

[113] J. Yu et al. "Fine-Grained Abnormal Driving Behaviors Detection and Identification with Smartphones". In: *IEEE Transactions on Mobile Computing* 16.8 (Aug. 2017), pp. 2198–2212. ISSN: 1536-1233. DOI: 10.1109/TMC.2016.2618873.

[114] Jiadi Yu et al. "SenSpeed: Sensing Driving Conditions to Estimate Vehicle Speed in Urban Environments". In: *IEEE Transactions on Mobile Computing* 15.1 (Jan. 2016), pp. 202–216. ISSN: 2161-9875. DOI: 10.1109/TMC.2015.2411270.

[115] Daqing Zhang et al. "iBAT: detecting anomalous taxi trajectories from GPS traces". In: *Proceedings of the 13th international conference on Ubiquitous computing*. Beijing, China: ACM, 2011, pp. 99–108. ISBN: 978-1-4503-0630-0.

[116] M. Zhang et al. “SafeDrive: Online Driving Anomaly Detection From Large-Scale Vehicle Data”. In: *IEEE Transactions on Industrial Informatics* 13.4 (Aug. 2017), pp. 2087–2096. ISSN: 1551-3203. DOI: 10.1109/TII.2017.2674661.

[117] Yang Zheng et al. “Driving risk assessment using cluster analysis based on naturalistic driving data”. In: *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2014, pp. 2584–2589. ISBN: 1-4799-6078-0.

[118] Yu Zheng, Furui Liu, and Hsun-Ping Hsieh. “U-Air: when urban air quality inference meets big data”. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. Chicago, Illinois, USA: ACM, 2013, pp. 1436–1444. ISBN: 978-1-4503-2174-7.