Security Analysis of Android Automotive

Conference Paper · April 2020

DOI: 10.4271/2020-01-1295

4 authors, including:

Mert Dieter Pesé
University of Michigan
7 PUBLICATIONS **33** CITATIONS
SEE PROFILE

Josiah Bruner
Georgia Institute of Technology
1 PUBLICATION **0** CITATIONS
SEE PROFILE

2020-01-1295 Published 14 Apr 2020

Security Analysis of Android Automotive

Mert Pese and Kang Shin University of Michigan

Josiah Bruner Georgia Institute of Technology

Amy Chu Harman International

Citation: Pese, M., Shin, K., Bruner, J., and Chu, A., "Security Analysis of Android Automotive," SAE Technical Paper 2020-01-1295, 2020, doi:10.4271/2020-01-1295.

Abstract

In-vehicle infotainment (IVI) platforms are getting increasingly connected. Besides OEM apps and services, the next generation of IVI platforms are expected to offer integration of third-party apps. Under this anticipated business model, vehicular sensor and event data can be collected and shared with selected third-party apps. To accommodate this trend, Google has been pushing towards standardization among proprietary IVI operating systems with their *Android Automotive* platform which runs natively on the vehicle's IVI platform. Unlike *Android Auto*'s limited functionality of display-projecting certain smartphone apps to the IVI screen, Android Automotive will have access to the in-vehicle network (IVN), and will be able to read and share various vehicular sensor data with third-party apps. This increased connectivity opens new business opportunities for both the car manufacturer as well as third-party businesses, but also introduces a new attack surface on the vehicle. Therefore, Android Automotive must have a secure system architecture to prevent any potential attacks that might compromise the security and privacy of the vehicle and the driver. In particular, malicious third-party entities could remotely compromise a vehicle's functionalities and impact the vehicle safety, causing financial and operational damage to the vehicle, as well as compromise the driver's privacy and safety.

This paper presents an Android Automotive system architecture and provides guidelines for conducting a high-level security analysis. It also describes what countermeasures have already been taken by Google to prevent potential attacks, and discusses what still needs to be done in order to offer a secure and privacy-preserving Android experience for next-generation IVI platforms.

Introduction

Android was launched in late 2008 as a mobile operating system by Google. While this open-source Linux-based platform was initially designed for touch screen-equipped mobile phones - dubbed as *smartphones* - the success of this versatile operating system on widely popular phones led Google to develop Android versions for TVs and smartwatches in the early 2010s. Android's penetration into different markets culminated in the launch of Android Auto in 2015, which was also due partially to the introduction of touch screens for in-vehicle infotainment (IVI) systems.

Android Auto is an app that runs on mobile handsets and once connected to the IVI (over USB, WiFi or Bluetooth AVRCP) projects certain select apps to the IVI screen. The focus of Android Auto is to offer multimedia and navigation apps with an enhanced UX to reduce distraction of the driver. Despite its support by all major OEMs (and Apple Carplay) for most of their models as of 2019, the major drawback of Android Auto is the lack of having access to any data generated inside the vehicle. It solely relies on the handset's sensors and does not read nor write any data to the in-vehicle network (IVN). As a result, a full car integration was not possible with Android Auto.

Android's business model is an extension of the existing Google business model: Revenue is obtained from Search, AdSense and the applications that enable these two. In the case of Android there is also revenue from app sales (Google Play Store), licensing fees (Google Play Services) as well as Google Play's multimedia contents (e.g., Music) [7]. For the automotive use-case, this is not entirely possible with Android Auto since no additional data can be leveraged.

This is a reason why Google introduced Android Automotive at Google I/O 2018. They announced a partnership with a massive car-making alliance of Renault-Nissan-Mitsubishi to run Android Automotive powered infotainment systems in millions of cars beginning 2021 [9]. Although car-makers have been hesitant to share valuable vehicle data with Google, the latter's efforts in creating a clean, powerful operating system tailored to run stand-alone on IVIs has convinced car-makers to adopt this new technology. A vehicle-specific Google Play Store will allow third-party apps to be deployed in numerous vehicles independent of OEM [27] and can possibly allow car-makers to easily access a share of revenue with Google. Third-party applications that require IVN access can range from smart home apps for optimizing customers' charging management in their home garage to usage-based insurance (UBI) apps. The latter compute the driving behavior from a set of sensors, such as speed or braking, to automatically adjust the insurance premium.

In recent years, wireless connectivity in vehicles has gained popularity. According to [6], 250M vehicles will be connected to the Internet of Things (IoTs) by 2020. Existing connected vehicles' (CVs') functionalities comprise infotainment, safety, diagnostics efficiency, navigation and payments [29]. In the next phase of CVs - which is starting now - cars will connect to third-party services using a built-in data connection, introducing novel vehicular data-collection platforms, such as BMW CarData [8]. Already 78M vehicles are connected to the web, with 98% of all new vehicles sold in the US and Europe expected to have cellular connections by 2021 [26].

However, all these positive developments come at the expense of security and privacy risks. Third-parties (and the platform provider Google) will have access to private/sensitive data which can be used by malicious entities to infer more information about the driver of the vehicle. Furthermore, CAN injection (write access to the IVN) has to be restricted to OEM apps. For instance, the HVAC app that Google offers as a native application [1] usually requires to write to the CAN bus to change fan or temperature settings. Other third-party apps do not have a reason to write to the IVN, and hence should be limited to read-only mode.

The goal of this paper is to introduce Google's Android Automotive framework based on all of its available documents and point out both security and privacy threats that this useful addition to the IVI world can cause. After describing three potential attacks on Android Automotive-equipped IVIs, we will discuss possible countermeasures or precautions that need to be taken by the OEM and Tier 1suppliers to mitigate the security and privacy risks.

The paper is structured as follows. First, we would like to review some existing academic work done on Android Automotive security, as well as point out how IVIs can be leveraged for automotive security attacks. Then, we will introduce the Android Automotive architecture as defined by Google, as well as a primer of the CAN bus to provide insights into the impact of CAN injection attacks. This will be followed by an overview of how to analyze the security of Android Automotive, including the EVITA's methodology for classifying potential attacks. As part of the security analysis, we will show three different potential attacks that can be mounted on an IVI running Android Automotive. Finally, we will discuss some countermeasures and recommendations for OEMs and Tier 1s to mitigate the attacks discussed within the threat model before concluding the paper.

Related Work

Android Automotive

Recently, some research has been done on Android Auto, mostly focusing on static analysis of the offered infotainment apps [22]. The main threat emerging from deploying Android Auto is distraction by using poor user interfaces, which can, in turn, impact driver safety. The authors of [23] found JavaScript vulnerabilities in a quarter of all analyzed apps, claiming that this leaves the infotainment system at serious risk. They did not analyze if the Android Auto client-side software on the IVI can gain access to the IVN, so that third-party Android Auto apps can actually access the car's data. The Android Auto specification mandates a gap between the IVN and Android Auto, partially due to its media- and navigation-based functions, unlike Android Automotive which needs access to the IVN.

As of now, there are only two major publications on Android Automotive. [28] proposes a sensing model for vehicular sensor data. It points out how its architecture can be embedded into the Android Automotive platform as a proof-of-concept. [13] is a high-level description of Android Automotive security. It focuses on malicious third-party apps and their potential impact on safety, security and privacy. Unlike our work, its focus lies on static and dynamic app analyses, not a security analysis of the framework. In fact, the authors developed a tool for vehicle-specific code analysis, called *AutoTame*. Their attacks focus on driver disturbance (by raising the volume of a media app) and availability (forking an app until it crashes). They also briefly discuss privacy, i.e., leakage of sensitive information, through the use of two apps. Only one app has permission to upload data to the Internet, whereas the other app can read sensitive sensor information. Through the means of inter-process communication (IPC), the app that reads sensitive information can communicate the sensitive information to the other app that uploads it to the malicious third-party. For this attack to succeed, the third-party must have control over both apps, which makes this attack harder to mount.

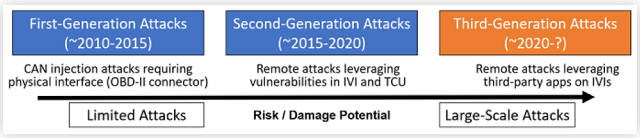
IVI Security

The in-vehicle infotainment (IVI) system is a major point of entry into the vehicle due to its connectivity provided by the telematic control unit (TCU). As a result, it is a popular attack vector since wireless surfaces, such as Bluetooth or WiFi, or wired interfaces, such as the USB port, can be exploited. Although manufacturers claim that the IVI usually has an air gap to the in-vehicle network (IVN), it has been shown in [25] that this is not always true. Once access to the CAN bus - the dominant IVN technology - has been obtained, it is possible to perform CAN injection attacks and compromise the functionality of the vehicle, as well as impact the safety of the driver.

Here we focus on CAN bus security. [18] identified that messages sent on the CAN bus can be easily eavesdropped on, and injected into the CAN bus since CAN messages are not encrypted or authenticated. [21] and [24] comprehensively demonstrated several CAN injection attacks to various components such as the brakes and engine on different vehicles. They show that all these attacks require access to the vehicle's internal CAN bus and can override several body- and powertrain-related events.

All of the aforementioned attacks were mounted by having physical access to the vehicle, usually through the OBD-II interface. Leveraging the wireless connectivity of infotainment and telematics platforms, remote attacks are becoming a reality. In addition, these attacks can be scaled easier since multiple vehicles can be targeted at the same time.

FIGURE 1 Classification of Automotive Attacks



This increases the probability of an attack to be mounted and thus the risk goes up. [10] demonstrated that remote vulnerabilities such as in GM's OnStar telematic solution would allow an attacker to gain access to the IVN. Besides leveraging the cellular interface to access the vehicle, short-range wireless attack vectors such as Bluetooth have also been evaluated. In the famous Jeep hack [25], two researchers staged a remote attack over the cellular network by compromising a software vulnerability in the IVI. Another use of cellular interfaces as point of entry into the vehicle was discussed in [14] when researchers crafted SMS messages to exploit an aftermarket dongle connecting to a vehicle via the standardized OBD-II port. [16] exposed vulnerabilities in the GM OnStar app that allowed unauthenticated API calls. The attack was only limited to sending CAN messages that were predefined by the OnStar app. Arbitrary CAN injection was impossible.

All attacks reported so far can be categorized into two generations as depicted in Fig. 1. The first generation of automotive attacks required a physical connection whereas most IVI attacks so far were exploiting wireless interfaces and became popular in research around 2015. More advanced and standardized IVI platforms such as Android Automotive will likely lead to the next generation of attacks on vehicles. In particular, remote attacks leveraging third-party (Android) apps on IVIs will penetrate more vehicles and lead to larger-scale attacks.

Threat Model and Background

Threat Model

Android Automotive marks the start of a new era for IVI platforms. Google began opening APIs for third-party developers in May 2019 [15] and encouraged development of multimedia apps. It was announced in May 2019 that the 2020 model year Polestar 2 EV (manufactured by Volvo) will be the first vehicle to feature an IVI equipped with Android Automotive. In October 2019, Volvo unveiled the XC40 Recharge which is also equipped with Android Automotive [31]. Remote attacks as described in the previous section are likely to increase with an open third-party developer ecosystem. As a result, working on the security of the Android Automotive framework is vital at an early stage. Infrastructure vulnerabilities such as security flaws in network or OS components are always possible and will require immediate response from the security team of the IVI manufacturer during the lifecycle of the unit. These vulnerabilities can be easily patched via over-the-air (OTA) updates in the future and can thus limit serious damage in a short time. Currently, only a few vehicles other than Tesla offer a fully functional OTA update process. This is partially due to the cumbersome integration of existing update solutions into the existing legacy OS architecture. If an OEM offers multiple IVIs in different models with different OS, they have to repeat this integration process which can turn out very tedious. Android Automotive is based on Android which supports OTA updates for many years now. The Android Open Source Project (AOSP) has even introductions for the specific automotive use case which "differs slightly in the Download step due to support for Garage Mode" [40]. Besides updating the IVI OS, OTA firmware updates for other ECUs are also supported according to Google's documentation. Aforementioned "Garage Mode" is a low-power mode that a parked car can use when the ignition off. This enables overnight OTA updates when the car is in the garage and connected to the customer's home WiFi network. Although Google allows OEMs to also enable USB updates on the IVI, we highly recommend against this sort of physical update process due to an increased security risk as depicted in the past [41]. We can assume that with gradual deployment of Android Automotive on IVIs, the OTA update process can be further standardized and made easier to integrate for automotive OEMs along all their future vehicle models.

In comparison, framework design vulnerabilities such as issues with Android Automotive's core modules or permission model are harder to fix after the system has been adapted for a certain time. This group of vulnerabilities can have a far-reaching impact and can cause significant disruption in the future. Hence, an early security framework is beneficial, especially with the notion that Android-equipped IVIs will hit the market in the following years.

Unlike static code analysis of Android Automotive apps (such as the one in [13], this paper is confined to an architecture-level security analysis. We look at the Android architecture itself, especially the vehicle-specific modules that have been introduced in Android Nougat (7.x). We focus on the interaction of third-party apps with the underlying Android Automotive-specific modules. The bottom module where our inspection ends would be the *Vehicle Hardware Abstraction Layer* (VHAL) since this is the module that communicates with the IVN.

A secure architecture will mitigate most, if not all, of framework vulnerabilities in our opinion. In what follows, we will introduce the Android Automotive stack in Google's codebase [1], elaborate on the existing permission model for vehicles, provide a primer on the CAN protocol, as well as discuss the VHAL which is the most critical component in interfacing with the car.

Overview

Android Automotive is a regular Android build with extra modules that have been specifically tailored for automotive apps to interact with the vehicle. We use the Android Oreo (8.1) branch of Android Automotive codebase for our analysis that can be found from: https://android.googlesource.com/

platform/packages/services/Car/+/refs/heads/oreo-release. The main modules are listed as follows:

- platform/packages/services/Car
- hardware/libhardware/include/hardware/vehicle.h
- hardware/libhardware/modules/vehicle/vehicle.c

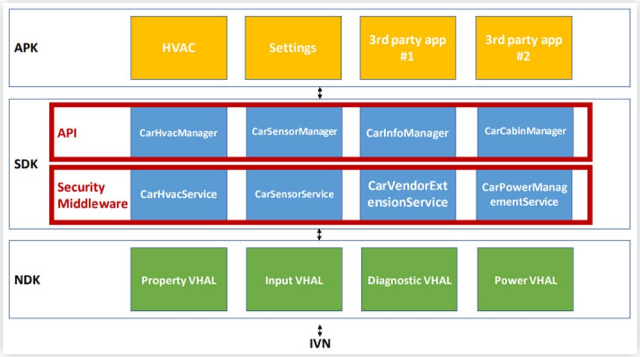
The system architecture of the Android Oreo branch (it is getting revamped significantly in newer flavors Android Pie and 10) is depicted in Fig. 2. It consists of three layers of modules. On the top layers, we have Android apps (APK) that can be categorized into native (manufacturer-provided) and third-party apps. Native apps are provided by Google (e.g., the HVAC app), but can be overridden by the OEM to offer customized builds. Third-party apps can be downloaded in the future from the vehicle-specific Play Store that is planned to be offered by Google. Examples of those apps can be usage-based insurance (UBI) apps such as Progressive Snapshot as mentioned earlier. Each app interacts with its *CarManager* component that is part of the SDK layer. Think of these components as APIs that are available to the apps to interact with the vehicle. Each *CarManager* module talks to a *CarService* component. Android Oreo defines certain instances of the latter, e.g., *CarHvacService* for the HVAC app or *CarSensorService* for third-party apps that read certain sensors from the IVN. This component of the SDK can be regarded as a security middleware where access control is defined, i.e., permissions are checked for that specific app. The permission model will be detailed later. If an app is allowed to communicate with the IVN, it will finally pass a final layer, the *Vehicle Hardware Abstraction Layer* (VHAL) that is part of the NDK. The VHAL is responsible for mapping vehicle properties (Google-specified signals that shall be supported by any vehicle, extendable by the vendor) to CAN signals defined in the vehicle's DBC file. Eventually, the VHAL reads from, or writes to the IVN, e.g., the CAN bus. Please note that other IVN technologies, such as Automotive Ethernet, are also possible. In the following, we will assume that the IVN consists of CAN buses.

Permission Model

Android permissions are divided into four protection levels [2]:

- **Normal:** Normal permissions result in minimal risk to the user's privacy. If an app declares in its manifest that it needs a normal permission, the system automatically grants the app that permission at installation time without any explicit confirmation. Users cannot revoke these permissions.
- **Dangerous:** Dangerous permissions are defined if the user's private information has to be accessed. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app at installation time and/or its first launch. The app might degrade or not work at all if these permissions are not granted.
- **Signature:** The system grants these app permissions at installation time, but only when the app is signed by the same certificate as the app that defines the permission. In the automotive domain, only OEM-native apps can use signature permissions.
- **signature|privileged:** These permissions are granted to either cryptographically signed (see signature above) or preinstalled apps.

FIGURE 2 Android Automotive System Architecture



All vehicle-specific permissions are defined in package *android.car.permission* [33] and summarized in Table 1. Third-party apps will either use normal or dangerous permissions. In any case, signature permissions are limited to native (OEM) apps, i.e., a regular app cannot access HVAC settings or control body functions such as the seats or windows. Currently, most permissions are signature or privileged. The only dangerous permissions at this time that require explicit user consent are speed and some more information about the vehicle's energy state. Nevertheless, several powertrain-related information, such as gear position or engine speed (RPM) are available to anyone without explicit permission. Although the current permission model is relatively strong, it can be designed more fine-grained as we will discuss later. Furthermore, while writing this paper, we encountered several iterations of the permission model, and hence assume that changes in the near future are unavoidable.

CAN Primer

Vehicular sensor data is collected from ECUs that are typically interconnected via an in-vehicle network (IVN), with the CAN bus being the most dominant technology in current vehicles. Fig. 3 depicts the structure of a CAN 2.0A data frame:

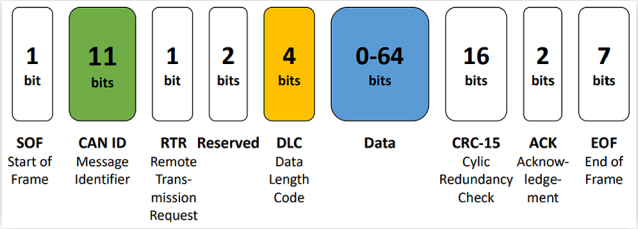
CAN is a multi-master, message-based broadcast bus which is message-oriented. Instead of having a source or destination address, each frame carries a unique message identifier (ID) that represents its meaning and priority. Lower CAN IDs have higher priority and will "win" the distributed arbitration process that occurs when multiple messages are sent on the CAN bus at the same time. The basic CAN ID in the CAN 2.0A specification is 11 bits long and thus allows for up to 2048 different CAN IDs. Data in a CAN frame (called payload) can be up to 8 bytes long.

Next, we will describe the structure of the data payload field, which consists of one or more signals. A *signal* is a piece of information transmitted by an ECU, such as vehicle speed. For instance, a message targeted for the Transmission Control Module (TCM) might contain both the vehicle speed (m/s)

TABLE 1 Permissions defined in Android Automotive (android.car.permission)

Permission Name	Protection Level
READ_CAR_DISPLAY_UNITS	Normal
CONTROL_CAR_DISPLAY_UNITS	Normal
CAR_ENERGY_PORTS	Normal
CAR_INFO	Normal
CAR_EXTERIOR_ENVIRONMENT	Normal
CAR_POWERTRAIN	Normal
CAR_SPEED	Dangerous
CAR_ENERGY	Dangerous
BIND_VMS_CLIENT	Signature
BIND_PROJECTION_SERVICE	Signature
BIND_INSTRUMENT_CLUSTER_RENDERER_SERVICE	Signature
BIND_CAR_INPUT_SERVICE	Signature
CAR_MOCK_VEHICLE_HAL	signature|privileged
READ_CAR_STEERING	signature|privileged
CAR_IDENTIFICATION	signature|privileged
CAR_MILEAGE	signature|privileged
CAR_TIRES	signature|privileged
CAR_ENGINE_DETAILED	signature|privileged
CAR_DYNAMICS_STATE	signature|privileged
CAR_VENDOR_EXTENSION	signature|privileged
CAR_PROJECTION	signature|privileged
ACCESS_CAR_PROJECTION_STATUS	signature|privileged
CONTROL_CAR_SEATS	signature|privileged
CONTROL_CAR_MIRRORS	signature|privileged
CONTROL_CAR_WINDOWS	signature|privileged
CONTROL_CAR_DOORS	signature|privileged
CONTROL_CAR_CLIMATE	signature|privileged
CAR_EXTERIOR_LIGHTS	signature|privileged
CONTROL_CAR_EXTERIOR_LIGHTS	signature|privileged
READ_CAR_INTERIOR_LIGHTS	signature|privileged
CONTROL_CAR_INTERIOR_LIGHTS	signature|privileged
CAR_POWER	signature|privileged
CAR_NAVIGATION_MANAGER	signature|privileged
CAR_DIAGNOSTICS	signature|privileged
CLEAR_CAR_DIAGNOSTICS	signature|privileged
VMS_PUBLISHER	signature|privileged
VMS_SUBSCRIBER	signature|privileged
CAR_DRIVING_STATE	signature|privileged
CONTROL_APP_BLOCKING	signature|privileged
CAR_CONTROL_AUDIO_VOLUME	signature|privileged
CAR_CONTROL_AUDIO_SETTINGS	signature|privileged
RECEIVE_CAR_AUDIO_DUCKING_EVENTS	signature|privileged
CAR_INSTRUMENT_CLUSTER_CONTROL	signature|privileged
CAR_HANDLE_USB_AOAP_DEVICE	signature|privileged
CAR_UX_RESTRICTIONS_CONFIGURATION	signature|privileged
STORAGE_MONITORING	signature|privileged
CAR_ENROLL_TRUST	signature|privileged

FIGURE 3 CAN Data Frame Structure



and engine speed (RPM) signals in one CAN message. The length and number of signals vary from CAN ID to CAN ID and are defined in the DBC file for that specific vehicle. This translation file is proprietary to the OEM and usually specific to a model generation. It specifies the start position and length of a signal, so it can be easily retrieved from the payload and mapped to a context.

All recorded CAN data can only be interpreted if one possesses the translation tables for the target vehicle. There are multiple standards for those tables, such as KCF [20] and ARXML [5]. However, the most common format used for this purpose is the aforementioned DBC [12], a standard created by German automotive supplier company Vector Informatik.

Vehicle Hardware Abstraction Layer (VHAL)

The *Vehicle Hardware Abstraction Layer* (VHAL) is a vendor-extendable Android Automotive module which abstracts vehicle data from the IVN to higher-layer components (see Fig. 2). Google provides a list of VHAL properties [32] that shall be supported by all vehicles implementing Android Automotive. Examples of these properties are listed as follows:

- HVAC Controls
- Vehicle Information (e.g., make, model, year, etc.)
- Powertrain-related data (e.g.., speed, RPM, etc.)
- Body-related data (e.g., windows, seats)
- Driver Safety (e.g., ABS)
- Diagnostic data (e.g., OBD-II DTCs)
- IVI-related data (e.g., display brightness).

Each property can be configured with the following attributes:

- Access Type (read/write)
- Change Mode (on change, continuous, polling, static)
- Area Type (global, zoned)
- Min/Max Values
- Min/Max Sampling Rate (for continuous properties).

Properties can be accessed using *get*, *set* and *subscribe* functions according to their protection level defined in

Android Automotive's permission model as described before. The OEM/vendor is responsible for mapping the supported properties to the associated vehicle signals. Properties are usually mapped to CAN signals defined in a DBC (or a selection of it) which can be stored on the IVI. As a result, properties will be transparent to all third-party app developers, i.e., anyone can create an app using the available properties without having to know the vehicle architecture. Google has also provided a method for vendors to extend the AOSP VHAL and add their own vendor-specific properties. These properties are tagged as vendor properties and reside in a specific ID range provided by Google. These properties will not be made public and can only be accessed by vendor services/apps via the *CarVendorExtensionService*. Most properties are read-only and have *get* methods implemented. *Set* methods are usually reserved for properties that are covered by *signature*|*privileged* permissions and are thus only usable by OEM/vendor apps.

EVITA Methodology

The objective of EVITA, a project co-funded by the European Union within the Seventh Framework Programme for Research and Technological Development, is to design, to verify, and to prototype building blocks for secure automotive on-board networks protecting security-relevant components against tampering and sensitive data against compromise. [17] outlines the process of analyzing the security requirements for automotive on-board networks. It introduced a way to calculate the risk of an attack path based on attack probability and severity. According to EVITA, the severity of an attack is considered in terms of safety, privacy, financial and operational security threats that may be associated with harm to the stakeholders. Safety severity can range from no injuries to life-threatening or fatal injuries. Privacy severity ranges from no unauthorized access to data to vehicle/driver tracking for multiple vehicles. Financial severity ranges from no financial loss to heavy losses for multiple vehicles. Finally, operational severity ranges from no impact on operational performance to significant impact for multiple vehicles. In what follows, we will introduce three theoretical attacks derived from the aforementioned severity categories. Financial and operational attacks will be merged into one category.

As mentioned in the threat model, we will analyze the system architecture of the platform. Although Google did a thorough job at implementing security countermeasures, such as access control and permission model to prevent potentially malicious third-party apps mounting serious attacks on the vehicle, we would still like to analyze if such a malicious app can perform damage as defined in the EVITA methodology. For now, it appears that the separation of apps into native and third-party apps with a proper permission model will prevent any malicious entities from gaining arbitrary write access to vehicle components. Nevertheless, there still remains room for improvement as the proposed security mechanisms are not yet fool-proof. Three theoretical **a**ttacks are described in the following section. Attack #1 focuses on compromising driver privacy, i.e., obtaining vehicular sensor data that should be restricted. Attack #2 shows how financial and operational damages can be done to the cluster by breaking it. Finally, Attack #3 depicts how driver safety can be compromised by distracting the driver, as well as gaining access to DBC files which can then be leveraged for direct CAN injection attacks.

Possible Attacks

Attack #1: Privacy

The first attack revolves around compromising driver privacy by leaking privacy-sensitive information. Imagine a third-party app that has access to the engine speed (RPM) as well as gear position. According to the permission model of Android Automotive, these are normal permissions, i.e., any app can obtain them without explicit user consent. In contrast, vehicle speed is labeled as a dangerous permission. The driver has to explicitly approve the collection of this sensor data on first start of a third-party app. Speed, RPM and gear position share a physical relationship [34]. If both RPM and gear position are known, it is easy to calculate the vehicle speed. Although there is no closed-form equation, with a sufficient amount of collected data, inferring vehicle speed is relatively straightforward. As a result, a sensor with dangerous permission can be inferred by an unprivileged third-party app. This should not happen, especially since vehicle speed contains very sensitive information about the driver and can be abused if it is in the wrong hands. As academic research results [11, 30] show, it is even possible to derive the GPS location (comes also with a dangerous permission in Android) from speed. This enables driver location tracking, a grave privacy leakage that has to be avoided at all cost. There may be more physical relationships that can be exploited to show even further privacy leakage. This is something that can be part of future work.

Attack #2: Financial/ Operational

This attack describes how both financial and operational damages can be inflicted to the vehicle/driver. Our goal for this attack is to break the cluster. Breaking other units such as the HVAC is impossible with the current security mechanisms since only native apps have access to those permissions. As depicted in Table 1, the protection level for CONTROL_CAR_DISPLAY_UNITS is normal, i.e., accessible to any unprivileged third-party app. According to [33], this permission allows an application to control the display units for distance, fuel, tire pressure, EV battery and fuel consumption which are displayed on the cluster that is being controlled by the IVI. For instance, a malicious third-party app can be instructed to switch from minimal to maximal fuel level at set intervals. From Android Automotive's design documents, the maximum frequency that we can perform this with stands at 1 Hz. As a result, we can force an analog instrument

gauge indicating the fuel level on the cluster to alternate between both ends of its range once per second which will eventually break the needle. This results in (minor) financial damage, as the car owner has to get the cluster repaired. Furthermore, this can cause operational damage as well, as the cluster will not be able to display the correct fuel level to the driver. Even if the potential damage is minor, there is no reason why a third-party app should be able to control the cluster. Since CONTROL_CAR_DISPLAY_UNITS is zero-permission, any useful app can hide this malicious intent without disclosing it to the driver.

Attack #3: Safety

The last attack will compromise the driver's safety and thus needs to be avoided at all cost. The same aforementioned attack to cause financial and operational damages can also lead to driver distraction. A continuous switching of cluster units will confuse the driver initially which could potentially lead to an accident.

Finally, we will present an attack on driver safety by exploiting possible wrong design choices of the IVI vendor which can lead to arbitrary CAN injection --- the worst-case scenario. Note that this attack is not specifically a vulnerability of Android Automotive, but more a possible implementation flaw. As discussed before, the VHAL mapping table is part of the Android Automotive firmware deployed on the IVI. A change to this mapping table can cause unseen behavior. Let us assume that the attacker knows the CAN signal to accelerate the vehicle (e.g., the gas pedal position which has been shown to perform this safety-critical attack). They could have obtained it by manually reverse engineering this specific signal on another target vehicle or using automated reverse engineering tools for vehicles [35]. If complete or partial DBC files for the target vehicle are included as part of the system firmware and the attacker obtains access to the firmware, they can reverse-engineer it and have access to the mapping table and thus DBC files. With this knowledge, a benign command, such as displaying a value on the cluster, can be exchanged with a more malicious command, such as accelerating the vehicle. Once recompiled and reflashed on the victim's vehicle, the latter will seriously misbehave while displaying cluster units, as acceleration of the vehicle will be forced. This is very dangerous to the driver. Another possible way to perform this attack is to obtain remote code execution on the target IVI. If the attacker can run the shell as a root user (e.g., by accessing the ADB shell over the IVI's USB port or WiFi), they can modify the VHAL mapping table by overwriting the DBC file which results in the same effect as described above. In order to prevent this attack, simple countermeasures, such as signing the firmware or blocking shell access by open debug interfaces, is sufficient. Unfortunately, adhering to those basic security principles has been inconsistent in the automotive industry, such as in the famous Jeep hack [25]. Furthermore, using DBC files for VHAL mapping in general can also be rethought, since a leakage of those proprietary translation tables are not in the best interest of the OEMs and can be used for physical CAN injection attacks.

Recommendations

Based on our observations, we would like to summarize some recommendations which we would like to make to Google as the platform provider, as well as OEMs and Tier 1s that are in charge of integrating Android Automotive into their IVIs:

- **Fine-grained permission model:** Although we found that the permission model of Android Automotive is constantly evolving, the current version (see Table 1) is still too coarse-grained which can lead to privacy leakage. In particular, multiple properties are summarized in one permission (e.g., CAR_POWERTRAIN). A possible way to both extend and separate permissions are to assign a unique permission to each property. Furthermore, [42] provides a privacy score that quantifies the privacy risk of 20 typical vehicular sensors. Based on this metric, it is also possible to define *dangerous* and *normal* permissions.
- **Further standardization from Google:** Currently, many security-critical functions (such as the DBC mapping) are not specified by Google. As a result, OEMs and Tier 1s are free to design specific components on their own. Unfortunately, the risk of lacking a secure design and implementation for these components is imminent as we demonstrated. We believe that Google as the platform provider should be responsible for suggesting or standardizing security recommendations. In the case of DBC mapping, OEMs and Tier 1s should at no point include (plain text) DBC files in the firmware image. One possible option is to "hard-code" a lookup table and offload the mapping task to Trusted Execution Environment (TEE) so that the code with the lookup table is protected with respect to confidentiality and integrity.
- **Separation of domains in IVN architecture:** Usually, the IVI is part of a dedicated "Infotainment CAN" which is connected to other CAN buses such as "Powertrain CAN" or "Body CAN" via a gateway module. The easiest and most efficient way to prevent CAN injection attacks is to move safety-critical domains such as the powertrain modules on a different CAN bus is to implement some sort of access control (e.g., whitelisting) or firewall in the gateway [39]. As a result of this, a compromised IVI will not be able to write to safety-critical areas. From our understanding, vehicular gateways in IVN architectures are not mandatory and some OEMs might want to continue without a gateway at all, primarily due to cost [36].
- **Protection against runtime attacks:** Since Android Automotive is based on the mature Android platform, we assume that the IVI OS comes with existing countermeasures. Nevertheless, Return-Oriented Programming (ROP) attacks can still occur where an attacker gains control of the call stack of an Android app. This can be done by a malicious third-party app for instance and lead to a buffer overflow on the IVI. [37] shows that despite a recent change from Dalvik to the more secure Android Runtime (ART), ROP threats still exist. Countermeasures have been proposed, such as

in [38]. Nevertheless, the code written in C/C++ (device drivers, etc.) is most vulnerable to runtime attacks. Vendors (OEMs and Tier 1s) write a majority of these components themselves to accomplish vehicle-specific functionality. This code should be scrutinized heavily (relative to the Java layer). Furthermore, we discussed the possibility to execute an ADB shell **i**n Attack #3. A good practice to avoid this is to disable USB debugging by default so the attacker cannot get authorized to launch an ADB shell or use other ADB functionalities. Since it is also possible to connect to ADB remotely over WiFi, developers have to think of restricting this attack surface as well. Even if the ADB shell can be accessed through USB or WiFi, it should be impossible to be ran as root user by default. This is controlled by a flag in the boot partition.

Summary/Conclusions

We conducted a first high-level security analysis of the emerging Android Automotive platform. We described its system architecture and discussed how automotive attacks can be scaled under a new threat model compared to previous generations of security exploits in vehicles. In particular, we discussed the newly introduced vehicle-specific Android permissions under the current permission model and found that it is still too coarse-grained and needs improvement. We showed its vulnerabilities through three theoretical attacks and presented possible mitigations. Furthermore, we showed how a bad implementation of the framework can compromise driver safety. Recommendations for a more secure design are presented. Nevertheless, the proposed security mechanisms of Android Automotive are promising. Since the framework is constantly evolving and first production units that leverage the full functionality of it are yet to be released, we are confident that the issues described in this paper can be addressed and incorporated into future versions of Android Automotive.

References

1. [n.d.], https://android.googlesource.com/platform/packages/apps/Car/Hvac/refs/heads/master.
2. [n.d.], https://developer.android.com/guide/topics/permissions/overview.
3. [n.d.], https://ibotpeaches.github.io/Apktool/.
4. [n.d.], http://craig.backfire.ca/pages/autos/transmissions.
5. AUTOSAR XML Schema, n.d., https://automotive.wiki/index.php/AUTOSAR_XML_Schema.
6. "Gartner Says By 2020, a Quarter Billion Connected Vehicles Will Enable New In-Vehicle Services and Automated Driving Capabilities," n.d., https://www.gartner.com/newsroom/id/2970017.
7. "Android Economics: An Introduction," 2012, http://www.asymco.com/2012/05/13/android-economics-an-introduction/.
8. "BMW Group Launches BMW CarData: New and Innovative Services for Customers, Safely and Transparently," 2017, https://www.press.bmwgroup.com/global/article/detail/T0271366EN/bmwgroup-launches-bmw-cardata:-new-and-innovative-services-for-customerssafely-and-transparently?language=en.
9. "Google Just Struck a Deal with a Major Auto Alliance to Run Android on Millions of Future Cars," 2018, https://www.androidpolice.com/2018/09/18/googleandroid-infotainment-renault-nissan-mitsubishi/.
10. Checkoway, S., McCoy, D., Kantor, B., Anderson, D. et al., "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security Symposium*, San Francisco, 2011, Vol. 4, 447-462.
11. Dewri, R., Annadata, P., Eltarjaman, W., and Thurimella, R., "Inferring Trip Destinations from Driving Habits Data," in *Proceedings of the 12th ACM Workshop on Privacy in the Electronic Society*, ACM, 2013, 267-272.
12. CSS Electronics, "CAN DBC File - Convert Data in Real Time (Wireshark, J1939)," n.d., https://www.csselectronics.com/screen/page/dbc-database-can-busconversion-wireshark-j1939-example/language/en.
13. Eriksson, B., Groth, J., and Sabelfeld, A., "On the Road with Third-Party Apps: Security Analysis of an In-Vehicle App Platform," in *Proc. 5th Int. Conf. Vehicle Technology and Intelligent Transport Systems (VEHITS)*, 2019, 64-75.
14. Foster, I., Prudhomme, A., Koscher, K., and Savage, S., "Fast and Vulnerable: A Story of Telematic Failures," in *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.
15. Ganz, A., "Google Opens Its Android Infotainment Operating System to Third-Party Apps," 2019, https://www.motorauthority.com/news/1122899_googleopens-its-android-infotainment-operating-system-to-third-party-apps.
16. Greenberg, A., "This Gadget Hacks GM Cars to Locate, Unlock, and Start Them (UPDATED)," 2017, https://www.wired.com/2015/07/gadget-hacks-gm-carslocate-unlock-start/.
17. Henniger, O., Apvrille, L., Fuchs, A., Roudier, Y. et al., "Security Requirements for Automotive On-Board Networks," in *2009 9th International Conference on Intelligent Transport Systems Telecommunications (ITST)*, IEEE, 2009, 641-646.
18. Hoppe, T. and Dittman, J., "Sniffing/Replay Attacks on CAN Buses: A Simulated Attack on the Electric Window Lift Classified Using an Adapted CERT Taxonomy," in *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS)*, 2007, 1-6.
19. Java-Decompiler, "java-decompiler/jd-gui," 2019, https://github.com/javadecompiler/jd-gui.
20. Julietkilo, "julietkilo/kcd," 2017, https://github.com/julietkilo/kcd.
21. Koscher, K., Czeskis, A., Roesner, F., Patel, S. et al., "Experimental Security Analysis of a Modern Automobile," in *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, 447-462.
22. Mandal, A.Kr., Cortesi, A., Ferrara, P., Panarotto, F. et al., "Vulnerability Analysis of Android Auto Infotainment

Apps," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ACM, 2018, 183-190.

23. Mandal, A.K., Panarotto, F., Cortesi, A., Ferrara, P. et al., "Static Analysis of Android Auto Infotainment and On-Board Diagnostics II Apps," *Software: Practice and Experience* 2019, 2019.
24. Miller, C. and Valasek, C., "Adventures in Automotive Networks and Control Units," *Def Con* 21(2013):260-264, 2013.
25. Miller, C. and Valasek, C., "Remote Exploitation of an Unaltered Passenger Vehicle," *Black Hat USA* 2015(2015):91, 2015.
26. Pymnts, "Who Controls Data in Web-Connected Vehicles?," 2018, https://www.pymnts.com/innovation/2018/data-sharing-smart-cars-privacy/.
27. Ryan.whitwam, "For Google, It's Full Speed Ahead with Android Automotive, But Not So Much with Android Auto," 2018, https://www.androidpolice.com/2018/05/14/google-full-speed-ahead-android-automotive-not-much-android-auto/.
28. Sadio, O., Ngom, I., and Lishou, C., "A Novel Sensing as a Service Model Based on SSN Ontology and Android Automotive," *IEEE Sensors Journal*, 2019.
29. Walford, L., "Definition of Connected Car - What Is the Connected Car? Defined," 2018, http://www.autoconnectedcar.com/definition-of-connected-carwhat-is-the-connected-car-defined/.
30. Zhou, L., Chen, Q., Luo, Z., Zhu, H. et al., Speed-Based Location Tracking in Usage-Based Automotive Insurance," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, 2252-2257.
31. Korosec, K., "Volvo Unveils Its First Electric Car, the XC40 Recharge," TechCrunch, Oct. 16, 2019, https://techcrunch.com/2019/10/16/volvo-unveils-its-first-electric-car-the-xc40-recharge/.
32. Android Developers, "VehiclePropertyIds: Android Developers," https://developer.android.com/reference/android/car/VehiclePropertyIds, accessed Oct. 29, 2019.
33. Google Git, "Service/AndroidManifest.xml - Platform/Packages/Services/Car - Git at Google," https://android.googlesource.com/platform/packages/services/Car//master/service/AndroidManifest.xml, accessed Oct. 29, 2019.
34. "Transmissions - Craig's Website at Backfire.ca," http://craig.backfire.ca/pages/autos/transmissions, accessed Oct. 29, 2019.
35. Pesé, M.D., Stacer, T., Campos, C.A., Newberry, E. et al., "LibreCAN: Automated CAN Message Translator," 2019.
36. "Diagnostic Link Connector Security," https://doi.org/10.4271/J3138_201806.
37. Raja, A.V., Lee, J., and Gao, D., "On Return Oriented Programming Threats in Android Runtime," in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, IEEE, 2017, 259-2598.
38. Parikh, V. and Mateti, P., "ASLR and ROP Attack Mitigations for ARM-Based Android Devices," in *International Symposium on Security in Computing and Communication* (Singapore, Springer, 2017), 350-363.
39. Pesé, M.D., Schmidt, K., and Zweck, H., "Hardware/Software Co-Design of an Automotive Embedded Firewall," SAE Technical Paper 2017-01-1659, 2017.
40. Android Open Source Project, "Over-the-Air Updates: Android Open Source Project," https://source.android.com/devices/automotive/security/ota#android-otas, accessed Dec. 29, 2019.
41. Lin, T. and Chen, L., "Common Attacks against Car Infotainment Systems," July 2019, https://events19.linuxfoundation.org/wp-content/uploads/2018/07/ALS19-Common-Attacks-Against-Car-Infotainment-Systems.pdf.
42. Pesé, M. and Shin, K., "Survey of Automotive Privacy Regulations and Privacy-Related Attacks," SAE Technical Paper 2019-01-0479, 2019, https://doi.org/10.4271/2019-01-0479.

Contact Information

Mert D. Pesé, M.Sc.
University of Michigan
4956 Beyster Building, 2260 Hayward St. Ann Arbor, MI 48109-2121, U.S.A.
mpese@umich.edu

Kang G. Shin, Ph.D.
University of Michigan
4605 Beyster Building, 2260 Hayward St. Ann Arbor, MI 48109-2121, U.S.A.
kgshin@umich.edu

Josiah Bruner, B.S.
Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30318, U.S.A.
jsbruner@gatech.edu

Amy Chu, B.S.
Harman International
30001 Cabot Dr, Novi, MI 48377, U.S.A.
Amy.Chu@harman.com

Acknowledgments

I would like to thank Harman International for their support during my summer internship. It was a very creative learning environment for me to dig into a novel research area such as Android Automotive. More specifically, I would like to thank my mentors Josiah Bruner, Ravi Tamilarasan and Ben Jones, my manager Amy Chu, as well as the folks at TowerSec Israel, Golan Yosef and Asaf Atzmon who contributed with their valuable feedback.

Definitions/Abbreviations

IVI - In-vehicle infotainment

OEM - Original Equipment Manufacturer

IVN - In-vehicle network

CAN - Controller Area Network

TCU - Telematic Control Unit

UBI - Usage-based insurance
RPM - Revolutions per minute, engine speed
EVITA - E-safety vehicle intrusion protected applications
ECU - Electronic Control Unit
CV - Connected Vehicle
HVAC - Heating, Ventilation and Air Conditioning
VHAL - Vehicle Hardware Abstraction Layer
IPC - Inter-Process Communication
TEE - Trusted Execution Environment
ROP - Return-Oriented Programming
ART - Android Runtime
ADB - Android Developer Bridge

ISSN 0148-7191