ProCMotive: Bringing Programmability and Connectivity into Isolated Vehicles

ARSALAN MOSENIA and JAD F. BECHARA, Princeton University, USA
TAO ZHANG, Cisco Systems, USA
PRATEEK MITTAL, Princeton University, USA
MUNG CHIANG, Purdue University, USA

In recent years, numerous vehicular technologies, e.g., cruise control and steering assistant, have been proposed and deployed to improve the driving experience, passenger safety, and vehicle performance. Despite the existence of several novel vehicular applications in the literature, there still exists a significant gap between resources needed for a variety of vehicular (in particular, data-dominant, latency-sensitive, and computationally-heavy) applications and the capabilities of already-in-market vehicles. To address this gap, different smartphone-/Cloud-based approaches have been proposed that utilize the external computational/storage resources to enable new applications. However, their acceptance and application domain *are still very limited due to programmability, wireless connectivity, and performance limitations, along with several security/privacy concerns.*

In this paper, we present a novel reference architecture that can potentially enable rapid development of various vehicular applications while addressing shortcomings of smartphone-/Cloud-based approaches. *The architecture is formed around a core component, called SmartCore, a privacy/security-friendly programmable dongle that brings general-purpose computational and storage resources to the vehicle and hosts in-vehicle applications.* Based on the proposed architecture, we develop an application development framework for vehicles, that we call *ProCMotive*. ProCMotive enables developers to build customized vehicular applications along the Cloud-to-edge continuum, i.e., different functions of an application can be distributed across SmartCore, the user's personal devices, and the Cloud.

In order to highlight potential benefits that the framework provides, we design and develop two different vehicular applications based on ProCMotive, namely, Amber Response and Insurance Monitor. We evaluate these applications using real-world data and compare them with state-of-the-art technologies.

CCS Concepts: • **Security and privacy**; • **Human-centered computing**; • **Computer systems organization** → **Embedded and cyber-physical systems**; **Real-time systems**;

Additional Key Words and Phrases: Architecture, Application development, Cloud, Internet connectivity, Smartphone, Smart vehicles, Performance, Privacy, Programmability, Security

ACM Reference Format:
Arsalan Mosenia, Jad F. Bechara, Tao Zhang, Prateek Mittal, and Mung Chiang. 2018. ProCMotive: Bringing Programmability and Connectivity into Isolated Vehicles. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 1, Article 26 (March 2018), 31 pages. https://doi.org/10.1145/3191758

Authors' addresses: Arsalan Mosenia, arsalan@princeton.edu; Jad F. Bechara, Princeton University, Department of Electrical Engineering, Princeton University, Princeton, NJ, 08540, USA; Tao Zhang, Cisco Systems, USA; Prateek Mittal, Princeton University, Department of Electrical Engineering, Princeton University, Princeton, NJ, 08540, USA; Mung Chiang, Purdue University, Department of Electrical and Computer Engineering, West Lafayette, IN, 47907, USA.

2474-9567/2018/3-ART26
https://doi.org/10.1145/3191758

1 INTRODUCTION

Rapid technological advances in sensing, signal processing, low-power electronics, and wireless networking are revolutionizing vehicle industry. To enhance the driving experience, passenger safety, and vehicle performance, numerous vehicular technologies have been suggested and partially deployed in recent years. For example, steering assistance and cruise control have been already integrated into state-of-the-art vehicles, and vision-based collision avoidance [49, 57] and sign detection [45, 46] have shown promising results and garnered ever-increasing attention from vehicle manufacturers. However, there still exists a significant gap between resources needed for a variety of vehicular (in particular, data-dominant, latency-sensitive, and computationally-heavy) applications and the capabilities of already-in-market vehicles [41, 50].

A few vehicle manufacturers (for example, Tesla and Toyota) and several third-party companies have explored different solutions to partially address the above-mentioned gap by utilizing external computational power and storage resources provided by either the Cloud or the user's smartphone. Manufacturers have started adding built-in Cloud-based services, e.g., radio, navigation, and software updates, to their state-of-the-art products. Third-party companies have offered different dongles that can be attached to the vehicle and gather various types of data from On-board Diagnostics (OBD) port, which provides a direct access to various sensors and built-in components. Such dongles collect and transmit data (with minimal on-dongle processing) to *smartphone or the Cloud* (either directly or through the smartphone) for further processing. The majority of OBD-connected products support a single or a small set of very basic service(s), such as locking/unlocking doors or closing/opening windows. Recently, a few companies (for example, Mojio [11]) have introduced new approaches to support multiple applications using a single OBD-connected dongle. Such a dongle transmits raw data to the smartphone/Cloud and enables developers to build on-smartphone or on-Cloud applications, however, it does not offer in-vehicle processing due to resource limitations.

Despite advantages that on-smartphone or on-Cloud (either manufacturer-enabled or dongle-based) applications offer, their acceptance and application domain *are still very limited due to four fundamental reasons (see Section 2.1 for further discussions):*

1. Lack of programmability: Typically, vehicle manufacturers do not allow third-party developers to build customized vehicular applications at all or offer limited APIs, e.g., only for entertainment technologies. Vehicles currently have several embedded systems, commonly referred to as Electronic Control Units (ECUs). However, ECUs are designed for and optimized to support basic vehicular operations, such as anti-lock braking system and adaptive cruise control, and are not capable of handling customized applications.

2. Drawbacks of wireless connectivity: Vehicle-to-Cloud/smartphone connectivity is not reliable for several (e.g., safety-related) applications due to its potential unavailability and intolerable round-trip delay time. Furthermore, transmitting the huge amount of data needed for data-dominant applications, e.g., traffic sign detection is not cost-efficient.

3. In-vehicle resource limitations: Several applications must offer a real-time response, and thus, require in-vehicle processing. Dongles and built-in computing units have limited resources and cannot host computational-heavy applications. Users' smartphones may offer extra resources, however, imposing computational-heavy operations on them significantly increases their power consumption, leading to user inconvenience.

4. Public security/privacy concerns: Add-on dongles do not use strong security mechanisms due to resource constraints, and as a result, they suffer from several security attacks, e.g., the attacker can remotely disable the breaking system [10]. Furthermore, third-party dongles can transmit a variety of private information to the Cloud, and thus, their introduction has led to public privacy concerns [7, 19, 31]. For example, Elastic Pathing [31], published in Ubicomp 2014, has shed light on how insurance companies can infer the user's location by processing the vehicle's speed.

We envision *an interoperable add-on solution* (i.e., solution that imposes minimal design modification to vehicle manufacturers and third-party vehicular companies) as key to enabling a proactive approach to offer new vehicular applications. As discussed later in Section 2, an in-vehicle programmable add-on, that imposes no design change on the vehicle, can offer a holistic solution to address the above-mentioned shortcomings of previous smartphone-/Cloud-based services. In this paper, we present a novel reference architecture for vehicular application development that relies on four fundamental components: (i) SmartCore, *a privacy/security-friendly programmable* OBD-connected dongle that can host multiple applications *in the vehicle*, (ii) the user's personal devices that provide additional resource to applications running on SmartCore and/or enable the user to control them (via a graphical user interface), (iii) Cloud servers, which provide extra resources, keep application installation packages, and offer remote software update, and (iv) add-on modules that enable adding extra input/output devices and computational/storage units to SmartCore if needed. *Based on this architecture, we propose an application development framework for vehicles, called ProCMotive, which enables developers and researchers to rapidly prototype and deploy customized vehicular applications.*

SmartCore is the core component of ProCMotive and aims to partially push computational/storage resources from the Cloud to the vehicle. In particular, it:

- attaches to the OBD port and replicates a similar interface for third-party dongles. This ensures interoperability, i.e., after adding SmartCore, both the vehicle and previously-designed OBD dongles can resume their regular functionalities.
- potentially enables the development of various novel (in particular, latency-sensitive and data-dominant) vehicular third-party applications. It exploits its in-vehicle computational/storage resources to either fully host lightweight latency-sensitive applications or partially implement data-dominant and computationally-heavy applications.
- acts as a gateway for third-party OBD-connected dongles. It enables real-time monitoring of other OBD-connected dongles to detect and address any malicious activities (e.g., launching a denial-of-service attack or stealing private information) initiated by them.
- offers a context-aware access control scheme that enables the user to decide what information he wants to share with each in-vehicle application or OBD-connected dongle with respect to the current context.
- implements a set of privacy-friendly data manipulation functions that aim to minimize the amount of private information leakage by removing inessential parts of data before sharing them with third-party applications and untrusted OBD-connected dongles.

Our key contributions can be summarized as follows:

(1) We discuss fundamental shortcomings of existing OBD-based add-ons and briefly describe how the proposed approach intends to address them. Furthermore, we suggest a list of additional goals for ProCMotive and justify why each goal is desired.
(2) We present a reference architecture that comprehensively specifies the functionalities and communication capabilities of its fundamental components (SmartCore, the user's personal devices, Cloud servers, and add-on modules). This architecture has been proposed to address shortcomings of previous OBD-based approaches, while taking suggested design goals into account.
(3) Based on the reference architecture, we design and implement an application development framework that enables developers/researchers to build new vehicular applications.
(4) Using the prototype implementation of ProCMotive, we design and implement two vehicular applications, namely, Amber Response and *Insurance Monitor*, which are *either completely or partially* hosted on SmartCore.
(5) We evaluate the applications using real-world data and comprehensively compare them with the state-of-the-art technologies.

The rest of this paper is organized as follows. Section 2 describes shortcomings of previous smartphone-/Cloud-based approaches and discusses our additional design goals. Section 3 presents the reference architecture. Section 4 explains how we have designed and developed a prototype of ProCMotive based on the reference architecture. Section 5 describes two novel applications that we have proposed and implemented based on ProCMotive, evaluates them, and summarizes how real-world applications benefit from the proposed framework. Section 7 discusses the related work. Eventually, Section 8 concludes the paper.

2 DESIGN CONSIDERATIONS

In this section, we first discuss common shortcomings of previous smartphone-/Cloud-based approaches in detail and briefly discuss how ProCMotive aims to address them. Second, we describe additional design goals, that we considered while designing ProCMotive, and the rationale behind each of them.

2.1 Addressing Shortcomings of Previous Approaches

As briefly mentioned in Section 1, previous approaches have several shortcomings that limit their scope of applications and acceptance. Next, we describe these limitations in more detail and discuss how ProCMotive addresses them.

2.1.1 Lack of programmability. State-of-the-art vehicles utilize a compound of ECUs and on-board buses. They incorporate several (up to 100) ECUs, which host vehicle-specific software. ECUs provide in-vehicle resources to enables a variety of basic vehicular operations, such as anti-lock braking system and adaptive cruise control [52]. Vehicle manufactures have supported ECUs programming and tuning to enhance the vehicle performance even after its initial sale or fix software bugs if needed [5]. However, these built-in computational resources do not offer the flexibility provided by general-purpose computing units: *they cannot be easily reprogrammed to host third-party vehicular applications*. This limitation has been imposed by manufacturers to ensure the quality of service (QoS) and reliability of critical (in particular, safety-related) operations handled by ECUs. Therefore, despite the existence of in-vehicle computational resources, utilizing them to implement customized vehicular applications is neither simple nor recommended. Some manufacturers have started offering APIs to application developers, however, these APIs are very limited and only target a small application domain, in particular entertainment applications.

2.1.2 Drawbacks of wireless connectivity. Here, we discuss the issues associated with the use of vehicle-to-Cloud and vehicle-to-smartphone wireless connectivity.
1. Unavailability of wireless connectivity: Using cellular connectivity to transmit the data from the vehicle to the Cloud will result in the unavailability of the Cloud-based services when the cellular connectivity is not available, for example, when the vehicle goes through a tunnel. Furthermore, if a dongle uses the smartphone to transmit the data to the Cloud, both vehicle-to-smartphone and vehicle-to-Cloud communications become unavailable when the smartphone dies. These will result in the interruption of vehicular services, and significantly limit their applicability. In particular, safety-related applications that need a *reliable continuous stream* of data, e.g., collision detection and security attack detection, cannot completely rely on wireless connectivity. Thus, such applications should be implemented (at least partially) on the vehicle itself.

2. Intolerable response time: Several vehicular applications need real-time decision making, for example, traffic sign detection and collision prediction, and therefore, may not tolerate the response time offered by the Cloud (i.e., time needed for transmitting the data from the vehicle to the Cloud, processing them on the Cloud, and getting the response back from the Cloud). Such applications should be implemented using close-to-the-vehicle computational/storage resources.

3. Limited cellular data and bandwidth: Data-dominant applications (in particular, image processing-based collision detection or a sign detection algorithm) capture and process a huge amount of data (up to tens of GBs of data every day). For such applications, transmitting the raw data to the Cloud is not cost-efficient as demonstrated later in Section 5.1.2. Moreover, if each data chunk is huge (e.g., a high-resolution image collected from an on-vehicles camera), sending it to the Cloud or the user's smartphone may be time consuming, leading to an intolerable end-to-end application response time.

2.1.3 *Public security/privacy concerns.*

Here, we discuss security and privacy concerns of previously-proposed approaches.

1. Security threats: Vehicles are interesting targets for attackers due to their mission-critical operations. Any security attack against vehicles, especially large-scale attacks, may lead to life-threatening consequences. As further discussed later in Section 5.2, *the federally-mandated OBD port offers an unprotected standard interface* that can be exploited by attackers to take the control of mission-critical components, e.g., braking system. It has been shown that attackers can launch a multitude of well-known security attacks against the vehicles using OBD port, ranging from Denial of Service (DoS) attacks to packet sniffing [38]. Several already-in-market OBD dongles are vulnerable to well-known security attacks, offering a valuable opportunity for attackers to remotely take the control of several components embedded in the vehicle [19].

2. Private information leakage: Since OBD interface offers a full access to all OBD-connected dongles, they can potentially collect a variety of sensitive information (e.g., sensory readings, GPS coordinates, and the vehicle's identification number and model) without the user's permission, leading to several privacy concerns [31, 47]. It has been shown that an insurance company can potentially track all user movements (and also extract several locations of interest) using the vehicle's speed collected from OBD port [31].

2.1.4 *In-vehicle resource limitations.*

Several applications (for example, both applications discussed in Section 5) must offer real-time (or near real-time) responses, and thus, require in-vehicle processing. As discussed earlier in Section 2.1, currently-in-use after-market dongles have very limited resources and cannot host customized applications in the vehicle. Furthermore, built-in computing units (for example, ECUs) embedded in already-in-market vehicles neither provide programmability nor high performance, and thus, cannot host (and tolerate the overhead of) in-vehicle customized applications. Moreover, relying on users' personal devices may offer extra resources, however, imposing computational-heavy operations on them may significantly increases their power consumption, leading to user inconvenience.

2.2 Additional Design Goals

2.2.1 *Interoperability.*

As mentioned in Section 1, SmartCore acts as a gateway for other third-party OBD-connected dongles. It connects to the OBD port and replicates a similar interface that can be used by other third-party OBD-based devices, e.g., insurance dongles. Such devices can send their requests to the OBD port *only through SmartCore*, while SmartCore is actively enforcing appropriate security and privacy policies. This ensures interoperability: if an OBD-based dongle complies with the policies, it can perform its regular operations without any design modification, despite the attachment of SmartCore to the OBD. Interoperability is essential to minimize the additional costs associated with the use of ProCMotive and maximize its acceptance.

2.2.2 *Extensibility.*

It is desired to implement ProCMotive so that its application domain can be extended in future with minimal design modifications. SmartCore offers short-range wireless connectivity (Bluetooth and WiFi), along with several Universal Serial Bus (USB) ports, so that additional input/output, storage, and computing devices can be easily added to the architecture if needed. For example, developers can process the sensory data gathered from the vehicle, along with data collected using add-on input devices (e.g., a camera), to design and develop novel vehicular applications.

2.2.3 Virtualization. Virtualization, i.e., creating multiple isolated containers to host different applications on the same operating system (OS), is highly desired to ensure the security. A container is an abstraction at the application layer that maintains application packages (i.e., codes and their dependencies). SmartCore hosts several third-party applications at the same time, and it uses a separate container for each application. This ensures that an application can neither see nor affect applications running in other containers. Moreover, each container has its own network stack, and therefore, it does not have privileged access to the sockets or interfaces of another container [42].

2.2.4 Remote update. To enhance the performance and security and provide regular fixes for features that are not working as intended, it is essential to offer remote software update. To ensure user convenience, an over-the-Internet remote update is highly desired. In ProCMotive, Cloud servers will host software updates (e.g., the latest version of applications, middleware, and OS) and regularly inform the user if a new update is available.

3 THE REFERENCE ARCHITECTURE

In this section, we present an architectural overview of ProCMotive. The proposed architecture is motivated by the insight that close-to-the-user computation can open up new opportunities *for addressing various security/privacy concerns associated with the use of Internet-connected vehicles, enhancing the performance of vehicular applications, and enabling new applications that were not feasible before using previous architectures.* Fig. 1 presents the proposed reference architecture. As illustrated in this figure, ProCMotive consists of four main components, namely SmartCore, personal devices, Cloud servers, and add-on modules. These components can communicate with each other via various communication channels. To ensure security, in this architecture, all communication channels (expect OBD-based channels that are implemented based on a federally-mandated guideline) can be encrypted. Next, we describe different components of the reference architecture.

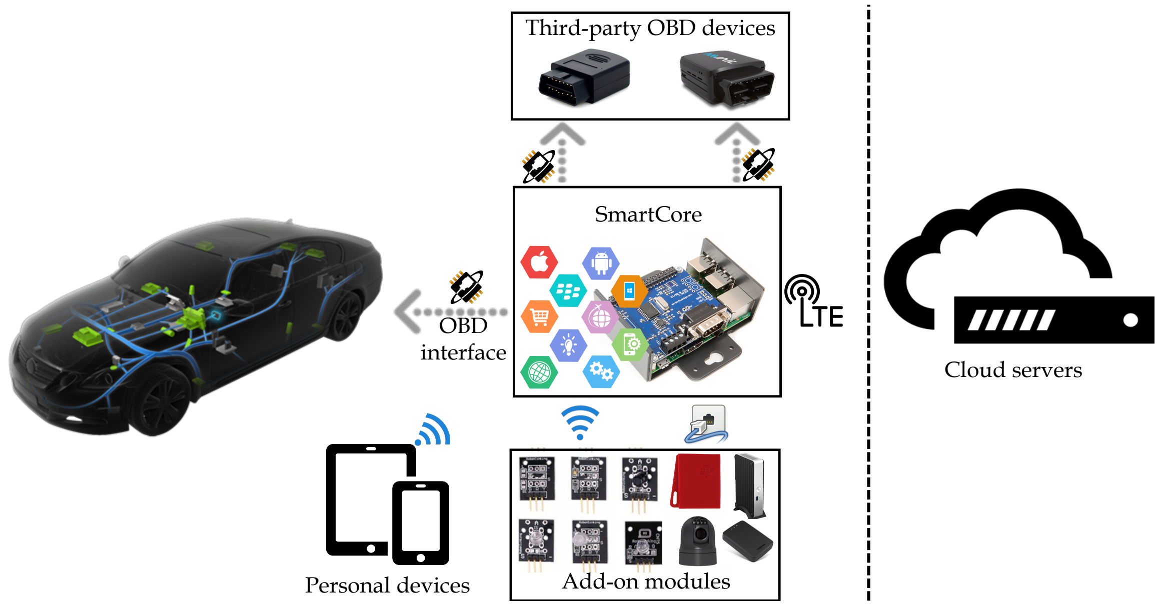

Fig. 1. An architectural overview of ProCMotive: it consists of SmartCore, personal devices, Cloud servers, and add-on modules. To ensure security, in this architecture, all communication channels (expect OBD-based links) should be encrypted.

3.1 SmartCore

SmartCore is an OBD-connected dongle that brings sufficient computational power and storage capacity to the vehicle to support several fundamental operations. In the proposed architecture, SmartCore is connected to the OBD port for two main reasons. First, it can access various types of sensory (e.g., coolant temperature, engine RPM, ambient temperature), and non-sensory data (e.g., GPS coordinates, the vehicle's make and model) from the vehicle. Second, it can be powered through this port by accessing the vehicle's battery. Next, we list and briefly describe the fundamental operations of SmartCore.

3.1.1 Data collection. SmartCore can collect data needed for various vehicular applications from two main sources: the sensors embedded in the vehicle and add-on sensors. OBD interface enables the SmartCore to access various built-in components including sensors. SmartCore can request different sensory data by sending their corresponding diagnostics parameter IDs (PIDs), which are supported by vehicle manufacturers to facilitate diagnostics. Moreover, add-on input sensors can be connected to SmartCore over WiFi or Bluetooth, providing additional information about the environment.

3.1.2 On-vehicle data processing. SmartCore has sufficient resources to perform a wide range of data processing (in particular, privacy-enhancing, data compression, and data analytics) algorithms in the vehicle. Depending on the available resources, performance requirements, and QoS guarantees, applications can be partially/fully implemented on SmartCore. In-vehicle processing opens up a new opportunity for developing several new applications. For example, consider a sign detection algorithm that aims to recognize the traffic signs by processing the images captured from the environment. If the vehicle manufacturer does not support this application by default, incorporating it into already-in-market vehicles is not feasible due to the shortcomings of previously-proposed architectures as described in Section 2. However, SmartCore enables in-vehicle image processing for such an application, minimizing the vehicle-to-Cloud transmission overhead and offering short response time. Moreover, as demonstrated later in Section 5, it can be used to implement privacy-enhancing algorithms that remove inessential portions of raw data (e.g., the whole image) before transmitting it to the Cloud or sharing it with other OBD-connected devices, e.g., insurance dongles.

3.1.3 Access control. While various access control schemes have been proposed for personal devices (smartphones and tablets) [25, 27, 44], they have been neither well-established nor well-studied in the domain of Internet-connected vehicles and vehicular applications. The OBD protocol itself does not offer any access control solution to specify when, where, and to what extent the sensitive data can be gathered from the OBD. In order to prevent forming a monopoly in the auto repair business, vehicles manufacturers are mandated by law to provide full access to built-in components via OBD port. Although OBD-connected dongles can access various sensors and components embedded in the vehicle to enable new vehicular applications, their usage can lead to serious security and privacy concerns if their access level is not limited. SmartCore offers an access control scheme to limit the access level of (i) applications hosted on the SmartCore and (ii) third-party OBD-based dongles. It continuously monitors the behaviors of hosted applications and third-party dongles and ensures that they comply with a set of access control policies. SmartCore supports two types of policies: predefined policies and context-aware user-defined policies, as described next.
Predefined policies: Upon the installation of an application or the attachment of a new dongle to SmartCore, a set of predefined policies are assigned to the application/dongle. These set of policies are determined based on two parameters: the vehicle's specifications (e.g., vehicles' manufacturer, make, and model) and the specification of the application/dongle (e.g., the application's intentions or the manufacturer/model of the dongle). The vehicle's specification can be directly obtained from the OBD port. It is used to take the vehicle's manufacturer-reported OBD issues and specific characteristics into account. For example, the attachment of any OBD dongle to a

Ferrari 430 will disable its Traction Control System [4]. Thus, for this vehicle, the default access level of applications/dongles should ensure that the OBD port cannot be accessed when the car is moving. The specification of the application/dongle is used to determine its expected access level based on its intended operations. For example, it is sufficient for an insurance dongle to only access a subset of the vehicle's sensors, e.g., the speedometer and odometer.

Context-aware user-defined policies: Although predefined policies provide basic protection against different security/privacy attacks, it is unlikely that their privacy and security implications would be fully understood by regular users. Indeed, users commonly under-/over-estimate the level of protection that these policies provide [44]. To take users' preferences into account, we included a domain-specific context-aware access control scheme in SmartCore. Context-aware user-defined policies offer the potential to correctly reflect the user's security/privacy preferences. However, if it is not user-friendly, the amount of essential user effort needed to initialize, modify, and maintain a comprehensive set of context-dependent policies is high [44]. A rich set of contexts enables the user to define fine-grained policies, however, it is well-known that regular users are not willing to spend significant amounts of time to adjust the policies with their preferences. In addition, it is questionable, whether users are capable of understanding the implications of their policy settings [44]. Thus, it is desired that SmartCore offers a user-friendly policy managements system, while enabling users to define/modify various policies with respect to a rich set of domain-specific high-level contexts (e.g., whether the vehicle is involved in an accident).

3.1.4 OBD port management. In order to enable real-time monitoring of other OBD-based third-party dongles, such dongles are only allowed to indirectly access OBD port through the interface implemented by SmartCore. SmartCore should isolate third-party OBD-based dongles (i.e., they can neither directly communicate with the vehicle's OBD port nor see each other), monitor and process commands/message initiated from them, and responds to such commands/message (with respect to certain security/privacy policies specified by the access control scheme). Port management enables this isolation and handles requests sent to or received from the vehicle's OBD port.

3.2 Personal Devices

Modern personal devices (e.g., smartphones and tablets) have become a vital part of our everyday lives. They are equipped with many compact built-in sensors (e.g., accelerometers, magnetometers, and barometers), various communication capabilities (e.g., WiFi, LTE, and Bluetooth), powerful microprocessors, and high-volume storage in order to support a variety of applications [47]. In addition to enabling such applications, their spare resources *can be potentially utilized to offer additional resources (e.g., computational power) and inputs/outputs (e.g., sensors) to applications running on SmartCore.* For example, if a developer wants to build an automatic headlight control application (i.e., an application that can automatically turn on the vehicle's headlights based on the existing ambient light) on a vehicle that does not support this functionality. He can potentially utilize the ambient light sensor embedded in the user's smartphone to sense the ambient light and then launch appropriate controlling commands to control the vehicle's headlight using SmartCore. Furthermore, its display can offer a user-friendly interface, which can be used to control various functionalities of SmartCore, as further described later in Section 4.1.3.

3.3 Cloud Servers

In the proposed architecture, Cloud servers are envisioned to have three fundamental responsibilities: (i) maintaining application packages, (ii) offering additional computational/storage resources that can be used to partially/fully implement an application on the Cloud, and (iii) enabling remote update of the software, in particular, applications installed on SmartCore, and the underlying middleware and/or OS.

3.4 Add-on Modules

These modules are either additional input/output devices, including, but not limited to, cameras, sensors, GPS receivers, microphones, voice assistant systems, and computing/storage units, that can be connected to SmartCore via WiFi, Bluetooth, or wired connectivity. For example, a vehicle-mounted camera can gather valuable information about the vehicle's surroundings, enabling a variety of image processing-based applications. In Section 5, we discuss an instance of novel applications that can be enabled using an add-on module, a vehicle-mounted camera. *In the proposed architecture, add-on modules offer extensibility.*

4 PROTOTYPE IMPLEMENTATION

Based on the proposed reference architecture and intended operations of its components, we designed and developed a prototype for ProCMotive. The prototype implementation offers a framework that provides the backbone for vehicular application development, considering various domain-specific challenges: programmability, wireless connectivity, and performance shortcomings, along with security/privacy concerns. Next, we discuss the implemented software components and underlying hardware/infrastructure.

4.1 Software Components

The prototype implementation consists of three main software components: (i) CARWare a middleware (i.e., a software that acts as a bridge between the native OS and applications) that enables SmartCore's intended operations, (ii) an Android application that enables the user to control different vehicular applications and manage access control policies using his smartphone, and (iii) a trusted web server on the Cloud that enables the user to download/update vehicular application packages. Next, we discuss these components in detail.

4.1.1 CARWare: The core middleware. As the core of reference architecture, SmartCore offers several functionalities. We have designed and implemented CARWare to enable the intended functionalities of SmartCore discussed earlier in Section 3. CARWare comes between the native OS (Raspbian [17] in our prototype) and the application layer (Fig. 2). It enables remote update, data collection (from both OBD port and add-on inputs), application management, OBD port management, and access control and provides a RESTFul API through which it handles various requests created by applications and returns a response in JSON format (i.e., an open-standard file format that uses human-readable text). Next, we describe the supported functionalities of CARWare and briefly describe the APIs that it provides.

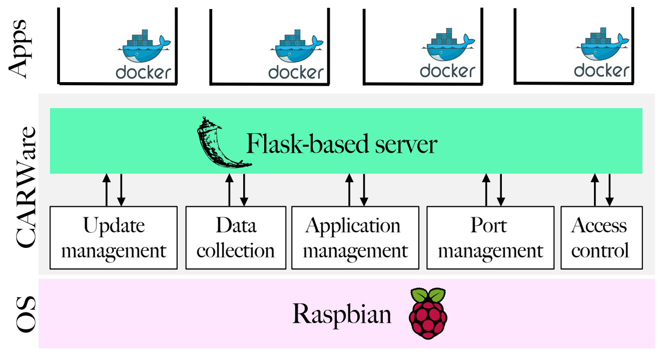

Fig. 2. CARWare: The middleware comes between the OS and the application layer and provides various APIs for update management, data collection, application management, port management, and access control, which can be accessed through its Flask-based web server.

Update management: The update management (one of five core software libraries in CARWare as shown in Fig. 2) provides various functions needed for remotely updating application packages and the CARWare. In the current implementation of CARWare, we have used the full functionally of the API to develop a *trusted Android application (with administrative privileges)* that can download, install, and run the last version of application packages and CARWare stored on the Cloud if an update is necessary. *Although the update management library enables updating applications and the middleware, for security reasons, we do not allow a regular application (i.e., an application without administrative privileges) to neither update other applications nor CARWare: a third-party application, that is not given the administrative password, can only update itself.* Furthermore, to ensure the safety of the driver and the passenger, the update management API only accepts requests when the vehicle is not moving: the update management creates an OBD request with the PID of $0D$, asking for the current vehicle's speed. If the vehicle's speed is zero it processes the request; otherwise the request is dropped. Next, we briefly describe how the update management component processes request for updating CARWare and applications.

Vehicular applications: In the current prototype, two applications can update a specific vehicular application: (i) the vehicular application itself or (ii) an application with administrative privileges (see Section 4.1.3 for more detail on our Android application that provides a user interface for updating applications). In fact, an update request must provide the application identifier along with either the application token or the administrative password (that the user sets in the initial setup of SmartCore)[1]. If the provided credentials are correct, then the request will be processed as follows. First, the latest version of the application will be requested from *Vehicular Application Store*. Upon receiving a request from CARWare, *Vehicular Application Store* provides the latest version of the application package (a Docker image [42] including the application and its dependencies along with a JSON file describing the application and its requirements and requested sensors/resources). Then, if the safety requirement is met (the car is not moving), the update management first pauses the container that runs the old version of the vehicular application and then runs a new container using the latest version of the application.

CARWare: In the current implementation of ProCMotive, update management API only allows an application with administrative privileges (e.g., our trusted Android application described in Section 4.1.3) to update the core middleware. To update CARWare, an update request [2] must be created and sent to the web server running in CARWare, which forwards the request to the update management component. If the provided credentials are correct and the vehicle is not moving, an external script (running on the underlying OS) will download the latest version of CARWare, pause all application containers, install the new version of CARWare, kill/remove the previous version and run the new version, and eventually resume the paused containers.

Data collection: CARWare offers an API that can be used to facilitate data collection from the vehicle's sensors and SmartCore's add-on inputs. The current implementation of the API enables accessing over 30 types of the sensory data from the vehicle, along with different types of data provided by three add-on inputs: a vehicle-mounted camera (Raspberry Camera Module V2 [16]), a set of sensors (TI Sensor Tag [20] that has built-in accelerometer, magnetometer, temperature, and air pressure sensors and is connected to SmartCore via Bluetooth Low Energy), and a GPS receiver (USGlobalSat GPS receiver [9]). In order to fetch sensory data, the vehicular application, that runs in a container, communicates with the web server running within CARWare. For data collection, the application creates a request including its unique credentials (an application identifier and a token) along with the description of data that are needed (for each data type, the request includes two fields: source of data, i.e.,

[1]An application update request can be in one of the following formats: (i) {"appID": "<application Identifier>", "appToken": <Application Token>, "requestType": "update", "updateTarget": <Application identifier>} or (ii) {"appID": "ROOT", "appToken": <Administrative Password>, "requestType":"update", "updateTarget": <Application identifier>}

[2]CARWare can be updated by sending the following request: {"appID": "ROOT", "appToken": <Administrative Password>, "requestType":"update", "updateTarget": "CARWARE"}

from the 'vehicle' or 'add-on' inputs, and type of data, e.g., acceleration or the vehicle's speed) [3] . Upon the arrival of a request, CARWare first checks if the request complies with access control policies. If so, it reaches the requested data source, collects the data, and returns a response including the data (in JSON format) to the application. Otherwise, it rejects the request.

Application management: In-vehicle application hosting requires fine-grained application management. Application management enables the user to (use an Android application developed based on its API and) download a vehicular application from Cloud server to the SmartCore, run the application (inside a container separated from other applications), pause all processes involved in the application, and completely halt the application. Using containers allows independent isolated applications to run simultaneously within a single OS, avoiding the overhead of starting and maintaining several virtual machines. In our prototype, we utilize Docker technology [42] to create our isolated containers. Docker is one of the world's leading software container platform that facilities the repetitive tasks of building containers and configuring development environments [23].

Port management: CARWare offers an API to enable applications to control the OBD port if needed. In particular, it provides four functions: port block, rate adjustment, probing a dongle, and sending a request. Using port management, an application can (i) block all requests coming from an OBD-connected dongle (given its unique identifier), (ii) set the maximum expected rate of OBD requests initiated from an application/dongle, (iii) capture/monitor all the packets initiated from a dongle, and (iv) create and transmit an arbitrary OBD request. As demonstrated later in Section 5, using this API, developers can easily design security/privacy protection applications, which can take the control of OBD port upon detection of a malicious activity.

Access control: The access control component consists of three subsystems: policy enforcement, context recognition, and policy management that closely collaborate with each other (Fig. 3):

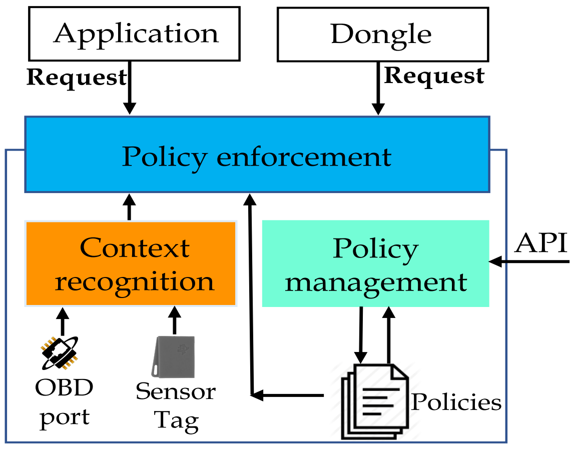

Fig. 3. The access control component consists of three subsystems: policy management, policy enforcement, and context recognition

1. Policy enforcement: It ensures that all applications and third-party OBD dongles always comply with both predefined and user-defined policies. For each request generated from an application or a dongle, if the request is

[3]A data collection request has the following format: [{"appID": "<application Identifier>", "appToken": "<Application Token>", "requestType": "dataCollection" },{"source": "<vehicle, sensorTag, camera, or GPS>", "type": "<over 30 types are currently supported, such as vehicle_speed>"}]

authorized, it lets the request to proceed; otherwise, it blocks the request, i.e., the request is neither processed by SmartCore nor forwarded to the OBD port.
2. Context recognition: Context recognition supports a list of contextual information (it continuously detects the current contexts), enabling the user to set his preferences with respect to this information. Next, we describe different type of contextual information supported in the prototype.
Operational contexts: In the prototype, the context recognition supports two contextual information related to the operation of the vehicle: *vehicle status and health status*. It continuously detects whether the vehicle is idle or moving. This allows the user to limit applications/dongles based on the current status of the vehicle. For example, the user can set a policy to disable all diagnostic OBD dongles (that may send safety-critical commands to the vehicle) when the vehicle is moving. This can potentially prevent several life-threatening security attacks (e.g., disabling the braking system [10]) even if the dongle is hacked and can be controlled by a remote attacker. Moreover, it detects the engine's health status (e.g., checks whether the check engine light is on/off). Several insurance companies, e.g., MetroMile [12], offer dongles that are also able to read all in-vehicle data, find fault codes, and describe how the user can address the issue. By setting policies based on the health status, the user can allow such dongles to only access diagnostic data upon the appearance of a fault.
Situational contexts: In the prototype, two types of situational contexts are defined: *involvement in an emergency and the presence of an external alert message*. Context recognition frequently collects data from sensors embedded in TI Sensor Tag [20], e.g., accelerometers, and on-vehicle sensors, e.g., ABS and airbag sensors, to detect the occurrence of a collision. Moreover, it listens to a trusted communication channel through which trusted alert messages (e.g., from law enforcement) are transmitted to the vehicle. This enables the user to set situational policies. For example, a user may be willing to give an insurance dongle the permission to transmit the location of an accident to the company. Similarly, he may want to allow emergency responders to access his location information when it is asked by a law enforcement agency (see Section 5 for an application, called Amber Alert, that needs this type of permission).
Location-based contexts: There are different scenarios in which the user might like to control the access level of the application/dongle with respect to the current location of the vehicle. For example, the user may be willing to share some information with applications/dongles only when he is in trusted locations. Furthermore, he might want to stop sharing his sensitive information (e.g., GPS coordinates) when he is in specific locations (e.g., his home or office). The current prototype implementation lets the user to set policies with respect to a set of locations of interest: user-defined locations (home and office addresses) and manufacturer-trusted addresses (locations of trusted auto repair shops).

3. Policy management: This subsystem is responsible for getting the feedback from the user and enabling the user to enforce his security/privacy preferences. It provides an API which can be used to add, edit, and remove access control policies for each application or third-party dongle attached to SmartCore [4].

4.1.2 Cloud server. In the prototype, ProCMotive has a trusted Cloud-based web service, called *Vehicular Application Store* that is written in Python based on Flask framework [32] and is hosted on Amazon Web Services (Model t2.2xlarge [1]). It offers an API that can be used to: (i) list available vehicular application packages, and (ii) download the last (or a specific) version of an application files (a Docker image [42] including the application and its dependencies along with a JSON file describing the application and its requirements and requested sensors/resources). Using the API provided by this sever and the API provided by the update management

[4]For example, a predefined policy can be set/modified by sending the following request to CARWare: [{"appID": "ROOT", "appToken": "<Administrative Password>", "requestType":"policyManagement", "targetID": "App_1"} , {"source": "vehicle", "type": "vehicle_speed", "policyType": "strict", "access": "always" }]. This specifies that the vehicle's speed can be always shared with an application with the identifier of "APP_1"

unit of SmartCore, we developed and Android application that allows the user to download, install, and update applications/CARWare, as described later in Section 4.1.3.
Note: As described earlier, the Cloud has a vital role in the implementation of various vehicular applications: it can provide extra computational/storage resources for each application. While designing different vehicular applications based on ProCMotive, we have also used additional resources of the Cloud (see Section 5 for more detail, where we describe two vehicular applications developed based on ProCMotive).

4.1.3 *Android application.*

Using the APIs provided by Vehicular Application Store and SmartCore, we have implemented an Android application that communicates with both the store and SmartCore and offers a user-friendly interface for managing access control policies and vehicular applications and updating CARWare/applications:
Access control management: The user can set, modify, delete user-defined access control policies. For each vehicular application or OBD dongle, CARWare maintains an access control file (in JSON format) that specifies its access control policies. The smartphone communicates with the web server within CARWare to reflect user preferences and update the access control files.
Application management: To install and run a new application, the user can select one application from the list of available vehicular applications stored on Vehicular Application Store. When the user intends to download and run an application on SmartCore, his request is sent from the smartphone to the web server within CARWare and is handled as follows: the server communicates with Vehicular Application Store and fetches the application package, it then runs the application in an isolated container. Using the smartphone application, the user can check the status of all vehicular applications running on SmartCore and manage them (pause, halt, and remove their containers) if needed.
Application/CARWare update: The smartphone application first sends a request to SmartCore and fetches the descriptions of applications/CARWare currently installed on SmartCore. Then, the user can select one vehicular application (or CARWare) from the list of available applications installed on the SmartCore to be updated. Upon selection of a vehicular application (or CARWare), the smartphone application sends an update request including the application identifier and administrative password to the web server running inside the CARWare. Afterwards, SmartCore communicates with Vehicular Application Store and downloads the latest version of the application package. Eventually, if the safety requirement is met (the car is not moving), CARWare updates the vehicular application (or itself) as described earlier in Section 4.1.1.

4.2 The Underlying Hardware/Infrastructure

SmartCore is implemented based on Raspberry Pi 3 that comes with Raspbian, its native Debian-based computer OS [17]. SmartCore also utilizes two other hardware components: NETGEAR 4G LTE Modem (Model LB1120) [14] with a T-Mobile Prepaid Plan [21] that enables wireless connectivity over LTE and OBD PiCAN 2 board [15] that provides CAN-Bus capability for the Raspberry Pi. It currently offers three wireless channels: LTE, Bluetooth Low Energy, and WiFi. Raspberry Pi has built-in cryptographic modules that support strong encryption for these communication channels. Thus, to ensure security, all communication channels to/from SmartCore can be encrypted (except the wired OBD-based communication channels).

Furthermore, SmartCore currently supports three add-on modules: (i) TI Sensor Tag [20], a Bluetooth-enabled sensory unit that includes various sensors such as accelerometer, magnetometer, and air pressure, (ii) Raspberry Camera V2 [16], a vehicle-mounted camera that can capture video frames from the environment, and (iii) USGlobalSat GPS [9] that is a GPS receiver.

We have used a Nexus 5S to test all Android applications developed in this paper and utilized Amazon Web Services (Model t2.2xlarge [1]) as the Cloud.

5 APPLICATIONS

In this section, we propose two novel applications, which are implemented based on ProCMotive. We evaluate these applications using real-world data and discuss how they benefit from in-vehicle processing (SmartCore).

5.1 Application 1: Amber Response

In this section, we discuss a novel application that has been enabled by ProCMotive, which we call Amber Response.

In the U.S., an Amber Alert is activated when a law enforcement agency has admissible reasons to believe that a child has been abducted and he is in danger of serious life-threatening conditions or death. The Amber Alert system relies on the nearby people to get information about the abduction. It informs the public about the abduction by broadcasting the *make, model, color, and plate number* of the abductor's vehicle to nearby smartphones, enabling the entire community to assist in the safe recovery of the child. It has been shown that this scheme is only slightly effective and may cause user inconvenience (for example, the alert will be sent to all nearby people even if they are not walking/driving and cannot provide useful information). Since the inception of the program in 1996 through 2015, around 43 children, on average, have been safely recovered every year specifically as a result of an AMBER Alert being issued, whereas the average number of abduction in the U.S. is around 800,000 every year (see [2] for detailed annual statistics). Thus, a more effective alternative system is highly needed. We propose such a system, called Amber Response, and implement it using ProCMotive. Amber Response utilizes a vehicle-mounted camera, that continuously captures several frames per second from the environment, and processes image frames to automatically find the abductor's vehicle (given the database of active Amber Alerts). Different functions of this application can be distributed across SmartCore and the Cloud, as described next.

5.1.1 Prototype implementation. Amber Response application maintains a database of active Amber Alerts, including make, model, color, and plate number of abductors' vehicles. This database is located on the Cloud server and can be updated by responsible agencies. The application searches through the video frames to find a vehicle whose features match the ones of a record in the database. Upon the detection of a suspicious vehicle, the application sends the vehicle's GPS coordinates to the Cloud server, informing law enforcement agencies. Using ProCMotive, we have implemented three different versions of Amber Response: (i) a Cloud-based, (ii) a SmartCore-based, and (iii) a hybrid version that exploits both Cloud and in-vehicle computation/storage resources. We next describe how we have implemented these three versions.

Cloud-based: In this version, SmartCore only collects the data (video frames) and uploads them to the Cloud without modification. After uploading the frames, an on-Cloud server receives and processes them to find a plate number that matches the plate number of a suspicious vehicle in the database. We have utilized OpenALPR library [3] to implement plate detection algorithm on the Cloud. Plate detection algorithm has eight main steps, which are briefly described in Table 1. Implementing the Cloud-based version of Amber Response was potentially feasible using previously-proposed Cloud-based architectures that rely on minimal computational power in the vehicle, and thus, the Cloud-based version can be used as a baseline to compare ProCMotive-enabled (SmartCore-based and Hybrid) implementations with previously-presented Cloud-based proposals for connected vehicles (e.g., Azure-based connected vehicles [13]). As described later in Section 5.1.2, despite demonstrating a promising performance, the Cloud-based implementation cannot be utilized in real-world scenarios due to the cost overhead associated with transmitting the data needed for this implementation.

SmartCore-based: In this version, the application installed on SmartCore frequently (e.g., every 30 seconds) fetches the database of active Amber Alerts to ensure that it maintains the last updated version of the database. It then captures video frames from the camera and runs the plate detection algorithm described above. After

Table 1. Different steps of plate detection algorithm [3]

Step	Description
1. Plate detection	Finds potential license plate regions
2. Binarization	Converts the plate image into black and white
3. Char Analysis	Finds character-sized "blobs" in the plate region
4. Plate Edges	Finds the edges/shape of the plate
5. Deskew	Transforms the perspective to a straight-on view
6. Segmentation	Isolates and cleans up the characters
7. Char Recognition	Analyzes each character image
8. Post Process	Creates a top N list of plate possibilities

extracting all plate numbers from the frames, it searches through the database to find a match and sends a report, including the vehicle's GPS coordinates, to the Cloud if a match is found.
Hybrid: The hybrid implementation exploits both in-vehicle and on-Cloud resources. In this scenario, Amber Response application has been partially implemented on SmartCore. On SmartCore, it first captures the frames from the camera. Then, it performs a lightweight image processing function to extract all plate areas in each frame (Step 1 in Table 1). Afterwards, for each vehicle in the frame, it estimates the vehicle's color from a small area above its plate (whose size is %10 of the detected plate's area). If the vehicle's color matches the color of one of the suspicious vehicles reported in the database, it transmits the corresponding plate area to the Cloud for further processing. The on-Cloud side of the application, receives and processes the images that only contain the area. Upon detection of a suspicious vehicle, it sends a request to the application installed on SmartCore, asks for current location of the vehicle, and informs the law enforcement agency.

5.1.2 Evaluation. Next, we first describe the dataset used to evaluate Amber Response, and then examine and compare different implementations of Amber Response from three perspectives: (i) performance, (ii) cellular data usage, and (iii) privacy leakage.
Dataset: We downloaded 12 videos uploaded on YouTube that were captured using a camera mounted behind the mirror of a moving vehicle. These videos have the resolution of at least 1080p and frame rate of at least 10 frames per second (FPS). To construct our dataset, we created 72 videos by varying both resolution and frame rate of the downloaded videos. For each original video, the dataset includes six videos with different resolutions, i.e., 1080p and 720p, and frame rates, i.e., 1, 5, and 10FPS. Each video is about 10 minutes and the suspicious vehicle, i.e., the vehicle that Amber Response aims to find, appears in a random time in the footage (i.e., for each video, we randomly choose a single vehicle and add its specifications to the database of suspicious vehicles stored on the Cloud). Based on our empirical results, Amber Response can accurately (with the accuracy of %100) detect the abductor's vehicle when the frame rate is equal to or greater than 1FPS. Therefore, in our evaluations, the minimum frame rate is set to 1FPS.
Comparison: We first quantitatively evaluate three implementations of Amber Response (Cloud-based, SmartCore-based, and Hybrid) using the above-mentioned dataset. Then, we briefly describe how in-vehicle data processing can enhance the user's privacy in this application.
1. Performance: In order to compare the performance of the three implementations, we define and report Detection Time Ratio (i.e., $DTR = \frac{T_{detection}}{T_{appearance}}$) that represents how much time each implementation takes for processing one second of the video until it finds the abductor's vehicle. This metric enables us to estimate the delay in the detection of the abductor's vehicles in real-world scenarios: if the suspicious vehicle appears after $T_{appearance}$ seconds (from when the camera starts capturing the video), Amber Response detects it after $T_{appearance} * DTR$.

Indeed, it reports the suspicious vehicle with a delay of $T_{appearance} * DTR - T_{appearance}$ seconds to the law enforcement agency. For each implementation, the DTR highly depends on frame rate and resolution of the video, and how much processing power is provided by SmartCore. Next, we examine how average DTR changes with respect to these parameters.

Experimental scenario 1: In order to evaluate how DTR changes with respect to the frame rate, we examined Amber Response using a subset of the videos (36 videos) in the dataset that have the same resolution (1080p) and we ensured that SmartCore provides similar processing power for all these videos by manually enforcing its CPU to work at 600MhZ (on Raspbian [17], this can be done by editing "config.txt" located at "/boot/config.txt"). Fig. 4 demonstrates how DTR changes with respect to the frame rate for this experimental scenario. As the frame rate increases (and consequently, the image processing algorithm becomes more computationally-heavy), utilizing the Cloud for processing becomes more reasonable from performance perspective, whereas when the frame rate is low (1FPS) all three implementations become similar from performance perspective even though the computational power of SmartCore is much less than that of the Cloud.

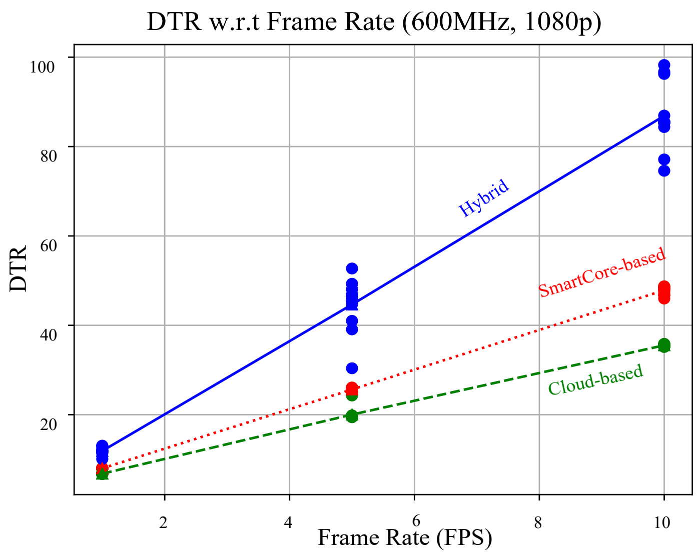

Fig. 4. DTR with respect to the frame rate: each point represents DTR for a single video from the dataset (lines show how average DTR changes with respect to the frame rate). DTR represents how much time each implementation needs for processing one second of the video until it finds the abductor's vehicle.

Experimental scenario 2: Furthermore, to evaluate how DTR changes with respect to the resolution of the video, we have repeated a similar experiment: we manually enforced the CPU to work at 600Mhz and used the videos with 1FPS. Table 2 summarizes the results of this experiment. As expected, for each implementation, a lower resolution video offers a better DTR.

A notable observation: As shown in Fig. 4, in the first experimental scenario, both Cloud-based and SmartCore-based implementations outperformed the hybrid one. For hybrid implementation, $T_{detection}$ highly depends on by how much time (i) on-SmartCore processing, (ii) SmartCore-to-Cloud data transmission, and (iii) on-Cloud processing take. For videos with the resolution of 1080p, the hybrid version of Amber Alert spends a significant amount of time for on-SmartCore processing (Step 1 from Table 1 and color detection), and therefore, it is slower than both SmartCore-based one (for which SmartCore-to-Cloud data transmission and on-Cloud processing times are zero) and Cloud-based one (for which on-SmartCore processing time is negligible). *However, when the resolution of input videos is changed to 720p, the hybrid implementation outperformed the SmartCore-based one (Table 2).* In this experimental scenario, for the hybrid implementation, on-SmartCore processing takes significantly less time (compared to when the resolution of the input video is 1080p) so that it is reasonable to shift several steps of the plate detection algorithm (Step 2 to Step 8 from Table 1) to the Cloud despite the additional time overhead associated with SmartCore-to-Cloud data transmissions.

Table 2. DTR with respect to the resolution (600 MhZ, 1FPS)

Resolution	Cloud-based	SmartCore-based	Hybrid
1080p	6.8	8.0	11.88
720p	4.4	5.7	5.57

Experimental scenario 3: Eventually, in order to examine how the performance of different implementations may vary with changes in computational power, in another experimental scenario, we have manually overclocked SmartCore's CPU to run at $1200MhZ$ and repeated our examination using a subset of videos (with the resolution of 720p and frame rate of 1FPS). Table 3 summarizes the results of this experiment. In this experimental scenario, where the computational power of SmartCore is significantly increased, SmartCore-based implementation outperformed both hybrid and Cloud-based implementations, indicating that the additional time needed for SmartCore-to-Cloud data transmissions and on-Cloud processing is more greater than performing all steps (Steps 1-8 in Table 1) on SmartCore.

Table 3. DTR for different implementations (1200 MhZ, 720p, 1FPS)

Implementation	DTR
Cloud-based	3.6
SmartCore-based	3.0
Hybrid	4.3

2. Cellular data usage: Using our real-world dataset, we have examined how much cellular data, on average, each implementation has used for processing the 10-minute videos in the dataset. For both Cloud-based and hybrid implementations, the data usage highly depends on both FPS (that specifies the data transmission frequency) and resolution of the video (that determines the size of each packet transmitted to the Cloud). Furthermore, for Hybrid implementation, the sparsity of the environment (e.g., how many vehicles are present in each video frame) has an effect on the cellular data usage: in crowded areas, the hybrid implementation of the application transmits more areas of interest to the Cloud. As shown in Table 4, the Cloud-based implementation consumes the most cellular data among the three implementations, whereas the SmartCore-based one utilizes the least (i.e., it only transmits the vehicle's GPS coordinates to the Cloud upon the detection of an abductor's vehicle). Hybrid implementation offers $34.8X$ reduction in cellular data usage in comparison to the Cloud one, at the cost of performing a lightweight algorithm on SmartCore. SmartCore-based implementation only occasionally communicates with the Cloud (to receive the updated database of active Amber Alerts and send the location of the vehicle upon the detection of a suspicious vehicle), however, it imposes significant computational overhead on SmartCore by locally processing all images.

Note that the Cloud-based implementation, that can be also implemented using the previously-proposed Cloud-based architectures described in Section 1, cannot be used in real-word scenarios due to its high cellular data usage.

Table 4. Cellular data usage (1FPS, 720p, 10 minutes)

Implementation	Cellular data usage (MB)
Cloud-based	115.0
SmartCore-based	0.0
Hybrid	3.3

For example, if the user only drives for one hour every day, it requires transmitting over 20 GBs of data every month over cellular network (assuming the video is captured at frame rate=1FPS and resolution=720p). This imposes a significant cost overhead on the user (currently, in the U.S., the cost is close to $100 per month for 20GBs).

3. Privacy leakage: Transmitting raw images captured from the vehicle to third-party servers, can potentially leak significant private information, including, but not limited to, the specifications of the vehicle (e.g., make, color, and model), the area that the vehicle's owner is travelling through, and the owner's locations of interest or even his identity (e.g., his face may be captured in some video frames when the Amber Response is running while the vehicle is stopped and the user is walking in front of the camera). Performing in-vehicle image processing can minimize the need of transmitting the raw data to the external servers, minimizing the private information leakage. Among three implementations of Amber Response, from privacy perspective, the Cloud-based implementation is the worst, whereas the SmartCore-based has the minimum information leakage (it does not transmit the raw image at all and only shares the user's location when it detects a suspicious vehicle in the surroundings). The hybrid version, that only transmits plate areas and removes other objects in the environment from the image, also significantly enhances the privacy of the user (in comparison to the Cloud-based version). However, based on our empirical results, it might occasionally detect some other objects in the environment as plates and transmit some inessential images, that can be potentially processed to reveal the user's location (e.g., images of logos and flyers), along with images of interest (i.e., images that only contain plates of nearby vehicles).

5.2 Application 2: Insurance Monitor

Pay-as-you-drive insurance policies are envisioned as the future of auto insurance [54]. Several insurance companies worldwide (for example, MetroMile [12]) have already introduced new low-rate insurance plans for which they take traveling mileage, along with the driver's behaviors, into account. They currently collect the required information (for example, the vehicle's speed and odometer readings) from a dongle that directly connects to the vehicle. Moreover, they commonly gather other types of data from the OBD port, including various diagnostic messages. Despite potential benefits that insurance dongles have offered, their usage is currently limited due to privacy concerns [31] and security threats [6, 10, 18].

A few solutions have been discussed in the literature to address the privacy/security issues associated with the use of insurance dongles [34, 54]. Such solutions commonly require *a design change in the hardware (insurance dongle) or back-end infrastructures (insurance servers).* They impose significant extra costs on companies due to at least one of the following reasons. First, insurance companies already have *millions of active dongles in the market* and changing the whole infrastructure (including dongles and servers) is very difficult (if not impossible). Second, to minimize design costs, they commonly use generic OBD dongles that are available from third-party companies, however, each new solution needs a specific dongle (i.e., a new type of dongles should be designed and developed for *each proposed solution*). Thus, insurance companies are unwilling to incorporate these solutions into their in-use scheme.

Based on ProCMotive, we design and develop an application that enables security/privacy-friendly pay-as-you-drive insurance, while imposing *no design change (and consequently, no additional cost) on insurance companies.* On the user side, the proposed application utilizes the access control library offered by SmartCore to ensure that the dongle only performs its intended activities (for example, only queries the speed data), preventing security attacks and minimizing the privacy leakage. Moreover, it uses the port management library, along with data manipulation techniques, to remove inessential sensitive data from the raw data requested by the insurance dongle, while maintaining the similar utility. Next, we first describe the privacy/security threats that we considered in this paper, and then discuss how we have developed an application based on ProCMotive to address these known threats.

5.2.1 Threat Model. Insurance Monitor aims to enhance the security of the vehicle and privacy of the user by addressing two well-known types of threats against Internet-connected vehicles enabled by the attachment of an insurance dongle: (i) the feasibility of tracking the vehicle based on the raw speed data collected from the OBD port (an attack against location privacy as discussed by Gao et al. in [31]), and (ii) the possibility of taking control of an insurance dongle. We assume that the adversary is (i) an insurance company or a third-party who has access to the speed data collected by the insurance dongle and is interested in collecting private location information or (ii) an attacker that exploits the vulnerabilities of an insurance dongle to inject malicious commands to the vehicle. In the following, we describe above attacks, along with their negative consequences, in more detail.

Attack 1: Tracking a vehicle based on its speed data: Gao et al. [31] have shown that the vehicle's location can be easily tracked by processing its speed data. They presented a novel algorithm, referred to as *Elastic Pathing*, that extracts location traces from raw speedometer data given an initial location (for example, the user's home address that is known to the insurance company). Applying their algorithm to real-world traces, they showed that how pay-as-you-drive policies, which rely on processing the speed data, are not privacy-preserving despite the claims of insurance companies. The leakage of private location information may expose the user to scams or unwanted advertisement. Furthermore, it may lead to several consequences, including the uncomfortable feeling of being monitored and actual physical harm [47, 59].

Attack 2: Taking control of the insurance dongle: Previous research studies [6, 10, 18, 53] have shed light on one common security vulnerability of third-party dongles: *dongles can be enforced (either remotely over the cellular network or within a short distance over Bluetooth connection) to send arbitrary messages to the OBD port with an arbitrary rate.* This vulnerability can potentially provide a direct access to several vital components and systems in the vehicle, enabling the attacker to perform various life-threatening attacks, ranging from remotely controlling the braking system [6] to launching DoS attacks against various built-in systems [38, 53]. For example, Foster et al. [6] have exploited security vulnerabilities of an insurance dongle (used by MetroMile [12]) to send arbitrary unauthorized messages to the OBD port. They constructed an end-to-end security attack, highlighting the potential seriousness of existing security flaws.

A note: The attachment of aftermarket dongles made vehicles vulnerable to remote threats since OBD port has been originally designed for diagnosis (that only requires a short-term attachment of an OBD reader) and has several security flaws. In particular, it does not offer any security scheme to distinguish authorized messages from unauthorized ones, assuming that every OBD-connected dongle is trusted and is allowed to transmit all requests and access all components. Furthermore, it utilizes a very simple congestion control protocol that always prioritize the messages with lower identification number over others. This congestion protocol makes DoS attacks against the vehicle very easy: the attacker can only send packets with the lowest possible identification number to the OBD port at a high frequency [38]. *Insurance Monitor* aims to address the threats emerged as a result of the widespread use of pay-as-you-drive insurance policies, which rely on aftermarket dongles.

5.2.2 Implementation. Next, we discuss how *Insurance Monitor* can be implemented on ProCMotive.
Addressing the attack against location privacy: Using the port management API, the proposed application first captures a packet from the insurance dongle and forwards it to the vehicle. It then gets the response from the vehicle and modifies its data field using a privacy-preserving function in such a way that the insurance company

can still get a similar utility (e.g., can correctly compute the number of times the user has speeding violation). Eventually, it sends the modified response to the dongle. In general, different privacy-preserving functions can be used to minimize the information leakage, for example, data shuffling, noise addition, and rounding techniques [30, 55] have been extensively discussed in the literature. Using these techniques, in the prototype implementation of *Insurance Monitor*, we have implemented three privacy-preserving algorithms.

(1) Alg. 1: Shuffling: Given a window size W, this algorithms aggregates W speed samples ($V = \{V_i, ..., V_w\}$) and returns a random permutation of them (V^*).
(2) Alg. 2: Rounding and then shuffling: Given a windows size W, for each sample of the vehicle's speed V_i, this algorithm first rounds V_i (to the nearest integer, nearest five, or nearest ten), then aggregates and shuffles W of them, and eventually returns the set V^*.
(3) Alg. 3: Noise addition: For each sample of the vehicle's speed V_i, this algorithm picks a float number Z_i drawn from a uniform distribution with the range of $R_{uniform}$, i.e., $0 < Z_i < R_{uniform}$, and returns $V_i^* = V_i + Z_i$.

Next, we demonstrate how each of these algorithms enhances the privacy of the user and affects the utility.

Preventing the security threat: *Insurance Monitor* ensures that (i) the dongle can transmit a set of expected requests (i.e., data request that are essential for pay-as-you-drive insurance) and (ii) the rate of requests generated by the dongle always remains below a reasonable threshold. This threshold can be predetermined by examination of an insurance dongle in a trusted environment (based on our empirical results, for MetroMile dongle, this threshold can be set to one request per second). Upon the attachment of the insurance dongle, using the Android application that we have developed to offer a user-friendly interface, the user can choose his insurance company. On SmartCore, the appropriate access control file, that specifies access control policies for the insurance dongle, will be automatically modified (using the access control API, which is accessible through the flask-based web server), and an upper bound will be set for the rate of requests (using the port management API) [5].

5.2.3 Evaluation. To verify that *Insurance Monitor* addresses the two attacks discussed earlier, we examined this application in two experimental scenarios, as described next.

Experimental scenario 1: Mitigating the attack against location privacy (Elastic Pathing): In order to examine how effectively the privacy-preserving algorithms utilized in *Insurance Monitor* address the Elastic Pathing attack [31], we examined how the accuracy of the attack, i.e., the distance between the estimated destination and the actual destination divided by the actual traveled distance reduces, and a speed-related utility required for the pay-as-you-drive policy degrade when *Insurance Monitor* exploits each privacy-preserving algorithm. The speed-related utility is defined as the number of times that the speed is above a certain threshold. In our experiments, we set the speed threshold to $25mph$, i.e., we assume that the insurance company intends to know how many times the vehicle's speed exceeded $25mph$. In our experiments, to offer a fair analysis, we utilized the database provided in [31] that includes several streams of a vehicle's speed collected from real-world driving traces (*Insurance Monitor* performs privacy-preserving function on the pre-recorded raw data). It is desired that *Insurance Monitor* reduces the accuracy of the attack, while maintaining the utility. Fig. 5 shows both accuracy of the attack and utility degradation (i.e., the difference between computed utility based on the modified data and actual utility divided by the actual utility) for three algorithms discussed above. Fig. 5 (a) demonstrates how windows size W affects both utility and attack accuracy when Alg. 1 is used. Fig. 5 (b) shows how both window size W and rounding precision affect the utility and attack accuracy when Alg. 2 is utilized. Eventually, Fig. 5 (c) demonstrates the accuracy of attack and utility degradation with respect to the range of the uniform distribution

[5]The port management API can be accessed through the flask-based web server. The application sets an upper bound by sending the following request to the web server running in CARWare: {"appID": "ROOT", "appToken": "<Administrative Password>", "requestType":"portManagement"}, {"type": "upperbound", "value":"1"}

($R_{uniform}$) for Alg. 3. Based on our experimental results, Algs. 1 and 2 slightly decrease the accuracy of the attack (or equivalently, enhance the user's privacy), whereas Alg. 3 can significantly reduce the accuracy of the attack with minimal utility degradation.

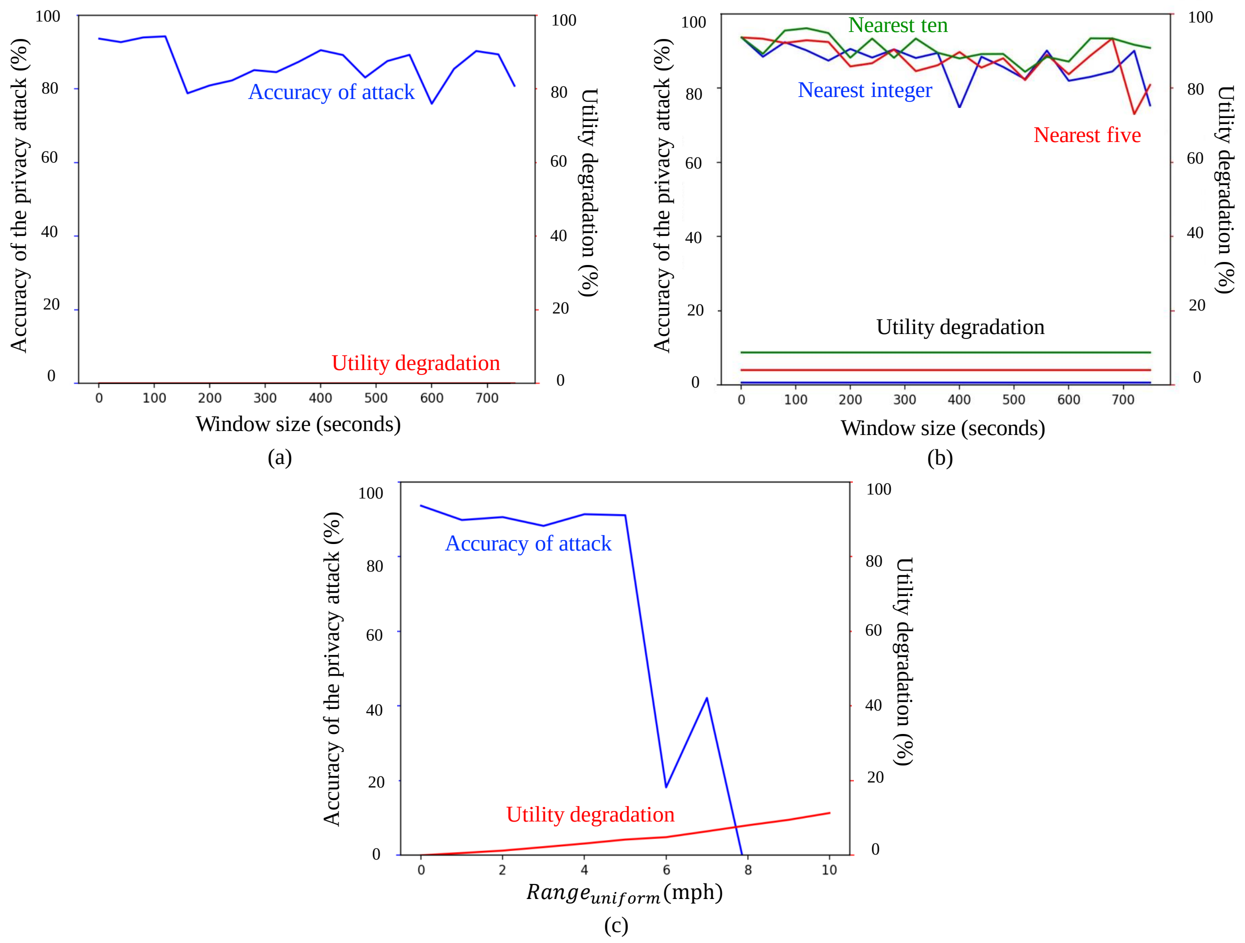

Fig. 5. Accuracy of the privacy attack (Elastic Pathing [31]) and utility degradation when privacy-enhancing algorithms (Algs. 1,2, and 3) are used.

Experimental scenario 2: Preventing the injection of malicious commands : As discussed earlier, based on empirical results, we assume that a legitimate insurance dongle should be able to collect speed data from the vehicle every second, i.e., *Insurance Monitor* modifies its access file so that the file specifies that the dongle can only access speed data, and sets an upper bound of 1 on its rate of requests. We connected a third-party dongle to SmartCore and first programmed it to create 100 legitimate requests (querying speed data with the frequency of 1). In this setup, we observed that SmartCore forwards all requests to the vehicle. We then reprogrammed the dongle twice in such ways that it violates the above-mentioned assumptions about the legitimate behavior of an insurance dongle dongle: (i) the dongle makes 100 attempts to send illegitimate requests (querying other data than speed data), and (ii) sends 100 legitimate requests at a high frequency (ranging from 10 to 200 requests per second). We observed that *Insurance Monitor* blocks all requests in both cases, preventing the attack against the vehicle. In the first case, CARWare simply drops the requests. However, in the second case, it still processes the requests, however, regulates their rate (CARWare puts legitimate requests in a queue and processes only one request per second). To sum up, *Insurance Monitor* prevents this attack by using two core libraries of CARWare, in particular, the access control and port management.

5.3 Summary and Highlights

In this section, we first summarize the advantages of ProCMotive-enabled implementations of the proposed vehicular applications and compare them with state-of-the-art approaches. We then briefly discuss how ProCMotive (that is implemented based on the reference architecture) addresses shortcomings of previous approaches for these two applications.

5.3.1 Summary. Table 5 highlights the advantages of ProCMotives for the two above-mentioned vehicular applications.

Table 5. Comparison of ProCMotive-enabled implementations with their baselines

Applications	Privacy	Security	Performance	Cellular data usage	Resiliency against connection unavailability
Amber Res. (SmartCore)	↑↑	N/A	Similar at 1FPS	↓↓	↑↑
Amber Res. (Hybrid)	↑	N/A	Similar at 1FPS	↓	Similar
Insurance Monitor	↑	↑	N/A	Similar	Similar

Comparing SmartCore-enabled (SmartCore-based and hybrid) implementations of Amber Response to its Cloud-based implementation (the baseline) demonstrates that they significantly reduce cellular data usage, enhance user privacy, and are more resilient against the potential unavailability of wireless connectivity, while they can provide promising performance results. As demonstrated earlier, although we have utilized a very powerful Cloud server (with 8 CPUs and 32GBs of RAM), performance results provided by SmartCore-enabled implementations are comparable to the Cloud-based implementation for low frame rates. In particular, with video inputs captured at 1FPS, the SmartCore-based version of Amber Response can accurately (with the accuracy of %100) detect the abductor's vehicle even faster than Cloud-based version (Table 3).

Furthermore, *Insurance Monitor* can provide several benefits for already-in-use pay-as-you-drive insurance policies (the baseline). In particular, it can prevent several security attacks and significantly enhance the privacy of the vehicle's owner, while maintaining the utility needed by insurance policies.

5.3.2 Highlights. Next, we discuss how ProCMotive addresses shortcoming of previous solutions.

Question 1: How does ProCMotive bring programmability to vehicles? As shown in Fig. 1, in the proposed architecture, SmartCore (the proposed programmable add-on dongle) is envisioned to host a variety of vehicular applications. In the prototype, we developed CARWare (the middleware specifically designed for vehicles) that enables intended functionalities/features of SmartCore, which are essential for hosting third-party applications (access control, application management, and update management). In addition, we have developed libraries (data collection and port management) that can facilitate, control, and regulate the interactions of third-party applications/dongles with the vehicle. In order to maximize the usability of the framework, we allow the developers to write their applications in the programming language of their choice: we implemented a flask-based web server inside CARWare that provides a RESTFul API for accessing the functions of core libraries (independent of the underlying programming language, an application only needs to send HTTP/HTTPS request to the web server). Both proposed vehicular applications have been developed based on the core libraries accessible via the RESTFul API. *Amber Response* utilizes the data collection library, and *Insurance Monitor* uses the assess control and port management libraries. Using the trusted Android application (discussed in Section 4.1.3), that has administrative privileges and is developed based on the application management, access control, and update management libraries, the user can easily manage both vehicular applications, modify their access level, and update them if needed.

Question 2: How does ProCMotive address drawbacks of wireless connectivity? As demonstrated in Section 5.1.2, in-vehicle data processing offered by SmartCore can be very beneficial to data-dominant applications. Both hybrid and SmartCore-based implementations of *Amber Response* significantly reduce cellular data usage (and the cost associated with transmitting the data over LTE network).

Moreover, we suggest the use of SmartCore for the implementation of safety and security-related applications. To ensure safety and security, we placed the core functions of *Insurance Monitor* (that aims to address security and privacy threats) on SmartCore and did not utilize external resources offered by the user's personal devices or the Cloud. This ensures that the application remains available and reliable even if the vehicle-to-Cloud or vehicle-to-smartphone communication channel becomes unavailable.

Depending on the desired response time, reliability requirements, and cost considerations, applications can be either fully or partially implemented on SmartCore. In particular, we expect that developers place the core functions of mission-critical and safety-related vehicular applications (for example, security attack detection or collision prediction algorithms), that should remain available even in the absence of a vehicle-to-Cloud or vehicle-to-smartphone communication channel, on SmartCore.

Question 3: How does ProCMotive enhance security/privacy? SmartCore offers sufficient in-vehicle resources to support strong cryptography mechanisms (for example, Advanced Encryption Standard [28]) needed for protecting wireless communications to/from the vehicle, limiting remote wireless attacks. Moreover, SmartCore protects the vehicle's OBD port and acts as a gateway for third-party dongles. Using the port management library, an application can monitor and regulate the behavior of other OBD-connected dongles to detect and block malicious activities initiated from them. Furthermore, the access control library enables the user to decide when, where, to what extent, and under what conditions, he wants to share his data with third-party applications or OBD-connected dongles. In addition, to improve the security of the vehicle, an application hosted on SmartCore can implement privacy-enhancing functions that manipulate the raw data before sharing it with third-party dongles or applications.

Question 4: How does ProCMotive take in-vehicle resource limitations into account? In the proposed architecture, SmartCore brings additional computational/storage resources to the vehicle and offers short-range and long-range communication to transfer data/tasks to nearby devices (e.g., the user's personal devices) or the Cloud. Indeed, SmartCore can run such applications with partially (or even without) utilizing either the user's smartphone or the Cloud: *Insurance Monitor* and SmartCore-based implementation of Amber Response completely rely on in-vehicle resources offered by SmartCore. These enable the implementation of a variety of low-latency/real-time applications on SmartCore.

6 DISCUSSION

In this section, we briefly discuss six items not yet explained in detail.

6.1 Additional Use Cases

Here, we suggest three more applications that can be implemented based on ProCMotive and be added to already-in-market vehicles. In particular, these applications can empower environmental monitoring, improve passenger safety, and enhance vehicle performance.

1. Empowering environmental monitoring: Crowdsensing: Environmental monitoring, that utilizes environmental data and scientific principles to resolve environmental issues (for example, climate change, urban heat islands, and air pollution) has garnered ever-increasing attention in recent years. The increasing prevalence of sensors and wireless sensor networks (WSNs) has significantly revolutionized environmental monitoring. *Despite numerous benefits that WSNs can provide, their scalability is very limited due to the cost associated with their design, deployment, and maintenance.* To address the scalability issue of sensor-based systems in environmental monitoring, we suggest an entirely new research direction, motivated by two fundamental observations: (i) already-in-market vehicles have numerous under-utilized sensors and (ii) their mobility and wide area of coverage make them promising contributors to crowdsensing (to carry additional sensors needed for crowdsensing and/or be used as crowdsensors). An application can be developed based on ProCMotive to collect different environment-related data from the vehicle's built-in sensors or add-on modules, partially/fully process them (e.g., compress the data, remove the inessential privacy-sensitive portions of the raw data, or train learning models) using in-vehicle or nearby resources, and send the results to the Cloud over the LTE network. We envision that such an application can transform vehicles into very powerful cost-efficient contributors to environmental crowdsensing, tremendously empowering environmental monitoring.

2. Improving passenger safety: Sign detection and collision prediction: In the recent literature, several research studies have proposed and evaluated different algorithms for sign detection and collision prediction. State-of-the-art algorithms offer a high accuracy in sign detection and early prediction of accidents, however, require powerful computational/storage resources that are now missing in the majority of already-in-market vehicles. ProCMotive potentially enables developers to rely on additional in-vehicle resources offered by SmartCore and develop new safety-enhancing technologies for in-market vehicles. For example, an application hosted on SmartCore can use a vision-based sign detection algorithm [46] to detect a stop sign and check if the vehicle's speed is above a safe threshold as the vehicle is moving toward the sign. Moreover, a vision-based collision detection algorithm can be implemented based on ProCMotive and informs the user about a possible accident a few seconds before the incident.

3. Enhancing vehicle performance: Transmission assistant: We propose and briefly discuss an application that assists the driver of a vehicle with manual transmission to find the optimal time for changing the gear. Although automatic transmission enhances user convenience and driving experience, manual transmission usually offers a better acceleration, less weight, and less power loss than an automatic transmission *if used correctly*. To maximize the engine lifetime, vehicle manufacturers commonly suggest a number representing the optimal revolutions per minute (RPM) for changing the gear in the vehicle's manual. However, in practice, it is not straightforward for drivers to figure out what is the optimal RPM for enhancing the fuel efficiency or acceleration since it depends on several parameters, including the vehicle's specification (e.g., make, model, type), the age of the vehicle, and the driver's goal (i.e., whether he wants to maximize the acceleration or minimize the fuel consumption). An application hosted on the SmartCore has access to several engine-related parameters, including, the current RPM, fuel consumption, and vehicle's speed. Such an application can utilize these parameters, along with the user's feedback, to find the optimal RPM for changing each gear (for example, the application can learn the optimal RPMs for minimizing fuel consumption over several days, and then provide active feedback, such as visual warnings, to let the user know when he should change the gear).

6.2 A Closer Look at Application Development

Developing a customized vehicular application involves two main steps. A developer first uses the core libraries discussed in Section 4 to interact with the vehicle and implement the algorithmic parts of the application. He then defines an access control file (the user confirms the access policies included in the file upon the installation of the application). In the following, we briefly describe each step.

6.2.1 Using the RESTFul API to build new application: As demonstrated in Fig. 2, CARWare consists of five core libraries. Core libraries enable several fundamental functionalities of the middleware (for example, the access control library has a private function, named *contextRecognition(...)*, that periodically detects the current contexts) and provide a set of public functions (for example, *getData(...)* implemented in the data collection library enables collecting different types of sensory/non-sensory data).

Table 6 summarizes key functionalities (public functions) of the core libraries that can be used in vehicular applications. Public functions are only accessible through the flask-based web server running inside CARWare. For example, a third-party application can query the vehicle's speed by sending the following request to the RESTFul API offered by CARWare: [{"appID": "<application Identifier>", "appToken": <Application Token>, "requestType": "dataCollection"}, {"source": "vehicle", "type": "vehicle_speed"}]. In this JSON array, the first element contains the application's credentials and indicates that the application wants to access the data collection library, and the second element specifies that the application queries the vehicle's speed [6]. For all requests sent to the flask-based web server, the first element carries the application's or administrative credentials and specifies one of five core libraries, however, the other elements of the array vary from one request to another (see Section 4 and its footnotes for further details).

Table 6. Core libraries and summary of their key functionalities

Library	Key functionalities
Update Management	It enables updating apps and CARWare.
Data Collection	It enables gathering data from the vehicle's sensors and add-on modules.
Application Management	It can be used to install, pause, halt, remove apps from SmartCore.
Port Management	It enables blocking a port, adjusting the rate of requests sent to the OBD port, monitoring third-party dongles, and sending arbitrary requests to the OBD port.
Access Control	It can be used to define, modify, and remove access control policies.

Utilizing external resources: As demonstrated earlier in Section 5 (*Application 1: Amber Response*), developers may prefer to impose minimal computational/storage overhead on SmartCore and push several functions of an application to other resources (for example, in hybrid and Cloud-based implementation of *Application 1: Amber Response* several functions are placed on the Cloud). In this case, developers must build application for SmartCore that enables the interaction of the vehicle with outside resources. For example, for the Cloud-based version of (*Amber Response*), a simple application hosted on SmartCore interacts with CARWare to gather frames from the environment and forwards the collected frames to an external server hosted on the Cloud (the application simply uses the data collection library that is accessible via the server running in CARWare).

6.2.2 Setting initial access control policies. For each application, the developer needs to specify what core software components (for example, the data collection or access control library) the application uses, what type of sensory/non-sensory data (for example, the vehicle's speed, GPS coordinates, and VIN) it collects from the

[6]In Python programming language, a request (*R*) in JSON format can be sent to the web server using the following syntax: response=requests.post(webserver_url, *R*, headers={'Content-type':'application/json'})

vehicle, what wireless communication module (e.g., LTE, WiFi, or BLE) it needs. The developer should initially build an access control file (in JSON format [7]) that reflects the requested access level for the application.

6.2.3 *Limitations.* Here, we discuss two limitations of the prototype:

- ProCMotive does not currently support automatic functional decomposition of applications and load balancing. In our prototype, we rely on developers to divide an application into different functions and place them on different components along the vehicle-to-Cloud continuum. In general, the optimal amount of offloaded computation to the Cloud depends on several factors including the network quality, temporal power and latency constraints of SmartCore, and the resource utilization and service price of deployed Cloud servers [51].
  State-of-the-art practical load balancing techniques commonly aim to distribute functions of an application over different computational resources to enhance the performance of the application. However, for several complex vehicular applications, developers may have other design considerations, such as reliability requirements and privacy concerns. For example, *Insurance Monitor* intends to remove inessential information from the raw data before sharing it with dongles. In this example, privacy concerns enforce us to place data manipulation techniques on SmartCore: to ensure privacy, sensitive data should be locally processed, and resources available on-Cloud cannot be utilized.
- In the current prototype, CARWare only supports diagnostics queries: an application hosted on ProCMotive can only query sensory/non-sensory data or send minimally-controlling commands (for example, commands that clear warning messages) available in standard OBD protocol.

6.3 Battery Isolation

Here, we discuss how we can ensure that SmartCore adds minimal energy consumption overhead to the vehicle's battery.

In the prototype, SmartCore uses an internal battery (with the capacity of 50,000mAh) that provides sufficient energy (when it is fully-charged) to main SmartCore operational for several days. When the vehicle is running, SmartCore's battery can be recharged via OBD port, which provides a direct access to the vehicle's battery that is charged by an electrical alternator. SmartCore can easily detect when the vehicle is off and enter a low-power mode as follows. It periodically queries the vehicle's RPM and examines if the RPM is equal to zero (or cannot be read). If so (the vehicle is not running), SmartCore immediately disconnects its battery from the vehicle's battery and informs all applications that it intends to pause them shortly. After isolating its internal battery, it waits for a predefined period of time to allow applications to enter a safe mode (if necessary) and then pauses all of them. Afterwards, it only checks the status of the vehicle on a regular basis (for example, every 30 seconds). Its internal battery can maintain this functionality for over 10 days. As soon as it detects the vehicle's engine is running, SmartCore resumes the paused services and activates an electrical relay to charge its internal battery through OBD (this relay disconnects the internal battery of SmartCore from the vehicle's battery when the vehicle is not running).

We propose to use an internal battery for SmartCore to ensure that running services, middleware, and underlying OS have sufficient time to enter a safe mode without imposing additional energy consumption overhead on the vehicle's battery. Furthermore, the above-mentioned procedure prevents battery draining attacks (i.e., an attack in which a service attempts to drain the vehicle's battery) since it automatically disconnects SmartCore from the vehicle's battery as soon as it detects the vehicle is not running and pauses applications shortly after.

[7]For example, {“source”:“GPS”, “type”:“location”, “policyType”:“context”, “contextType”:“situational-emergency”, “status”:“accident”} indicates that the vehicle's GPS coordinates is accessible only if the vehicle is involved in an accident (this is a situational policy).

6.4 Potential Interactions between Vehicles

Although the majority of vehicular applications already employed in modern vehicles and discussed in the literature offer non-collaborative vehicular services (i.e., a service in which vehicles do not need to communicate with each other), sharing data between vehicles is beneficial for emerging vehicular applications. For example, it enables a braking assistant service to react with respect to another vehicle's movements even when it is obscured by other vehicles or objects. As another example, consider an application that aims to find localized security attacks against vehicles, e.g., a malicious road-side device that attempts to send deceitful wireless commands to the vehicle in an attempt to active its warning systems [35] or a mechanic who performs illegitimate activities on the vehicle. Relying on collaborative reasoning can significantly benefit such an application: upon the occurrence of warning messages or non-regular events (in particular, rapid changes in the sensory readings and accessing OBD port), each vehicle can share different types of data (for example, sensory data collected by the vehicle or requests made by OBD dongles) with other vehicles, enabling them to reason about possible security attacks.

As shown in the reference architecture, SmartCore offers two types of wireless communication: short-range communication protocols (for example, BLE) and Internet connectivity over LTE. Short-range communication protocols enable a vehicle to communicate with nearby vehicles. In particular, BLE allows SmartCore to quickly transmit data (for example, a message can be encapsulated in an advertising packet that does not require any prior connection establishment [8]) to vehicles within a 100-meter distance (see [29] for a thorough discussion on potential uses of BLE for vehicle-to-vehicle communications). Furthermore, BLE enables the vehicles to indirectly communicate to each other (through a smartphone application). Moreover, in the proposed architecture, Internet connectivity enables indirect vehicle-to-vehicle communications through Cloud servers, for example, a vehicle can upload its data, along with its GPS coordinates, to a sever hosted on the Cloud and the sever can share the vehicle's data with nearby vehicles.

6.5 Ensuring Physical Security of SmartCore

SmartCore may contain some private information, such as, credit card information, and can be unplugged anytime by unauthorized user. It is essential that the device can authenticate the user (or equivalently his vehicle) before offering any services. Next, we briefly describe three authentication approaches, along with their shortcomings.

Approach 1: VIN-based authentication: Each vehicle carries a VIN, a unique identification number that serves as the car's fingerprint. Indeed, no two vehicles in operation have the same VIN. In the initial setup of SmartCore, the user connects the dongle to the vehicle and sets a private password for the device, while SmartCore requests the VIN via OBD port. SmartCore then uses the VIN number to make sure that it is connected to the legitimate user's vehicle: it frequently (e.g., every 30 seconds) checks if SmartCore is still connected to the authorized vehicle. Upon the detection of a new VIN number, it stops all services and encrypts sensitive data and requests for the user's private password. Thus, if SmartCore is removed from one vehicle and attached to another vehicle without the user's permission, it will not remain fully functional. Although this approach can significantly limit the attacker, it alone does not offer a bulletproof secure solution for two main reasons. First, since VIN cannot be read from the OBD port when the vehicle is off, it does not enable us to detect the detachment of device when the vehicle is off. Second, the attacker may still try to remove SmartCore from the vehicle and use an OBD simulator (i.e., a small device that replicates the OBD interface and implements vehicles' standard communication protocol) to provide a fake VIN to SmartCore. In this scenario, if the attacker knows the authorized vehicle's VIN (or finds the number by exhaustively searching all possibilities), he may be able to access private data stored on SmartCore.

Approach 2: Remote deactivation: Similar to the common approach used in smartphone industry, the security of SmartCore can benefit from *remote deactivation* that enables the user to remotely deactivate/format the device and report it as *"stolen"*. On the Cloud, we maintain a database of reported devices, and SmartCore frequently asks the Cloud server if the authorized user marked his device as *"stolen"*. This approach has one main limitation: the user cannot deactivate the device if it is completely disconnected from the Internet.

Approach 3: External authenticator: Each time SmartCore powers up, it can check the presence of an external authenticator device, for example, the user's smartphone. To impose minimal overhead on the user's device and minimize his involvement, we propose that SmartCore first performs a one-time authentication based on the user's smartphone and then utilizes *Approach 1* to continuously authenticate the vehicle based on its VIN. As soon as SmartCore becomes unable to authenticate the user based on the VIN (the vehicle becomes off or the dongle is removed), SmartCore enters a secure mode in which it halts services, encrypts sensitive data, and frequently checks for the presence of the secondary authenticator device.

A similar approach is using a Bluetooth or near-field communication (NFC) token hidden by the user in the vehicle that maintains identification credentials (shared with SmarCore at the initial setup). In this case, SmartCore can frequently check if it is located close to the token (e.g., in the vehicle). In this case, since it can periodically check the presence of the token without imposing any additional overhead on the user's personal devices, the VIN-based authentication approach may not be necessary.

6.6 Consequences of Failures in SmartCore

As mentioned earlier in Section 6.2.3, in the current prototype, we only allow a vehicular application to send non-/minimally-controlling commands to the vehicles. Although sending controlling commands (i.e., commands that can directly affect the operations of internal subsystems in the vehicle, for example, lock doors) through OBD port is not impossible, the availability of such commands varies significantly from one vehicle to another (it is manufacturer-specific). Furthermore, sending commands, that are not supported in the standard of OBD protocol, requires packet spoofing (i.e., using identification credentials of an internal component to create packets that can deceive other internal components). Packet spoofing is neither recommended by vehicle manufacturers nor safe. Therefore, we chose not to include controlling commands in the prototype. This limits the domain of vehicular applications, however, offers a side benefit: in the event of a failure in the dongle (for example, an application or connection failure), the core functions of the vehicle remain intact and stable. If the OBD port fails or malfunctions (for example, it cannot power up SmartCore due to a fuse failure), it negatively affects the functionality of the SmartCore: the dongle may not be able to access the vehicle's internal battery (however, it can remain functional for a limited time since it has an internal battery), or several services can become unavailable due to the unavailability of the data being captured from the vehicle. In both above-mentioned scenarios (failure of the dongle or the OBD port), the safety and security of the vehicle becomes similar to when the vehicle does not have the dongle.

A note: In some cases, developers may intend to ensure the full functionality of a set of mission-critical applications (in particular, security and safety-enhancing applications, such as collision prediction) even in the presence of a failure in SmartCore. Towards this end, a secondary dongle can be added to the reference architecture. The secondary dongle can offer less computational/storage resources and only host the small set of mission-critical services that should remain functional (at least for a short period of time until the driver can resolve the SmartCore's issue). This dongle can share the vehicle's OBD port with SmartCore and passively monitor SmartCore's behavior to detect its failure. Upon the detection of a failure in SmartCore, the secondary dongle can disconnect it from the vehicle and take its mission-critical responsibilities. However, if the OBD port fails, the functionality of the secondary dongle can also become very limited due to the unavailability of the needed data.

7 RELATED WORK

The emergence of the IoT paradigm has led to an exponential increase in the number of Internet-connected sensing and computing objects, and in particular, has provided the opportunity to transform an isolated vehicle into an Internet-connected smart object [48]. Several recent publications have discussed potential benefits that Cloud-based services can provide for Internet-connected vehicles and proposed novel architectures [24, 39, 56] to enable Cloud-based services for *future vehicles*. Furthermore, many researchers and developers have investigated novel Cloud-enabled vehicular applications [33, 36, 40, 43, 58]. For example, Ji et al. [36] have proposed a Cloud-based car parking system that aims to find the nearest available car parking lot by processing the data collected from nearby vehicles on the Cloud. Meseguer et al. [43] have implemented a Cloud-based smartphone-assisted system that continuously analyzes drivers' behaviors using a neural network.

In addition to Cloud-based services, different smartphone-based applications have been developed for diagnostic purposes [22] (e.g., finding a faulty unit), controlling the vehicle's basic components [37] (e.g., locking/unlocking doors), and assessing the driver's behaviors [26].

Despite the existence of several proposal for development of vehicular applications in the literature, there is still a significant challenge that hinders their deployment: the majority of already-in-market vehicles have limited resources and communication capabilities and rarely support programmability. Furthermore, as described earlier in Section 2.1, mission-critical operations cannot be implemented on the Cloud or nearby personal devices due to connectivity and reliability issues. ProCMotive provides an interoperable approach to shift computational/storage resources from the Cloud to the vehicle, considering several shortcomings of previous approaches and different domain-specific design goals. Its unique approach imposes no design change on vehicles, and therefore, can potentially facilitate rapid development and deployment of new vehicular applications.

8 CONCLUSION

In this paper, we presented a reference architecture that potentially enables rapid development of various vehicular applications. The architecture is formed around a core component, called *SmartCore*, a privacy/security-friendly programmable dongle that offers in-vehicle computational and storage resources and hosts applications.

Based on the reference architecture, we developed an application development framework for vehicles, that we call *ProCMotive*. To highlight potential benefits that ProCMotive offers, we proposed and developed two new vehicular applications based on ProCMotive, namely, Amber Response and *Insurance Monitor*. We evaluated these applications using real-world data and compared them with state-of-the-art technologies.

ProCMotive enables application developers and researchers, who are interested in proposing and examining vehicular applications, to rapidly design, prototype, and evaluate novel applications for vehicles.

REFERENCES

[1] 2017. Amazon Web Services. [Online] https://aws.amazon.com/ec2/instance-types/. (2017). Accessed: 07-30-2017.
[2] 2017. Amber alert reports. [Online] https://www.amberalert.gov/statistics.htm. (2017). Accessed: 07-30-2017.
[3] 2017. Automatic license plate recognition library. [Online] https://github.com/openalpr/openalpr. (2017). Accessed: 07-30-2017.
[4] 2017. Can a vehicle be harmed with bad inputs via an OBD-2 port? [Online] https://mechanics.stackexchange.com/questions/19109/can-a-vehicle-be-harmed-with-bad-inputs-via-an-obd-2-port. (2017). Accessed: 07-30-2017.
[5] 2017. Car ECU flash reprogramming and why reprogram. [Online] http://www.totalcardiagnostics.com/support/Knowledgebase/\Article/View/52/0/car-ecu-flash-reprogramming--why-reprogram. (2017). Accessed: 07-30-2017.
[6] 2017. Cars can be hacked by their tiny, plug-in insurance discount trackers. [Online] http://money.cnn.com/2015/08/11/technology/car-hacking-tracker/index.html. (2017). Accessed: 07-30-2017.
[7] 2017. Cheaper car insurance dongle could lead to a privacy wreck. [Online] https://nakedsecurity.sophos.com/2015/01/20/cheaper-car-insurance-dongle-could-lead-to-a-privacy-wreck/. (2017). Accessed: 07-30-2017.
[8] 2017. Custom GAP advertising packet. [Online] https://docs.mbed.com/docs/ble-intros/en/latest/Advanced/CustomGAP/. (2017). Accessed: 11-06-2017.

[9] 2017. GPS Receiver. [Online] http://www.globalsat.com.tw/s/2/product-199952/Cable-GPS-with-USB-interface-SiRF-Star-IV-BU-353S4.html. (2017). Accessed: 07-30-2017.

[10] 2017. Hackers cut Corvette's brakes via a common car gadget. [Online] https://www.wired.com/2015/08/hackers-cut-corvettes-brakes-via-common-car-gadget/. (2017). Accessed: 07-30-2017.

[11] 2017. The Leading Open Platform for Connected Cars. [Online] https://moj.io. (2017). Accessed: 07-30-2017.

[12] 2017. Metromile. [Online] https://www.metromile.com/dashboard/. (2017). Accessed: 07-30-2017.

[13] 2017. Microsoft launches a new cloud platform for connected cars. [Online] https://techcrunch.com/2017/01/05/microsoft-launches-a-new-cloud-platform-for-connected-cars/. (2017). Accessed: 07-30-2017.

[14] 2017. NETGEAR LTE modem. [Online] https://www.netgear.com/home/products/\mobile-broadband/lte-modems/LB1120.aspx. (2017). Accessed: 07-30-2017.

[15] 2017. OBD PiCAN. [Online] http://skpang.co.uk/catalog/images/raspberrypi/pi_2/PICAN2UGB.pdf. (2017). Accessed: 07-30-2017.

[16] 2017. Raspberry Pi camera module v2. [Online] https://www.raspberrypi.org/products/camera-module-v2/. (2017). Accessed: 07-30-2017.

[17] 2017. Raspbian. [Online] http://raspbian.org. (2017). Accessed: 07-30-2017.

[18] 2017. A remote attack on the Bosch Drivelog Connector dongle. [Online] https://argus-sec.com/remote-attack-bosch-drivelog-connector-dongle/. (2017). Accessed: 07-30-2017.

[19] 2017. Researchers Hack Car via Insurance Dongle. [Online] http://www.securityweek.com/researchers-hack-car-insurance-dongle. (2017). Accessed: 07-30-2017.

[20] 2017. SimpleLink SensorTag. [Online] http://www.ti.com/ww/en/wireless_connectivity/sensortag/. (2017). Accessed: 07-30-2017.

[21] 2017. Simply Prepaid. [Online] https://prepaid-phones.t-mobile.com/simply-prepaid. (2017). Accessed: 07-30-2017.

[22] 2017. Ten diagnostic apps and devices to make you a better driver. [Online] http://www.popularmechanics.com/cars/how-to/g767/10-diagnostic-apps-and-devices-to-make-you-a-better-driver/. (2017). Accessed: 07-30-2017.

[23] 2017. What is Docker? [Online] https://www.docker.com/what-docker. (2017). Accessed: 07-30-2017.

[24] Hassan Abid, Luong Thi Thu Phuong, Jin Wang, Sungyoung Lee, and Saad Qaisar. 2011. V-Cloud: Vehicular cyber-physical systems and cloud computing. In *Proc. 4th Int. Symp. Applied Sciences in Biomedical and Communication Technologies*. 165.

[25] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. 2010. Context-aware usage control for Android. In *Proc. Int. Conf. Security and Privacy in Communication Systems*. 326–343.

[26] German Castignani, Raphaël Frank, and Thomas Engel. 2013. Driver behavior profiling using smartphones. In *Proc. IEEE Int. Conf. Transportation Systems*. 552–557.

[27] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. 2012. Crêpe: A system for enforcing fine-grained context-related policies on Android. *IEEE Trans. Information Forensics and Security* 7, 5 (2012), 1426–1438.

[28] Joan Daemen and Vincent Rijmen. 2013. *The Design of Rijndael: AES-The Advanced Encryption Standard*. Springer Science and Business Media.

[29] Raphael Frank, Walter Bronzi, German Castignani, and Thomas Engel. 2014. Bluetooth Low Energy: An alternative technology for VANET applications. In *Proc. IEEE Conf. Wireless On-demand Network Systems and Services*. 104–107.

[30] Benjamin Fung, Ke Wang, Rui Chen, and Philip S Yu. 2010. Privacy-preserving data publishing: A survey of recent developments. *Comput. Surveys* 42, 4 (2010), 14.

[31] Xianyi Gao, Bernhard Firner, Shridatt Sugrim, Victor Kaiser-Pendergrast, Yulong Yang, and Janne Lindqvist. 2014. Elastic pathing: Your speed is enough to track you. In *Proc. ACM Int. Conf. Pervasive and Ubiquitous Computing*. 975–986.

[32] Miguel Grinberg. 2014. *Flask web development: Developing web applications with Python*. O'Reilly Media, Inc.

[33] Xiping Hu, Victor Leung, Kevin Garmen Li, Edmond Kong, Haochen Zhang, Nambiar Shruti Surendrakumar, and Peyman TalebiFard. 2013. Social drive: A crowdsourcing-based vehicular social networking system for green transportation. In *Proc. ACM International Symp. Design and Analysis of Intelligent Vehicular Networks and Applications*. 85–92.

[34] Muhammad U Iqbal and Samsung Lim. 2006. A privacy preserving GPS-based Pay-as-You-Drive insurance scheme. In *Proc. Symp. GPS/GNSS*.

[35] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. 2010. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *Proc. USENIX Security Symposium*. 11–13.

[36] Zhanlin Ji, Ivan Ganchev, Máirtín O'Droma, Li Zhao, and Xueji Zhang. 2014. A cloud-based car parking middleware for IoT-based smart cities: design and implementation. *Sensors* 14, 12 (2014), 22372–22393.

[37] Kenji Kato. 2014. Smartphone controller of vehicle settings. (June 10 2014). US Patent 8,751,065.

[38] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, and Hovav Shacham. 2010. Experimental security analysis of a modern automobile. In *Proc. IEEE Symp. Security and Privacy*. 447–462.

[39] Euisin Lee, Eun-Kyu Lee, Mario Gerla, and Soon Y Oh. 2014. Vehicular cloud networking: Architecture and design principles. *IEEE Communications Magazine* 52, 2 (2014), 148–155.

[40] Qing-quan Li, Tong Zhang, and Yang Yu. 2011. Using cloud computing to process intensive floating car data for urban traffic surveillance. *Int. J. Geographical Information Science* 25, 8 (2011), 1303–1322.

[41] Martin Lukasiewycz, Sebastian Steinhorst, Sidharta Andalam, Florian Sagstetter, Peter Waszecki, Wanli Chang, Matthias Kauer, Philipp Mundhenk, Shreejith Shanker, Suhaib Fahmy, and Samarjit Chakraborty. 2013. System architecture and software design for electric vehicles. In *Proc. IEEE Design Automation Conference*. 1–6.

[42] Dirk Merkel. 2014. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014).

[43] Javier E Meseguer, Carlos T Calafate, Juan Carlos Cano, and Pietro Manzoni. 2013. Drivingstyles: A smartphone application to assess driver behavior. In *IEEE Symp. Computers and Communications*. 000535–000540.

[44] Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N Asokan. 2014. ConXsense: Automated context classification for context-aware access control. In *Proc. ACM Symp. Information, Computer and Communications Security*. 293–304.

[45] Andreas Møgelmose, Dongran Liu, and Mohan M Trivedi. 2014. Traffic sign detection for us roads: Remaining challenges and a case for tracking. In *Proc. IEEE Int. Conf. Intelligent Transportation Systems*. 1394–1399.

[46] Andreas Mogelmose, Mohan Manubhai Trivedi, and Thomas B Moeslund. 2012. Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey. *IEEE Trans. Intelligent Transportation Systems* 13, 4 (2012), 1484–1497.

[47] Arsalan Mosenia, Xiaoliang Dai, Prateek Mittal, and Niraj Jha. [n. d.]. PinMe: Tracking a smartphone user around the world. *IEEE Trans. Multi-Scale Computing Systems* ([n. d.]). DOI: 10.1109/TMSCS.2017.2751462, 15 Sept., 2017.

[48] Arsalan Mosenia and Niraj Jha. 2017. A comprehensive study of security of Internet of Things. *IEEE Trans. Emerging Topics in Computing* 5, 4 (2017), 586–602.

[49] Amir Mukhtar, Likun Xia, and Tong Boon Tang. 2015. Vehicle detection techniques for collision avoidance systems: A review. *IEEE Trans. Intelligent Transportation Systems* 16, 5 (2015), 2318–2338.

[50] Simon Schliecker, Jonas Rox, Mircea Negrean, Kai Richter, Marek Jersak, and Rolf Ernst. 2009. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* 28, 7 (2009), 979–992.

[51] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. 2017. Incentivizing Self-capping to Increase Cloud Utilization. In *Proc. 2017 Symp. Cloud Computing (SoCC '17)*. 52–65.

[52] Shanker Shreejith, Suhaib A Fahmy, and Martin Lukasiewycz. 2013. Reconfigurable computing in next-generation automotive networks. *IEEE Embedded Systems Letters* 5, 1 (2013), 12–15.

[53] Wei Si, David Starobinski, and Moshe Laifenfeld. 2016. Protocol-compliant DoS attacks on CAN: Demonstration and mitigation. In *Proc. IEEE Vehicular Technology Conference*. 1–7.

[54] Carmela Troncoso, George Danezis, Eleni Kosta, Josep Balasch, and Bart Preneel. 2011. PriPayd: Privacy-friendly pay-as-you-drive insurance. *IEEE Trans. Dependable and Secure Computing* 8, 5 (2011), 742–755.

[55] Bhavna Vishwakarma, Huma Gupta, and Manish Manoria. 2016. A survey on privacy preserving mining implementing techniques. In *Proc. IEEE Symp. Colossal Data Analysis and Networking*. 1–5.

[56] Jiafu Wan, Daqiang Zhang, Shengjie Zhao, Laurence Yang, and Jaime Lloret. 2014. Context-aware vehicular cyber-physical systems with cloud support: Architecture, challenges, and solutions. *IEEE Communications Magazine* 52, 8 (2014), 106–113.

[57] Xueming Wang, Jinhui Tang, Jianwei Niu, and Xiaoke Zhao. 2016. Vision-based two-step brake detection method for vehicle collision avoidance. *Neurocomputing* 173 (2016), 450–461.

[58] Tao Zhang, Helder Antunes, and Siddhartha Aggarwal. 2014. Defending connected vehicles against malware: Challenges and a solution framework. *IEEE Internet of Things Journal* 1, 1 (2014), 10–21.

[59] Haojin Zhu, Suguo Du, Muyuan Li, and Zhaoyu Gao. 2013. Fairness-aware and privacy-preserving friend matching protocol in mobile social networks. *IEEE Trans. Emerging Topics in Computing* 1, 1 (2013), 192–200.

Received August 2017; revised November 2017; accepted January 2018