

Full length article

Android Automotive OS under the hood: Architecture and security threat classification

Faiyaz Khan ^a, Hanbo Yu ^b, Junjie Wu ^b, Natalia Stakhanova ^{a,*}, Benjamin C.M. Fung ^b

^a University of Saskatchewan, Canada

^b McGill University, Canada

ARTICLE INFO

Keywords:

Android Automotive OS
AAOS
In-vehicle security
Permission model
Automotive cybersecurity
IVI systems

ABSTRACT

Android Automotive OS (AAOS) is becoming one of the most widely used in-vehicle infotainment (IVI) systems. Integrated into the Android ecosystem, it interfaces with vehicle functions to support advanced applications and third-party development. Despite its rapid adoption as the core platform for modern IVI systems, a comprehensive understanding of its security challenges remains lacking, and existing surveys have not thoroughly addressed these issues. To bridge this gap, we conducted a comprehensive analysis of AAOS by statically examining five consecutive releases (versions 11–15). Our study explores its layered architecture, from applications and system services down to the Hardware Abstraction Layer (HAL), and contrasts the actual implementation with official documentation, with a focus on security mechanisms. We also survey and analyze threats specific to Android Automotive, proposing a novel categorization framework that organizes these threats along three dimensions and classifies existing research accordingly. By mapping known vulnerabilities into our framework, we identify recurring issues such as inconsistent documentation, weak permission enforcement, and vendor-specific extensions, factors that may lead to privacy breaches or safety risks.

1. Introduction

In 2017, Google announced a dedicated operating system for in-vehicle infotainment (IVI) systems called Android Automotive OS (AAOS) (Weiss et al., 2021). AAOS was designed as a customizable, automotive-grade operating system that runs natively on in-vehicle hardware. This architecture enabled tight integration with vehicle-specific functionality and introduced capabilities not previously seen in vehicles. Among these capabilities were advanced entertainment features provided through the IVI system, allowing users to access both traditional media content and internal vehicle information from a unified interface.

AAOS also enabled third-party developers to create customized automotive applications. This significantly opened up traditionally restricted and isolated vehicle functions, giving users controlled access to various vehicle properties. AAOS was designed to support extensive customization by Original Equipment Manufacturers (OEMs), enabling tied integration with proprietary vehicle systems and services. This includes access to low-level vehicle communication protocols such as the Controller Area Network (CAN) bus, and the ability to tailor user interfaces and application behavior to specific car model requirements. This level of OEM-driven customization in vehicle software design

raised numerous questions around standardization, interoperability, and security.

Volvo Polestar 2 was the first vehicle to adopt AAOS. Over the next few years, major automakers, such as General Motors and Renault, started integrating AAOS into their vehicles. In 2022, Google expanded AAOS by offering Google Automotive Services (GAS), which offered Google Maps, Google Assistant, and the Google Play Store. Today over 26 vehicle brands adopt AAOS including BMW, Hyundai, Honda, Ford, and Nissan. Although AAOS is one of the most rapidly adopted operating systems in the automotive world, there exists several other automotive operating systems. QNX Neutrino RTOS is a real-time OS developed by BlackBerry. Automotive Grade Linux (AGL) is an open-source Linux-based platform supported by Toyota, Subaru, and others. Many automotive companies develop their own proprietary systems, e.g., Tesla OS, Tesla's proprietary system built on Linux, Mercedes-Benz MBOS, a platform for Mercedes vehicles, Volkswagen CARIAD, software platform used by Volkswagen Group's vehicles.

The increasing integration of complex software systems in modern vehicles has raised numerous concerns about vehicle security. Researchers have identified a growing number of vulnerabilities within the automotive industry (Elkhail et al., 2021). Despite extensive work

* Corresponding author.

E-mail address: natalia@cs.usask.ca (N. Stakhanova).

<https://doi.org/10.1016/j.cose.2026.105175>

Received 9 October 2025; Received in revised form 13 May 2026; Accepted 15 September 2026

Available online 23 September 2026

0167-4048/© 2026 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC license (<http://creativecommons.org/licenses/by-nc/4.0/>).

on vehicle cybersecurity, there remains a lack of comprehensive studies addressing the unique challenges introduced by Android Automotive.

In this article, we present a comprehensive survey of Android Automotive OS (AAOS), its layered architecture, and its security landscape. Although automotive security has received considerable attention in the research literature, prior survey efforts on AAOS are limited. Table 1 provides an overview of existing survey studies. Only two of these focus on Android Automotive. [Pesé et al. \(2020\)](#) provided a broad introductory overview of the AAOS architecture and associated threats. In 2022, [Moiz and Alalfi \(2022\)](#) offered a more focused analysis of security vulnerabilities in Android Automotive applications. However, both studies offer a narrow treatment of the topic and its attack surfaces, leaving critical gaps in our understanding of AAOS-specific vulnerabilities. The open source nature of AAOS, the extensive customizability and deep integration with vehicle systems further amplify security concerns. Modern vehicles, once largely isolated, are now exposed to a broad threat landscape, including interconnected networks, third-party applications, and wireless interfaces. We argue that a comprehensive, systematic, and up-to-date survey of existing research is essential to support and guide future work on Android Automotive OS security.

Our contributions in this work are as follows:

- This survey provides a granular examination of the Android Automotive OS (AAOS) architecture. We analyze its layered structure, from the application layer down to the Linux kernel, and offer practical insights by comparing real-world implementations with official documentation. The breadth and depth of our analysis have not been presented in any previous studies. To the best of our knowledge, this is the first work to offer a detailed security-focused insight into Android Automotive.
- We propose a classification of threats targeting Android Automotive. Although numerous studies over the past decade have revealed vehicle vulnerabilities and demonstrated successful attacks on in-vehicle infrastructure, existing threat classifications remain largely generic, for example, based on the CIA triad ([Lu et al., 2019](#)) and focus primarily on the physical infrastructure of the vehicle, such as sensor-based or CAN-bus-initiated attacks ([Aliwa et al., 2021](#)). In this work, we present an Android-specific threat classification. Using this framework, we classify, characterize, and compare existing studies. Our goal is not only to provide a foundation for analyzing the Android Automotive threat landscape, but also to highlight existing challenges and promising directions for future research.

The paper is structured as follows. In Section 2, we present a detailed overview of the Android Automotive layered architecture. Section 3 examines AAOS's built-in security mechanisms for protecting resources, including the app sandbox, the permission model, guarded APIs, and vehicle-property access controls. In Section 4, we introduce our Android specific threat classification framework, categorizing attacks by origin, mechanism, and potential damage. Section 5 then surveys existing research on different attacks on AAOS and categorizes them based on attack mechanisms. In Section 7, we summarize key take-aways from our analysis and outline promising directions for future research.

2. Android Automotive architecture

Android Automotive OS (AAOS) is designed as a layered architecture. On one hand, this design reflects and supports the unique demands of in-vehicle systems, such as real-time responsiveness and seamless driver interaction. On the other hand, the layers are intended to serve as modular building blocks that integrate with other components of the Android platform. AAOS is organized into three distinct layers, each responsible for a specific set of functionalities:

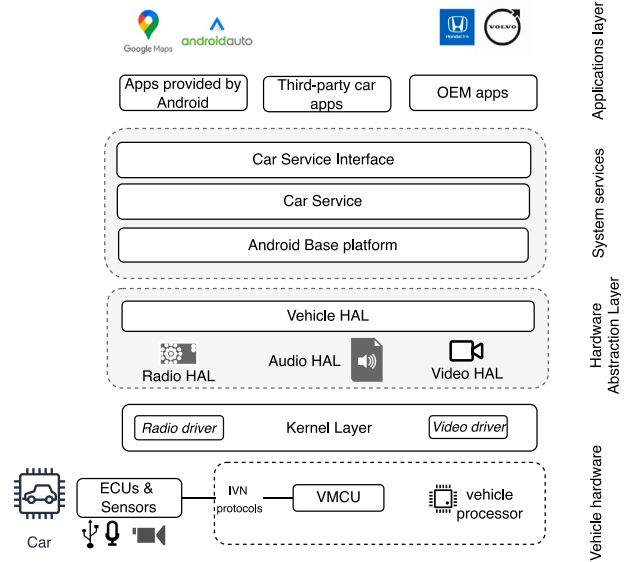


Fig. 1. Logical view of Android Automotive OS architecture.

- Hardware Abstraction Layer (HAL) that bridges the gap between hardware, e.g., sensors, camera, and software.
- System Services that provide core Android functionality tailored for the vehicle.
- Applications Layer that includes in-car apps such as navigation, media, and voice assistants, often customized by automakers.

AAOS builds on top of Android Open Source Project (AOSP), that contains the core source code for Android, including the operating system stack, and serves as the foundation of Android ecosystem. While AOSP provides the base Android platform, AAOS extends it with vehicle-specific features. This layered architecture allows AAOS to offer both the core functionalities of Android and the additional specialized functionalities needed for automotive use. Fig. 1 illustrates the conceptual organization of AAOS.

2.1. Applications layer

The applications layer in AAOS contains applications provided by Google, third-party applications and custom OEM applications. This layer operates within the constraints and privileges of AAOS and ensures that only verified applications interact with critical vehicle functions.

Android Applications. These include standard Android apps, e.g., browser, settings, which are adapted for AAOS to ensure seamless integration with the vehicle environment. These apps provide essential functionalities for users and serve as the backbone for the core infotainment experience. These are pre-installed system apps optimized for the automotive interface.

Google introduced a set of applications designed to support a safe and voice-driven in-car experience, known as *Google Automotive Services (GAS)* ([Gözübüyük et al., 2024](#)). By default, GAS includes Google Maps, Google Assistant, the Play Store, Search, Over-the-Air (OTA) updates, Voice and Natural Language Understanding (NLU), and Google Account integration. These applications can be licensed from Google and integrated into vehicle infotainment systems running Android Automotive OS (AAOS). However, this integration is optional, and AAOS images may include other Google apps that are not part of GAS. As opposed to Android apps designed for in-vehicle use, the *Android Auto* app serves as a platform for vehicle apps that run on the user's Android phone and are projected onto the car's display.

Table 1

Comparison of recent surveys on security in the automotive domain.

Survey	Year	Focus	Considered AAOS
Zheng et al. (2025)	2025	Attacks and defense mechanisms for keyless entry systems.	–
Ahmad et al. (2024)	2024	Security issues and ML-based defenses in smart vehicles.	–
Kifor and Popescu (2024)	2024	Frameworks and testing methods for automotive cybersecurity.	–
De Vincenzi et al. (2024)	2024	Security attacks and countermeasures in automotive ethernet.	–
Rajapaksha et al. (2023)	2023	AI-based intrusion detection for In-vehicle networks (IVNs).	–
Buscemi et al. (2023)	2023	CAN bus reverse engineering.	–
Lampe and Meng (2023)	2023	Deep learning based intrusion detection system (IDS) in In-vehicle networks (IVNs).	–
Fernandez de Arroyabe et al. (2023)	2023	Analysis of 537 papers focused on automotive security to identify research gaps.	–
Douss et al. (2023)	2023	Survey of in-vehicle network protocols, vulnerabilities and countermeasures for automotive ethernet.	–
Moiz and Alalfi (2022)	2022	Android automotive apps vulnerabilities.	✓
Talpur and Gurusamy (2022)	2022	Review of ML-based security approaches for IVNs.	–
Limbasiya et al. (2022)	2022	Analysis of attack detection and prevention systems for connected and autonomous vehicles.	–
Sun et al. (2022)	2022	Security of connected and autonomous vehicles.	–
Rathore et al. (2022)	2022	In-Vehicle communication security.	–
Tran and Nguyen (2022)	2022	Survey of lightweight cryptographic algorithms and protocols.	–
Jo and Choi (2022)	2022	Attacks on CAN.	–
Luo et al. (2022)	2022	Review of testing methods and testbeds in automotive domain.	–
Elkhail et al. (2021)	2021	Vehicle security & malware defenses.	–
Grimm et al. (2021)	2021	Context aware security for vehicles.	–
Luo et al. (2021)	2021	Threat analysis and risk assessment for connected vehicles.	–
Chatzoglou et al. (2021)	2021	Security of car management apps.	–
Mahmood et al. (2021)	2021	Survey of automotive security testbeds and testing methods.	–
Malhi et al. (2020)	2020	Security of vehicular ad-hoc networks.	–
Wu et al. (2020)	2020	Intrusion detection system (IDS) for in-vehicle networks.	–
Pesé et al. (2020)	2020	High-level overview of android automotive architecture and countermeasures.	✓
Lauser et al. (2020)	2020	Analysis of automotive protocols security.	–
Takahashi et al. (2020)	2020	In-vehicle infotainment system threat analysis.	–
El-Rewini et al. (2020)	2020	Vehicular communication security.	–
van der Sanden and Vasenev (2020)	2020	Architectural guidance for privacy and security in automotive systems.	–
Hao and Han (2020)	2020	Survey of security modeling methods for intelligent transport system.	–
Lu et al. (2019)	2019	Survey of security and privacy threats, and trust management models in Vehicular ad hoc networks	–
Young et al. (2019)	2019	Intrusion detection system (IDS) for CAN networks.	–
Sharma et al. (2019)	2019	CAN Security.	–
Cheng et al. (2019)	2019	Security design patterns for automotive cybersecurity	–
Sommer et al. (2019)	2019	Classification of automotive security attacks.	–
Mawonde et al. (2018)	2018	Comparative analysis of vehicle security approaches.	–
Li et al. (2018)	2018	In-vehicle network security (CAN, ECU, IVI).	–
Huber et al. (2018)	2018	Survey of security practices for integrating safety and security in automotive systems.	–
Jadhav and Kshirsagar (2018)	2018	Security in automotive networks.	–
Le et al. (2018)	2018	Security & privacy for automotive applications.	–

OEM Applications. Car manufacturers can customize Android applications or develop their own apps to provide infotainment-specific features. Examples include media players, navigation systems, vehicle diagnostics, advanced climate control, seat adjustments, and user profile management. These apps interact with AAOS-specific APIs to offer vehicle-integrated functionalities unique to each car brand.

Third-Party Applications. Third-party developers can create Android car apps. These apps can be installed directly on the IVI system through the Google Play Store or a vehicle OEM-specific app store, or they can be projected to the IVI display via Android Auto.

2.2. System services

In AAOS, the System Services layer connects application code to vehicle hardware. This layer consists of three main tiers: the Car Service

Interface, the Car Service, which provides the vehicle-specific service implementations, and Android base platform that is common across all Android ecosystems.

The Car Service Interface is represented by vehicle-specific APIs implemented with the ‘android.car’ package which is exposed through the public AAOS SDK library known as `car-lib`. The ‘androidx.car’ library, a part of AndroidX (Android Extension), provides a higher-level abstraction designed for automotive app development. It supports both generic features, such as apps designed for Android Auto, and vehicle specific tasks. For this reason, Google refers to ‘androidx.car’ as the Android for Cars App Library, or simply the Car App Library. The actual implementation of the Car Service resides in the platform under the CarService component.

The core of the Car Service implementation is split across two collaborating classes: `CarServiceImpl` and `ICarImpl` (both located at

packages/services/Car/service). While `CarServiceImpl` is responsible for registering the service with Android's service manager and linking to the vehicle HAL, `ICarImpl` provides the actual binder interface through which clients can invoke car specific operations. In other words, `CarServiceImpl` acts as the initializer of the service, whereas '`ICarImpl`' contains the concrete logic that binds together all individual vehicle related services.

When AAOS initializes, the car service implementation class `ICarImpl` dynamically creates and configures the different vehicle-specific services. This process involves setting up each service with the proper context, resource bindings, and dependencies required for it to operate correctly. The configuration process also takes into account the specific vehicle hardware and the associated capabilities available within the system. The implementation class also uses a feature controller to determine which services are enabled based on the vehicle's configuration and current operating mode. This gating ensures that only the relevant services are instantiated and made available to applications. If a vehicle does not support a particular feature, the interface will disable or not instantiate the associated service. The Car Service Interface also checks whether the caller has the necessary permissions. If the permission checks fail, it logs a warning and prevents the unauthorized access. Once a request passes the permission and feature checks, the interface delegates the request to the appropriate vehicle-specific service.

2.3. Hardware Abstraction Layer (HAL)

The Hardware Abstraction Layer (HAL) is an interface that enables AAOS to interact with the vehicle. HAL components interface directly with the Linux kernel, which operates as the system's low level control hub for vehicle hardware components.

AAOS HAL, also referred to as *Vehicle HAL* provides standardized APIs for communication between system services and the HAL implementations of various hardware components using AIDL (Android Interface Definition Language) which provides standardized access to vehicle properties such as engine status, climate control, and safety sensors ([Android Open Source Project, 2024](#)). In Android 12 and lower, a legacy version of AIDL called vehicle hardware abstraction layer (VHAL) was used ([Android, 2025b](#)). *Vendor-specific HAL implementations* (VHAL) handle the functionality of different hardware components.

The Vehicle HAL performs similar operations for vehicle-specific functionality. When Car Services request vehicle data through the VHAL, the Linux kernel communicates with the VMCU and other ECUs, translating the HAL's abstract commands into hardware-specific protocols. VHAL AIDL interface and provides VHAL logic generic to all hardware devices.

Service requests are mapped to specific HAL interfaces, which then translate these requests into low-level hardware instructions. The HALs ensure consistent communication with hardware, irrespective of hardware variations across vehicles. Data or status updates are channeled back up to the services, which format them into responses compatible with the API layer for applications to consume.

2.4. Linux kernel layer

At the base of AAOS's architecture is the Linux kernel, handling essential system operations and device management. It provides the critical infrastructure for resource management, task scheduling, and hardware communication. Automotive applications require real-time capabilities for timely responses to critical inputs. AAOS's kernel configuration typically includes real-time extensions, ensuring prioritized processing for time-sensitive tasks. The kernel includes device drivers specific to automotive needs, such as those for CAN bus communication, GPIO controls, and specialized sensors.

The Linux kernel interfaces with the Vehicle Processor and exchanges data with vehicle hardware components through the Vehicle Master Control Unit (VMCU) which communicates with the processor using low-level communication protocols such as Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), and General-Purpose Input/Output (GPIO) signals.

2.5. Vehicle hardware

The VMCU is a microcontroller that serves as the primary interface to in-vehicle networks and vehicle systems at the hardware level, such as the powertrain, body control, climate, lighting, and safety features. VMCU manages real-time data exchange between Android Automotive operating system and the vehicle's Electronic Control Units (ECUs), which control individual systems (e.g., powertrain, braking) and between ECUs ([Android Open Source Project, 2024](#)). ECUs collect data from sensors and execute commands issued by the VMCU. For example, an ECU responsible for climate control might receive a command from the VMCU to adjust the temperature. Sensors gather real-time data such as speed, temperature, and fuel levels, which is relayed to the VMCU when needed. This sensor data flows through the VMCU and is eventually passed to the Vehicle HAL and Car Services, enabling applications to access up-to-date vehicle information.

VMCU serves as the primary interface to in-vehicle networks, which enable communication among ECUs, sensors, actuators, and telematics components. As vehicle software and hardware have become more interconnected, manufacturers have adopted multiple in-vehicle network (IVN) protocols to support different performance and cost requirements ([Jadhav and Kshirsagar, 2018](#)). Common IVN protocols include Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, Media Oriented Systems Transport (MOST), and Automotive Ethernet ([Jadhav and Kshirsagar, 2018](#)).

Among these, CAN remains the dominant protocol for general-purpose in-vehicle communication because it offers a practical balance of reliability, cost, and flexibility ([Kim and Shrestha, 2020](#)). LIN is typically used for simpler and lower-cost subsystems that require only low data rates, often alongside CAN in larger vehicle networks ([Kim and Shrestha, 2020](#); [Aliwa et al., 2021](#)). FlexRay supports higher reliability and deterministic communication, making it more suitable for safety-critical applications, although its complexity and cost have limited broader adoption ([Kim and Shrestha, 2020](#); [Jadhav and Kshirsagar, 2018](#)). MOST was designed mainly for infotainment and multimedia communication, but its cost and sensitivity to node failures reduced its practicality in many deployments ([Jadhav and Kshirsagar, 2018](#); [Kim and Shrestha, 2020](#)). Automotive Ethernet provides substantially higher bandwidth and is increasingly used for data-intensive functions such as infotainment and ADAS ([Kim and Shrestha, 2020](#)). Together, these protocols form the communication backbone that connects the VMCU to the broader vehicle system.

3. Protecting resources

Access to protected resources in Android Automotive is tightly regulated. The platform employs several layers of protection to manage and restrict access to sensitive functionalities. At the core of this access control is the Android framework, which provides several protection mechanisms including sandbox environment for apps, the Android permission model that plays a crucial role by requiring apps to declare their intent to access certain resources and to obtain explicit user or system approval, and SDK methods that enforce security policies and abstract direct access to low-level vehicle components.

Beyond these standard Android mechanisms, Android Automotive introduces additional safeguards tailored to the unique requirements of vehicle environments. This includes access restrictions on vehicle properties, which are managed through the Vehicle HAL. To gain insight into this multilayered security framework, we perform a static

Table 2

AAOS release branches used for static analysis.

AAOS release	API	Branch	Source URL
AAOS 11	30	android11-release	https://android.googlesource.com/platform/packages/services/Car/+refs/heads/android11-release
AAOS 12L	32	android12L-release	https://android.googlesource.com/platform/packages/services/Car/+refs/heads/android12L-release
AAOS 13	33	android13-release	https://android.googlesource.com/platform/packages/services/Car/+refs/heads/android13-release
AAOS 14	34	android14-release	https://android.googlesource.com/platform/packages/services/Car/+refs/heads/android14-release
AAOS 15	35	android15-release	https://android.googlesource.com/platform/packages/services/Car/+refs/heads/android15-release

Table 3

Number of permissions across AAOS versions.

Protection level	AAOS 11	AAOS 12L	AAOS 13	AAOS 14	AAOS 15
Normal	6	7	7	8	8
Dangerous	2	2	2	2	2
Signature	4	4	4	0	0
System signature	1	1	0	0	0
Signature privileged	79	94	98	117	129
Signature privileged vendorPrivileged	0	0	0	3	3
Total	92	108	111	130	142

analysis of the source code for Android Automotive versions 11 through 15 (APIs 30, 32, 33, 34, and 35). Table 2 lists the AAOS release branches analyzed in our study.

We note that all source data was collected from public Android Open Source Project (AOSP)/Android Automotive OS release branches. For each version, we examined the same platform files. We used the `car-lib` source tree to identify public API declarations and annotations, `Car.java` to resolve permission constants into full permission strings, the corresponding manifest and permission declaration files to determine protection levels, and the vehicle property definition files to extract property to permission mappings. We analyzed each release separately. All counts in this section were generated by parsing source files.

3.1. Sandbox environment

The sandbox environment in AAOS isolates each application to prevent unauthorized access to system resources and user data. While similar to Android's standard app sandbox, AAOS's sandboxing integrates additional automotive-specific restrictions. Unlike typical Android apps, applications in AAOS require additional permissions to access sensitive vehicle data or interact with car hardware. For example, an app that needs access to the climate control system must explicitly request the `android.car.permission.CONTROL_CAR_CLIMATE`. Each app operates in its own memory and storage space, which prevents it from directly accessing the data or states of other apps. AAOS also limits the CPU, memory, and battery consumption of background applications to ensure that infotainment and other high-priority functions remain responsive (Android Open Source Project, 2024).

3.2. Permission analysis

AAOS introduces a more granular framework for categorizing car-related permissions compared to AOSP. Similar to how permissions are implemented as a system of safeguards in AOSP, AAOS also leverages a set of car-related permissions. These permissions are implemented using an Access Control List (ACL), which is designed to restrict access to sensitive resources and data. Traditionally, Android categorized permissions based on their granting time and required protection level.

According to their granting time, Android official documentation groups all permissions into several types (Developers, 2025):

- *Install-time permissions* that are automatically granted to the application when it is installed.

- *Runtime permissions* that are requested at the runtime of the app and allow access to restricted data and functions, such as calls, notifications, GPS location, etc. These require user's direct approval to be granted. In Android Automotive, the permission flow is similar to that on an Android phone, except that the user reviews and grants permissions through the car's IVI screen rather than a mobile device.
- *Special permissions* that can be defined and used by the Android platform and OEMs.

Based on their required protection level, Android categorized permissions as Android (2025a):

- *Normal permissions*. These grant access to resources with minimal risk to user privacy or device operation and are automatically granted at install time.
- *Dangerous permissions*. These involve access to sensitive user data or actions that could affect the user's privacy or device behavior. They require explicit user approval at runtime.
- *Signature permissions*. These are only granted to apps that are signed with the same certificate as the app that declares the permission.
- *KnownSigner permissions*. These are granted to apps signed with known certificates that are explicitly trusted by the platform (e.g., listed in an app lineage configuration).
- *SignatureOrSystem permissions*. Previously known as `signature|privileged` (deprecated in API level 23), these were granted to apps either signed with the same certificate or pre-installed in the system image.

These Android-defined permissions facilitate access to system resources for developers, and thus referred to as standard permissions. Beyond these, Android allows OEMs to include application-defined permissions known as custom permissions.

The standard permissions and their corresponding protection levels available in Android Automotive are listed in Table 3. The official documentation does not provide a comprehensive list of vehicle-specific permissions. Although a few car-related permissions are mentioned in the developer documentation in relation to used methods.

To compile a more comprehensive list of these permissions, we parsed the `AndroidManifest.xml` files available in each AAOS version. We extracted string literals that follow the Android permission naming convention, as well as comments present in the source code, following the XML structure illustrated in Fig. 2.

As shown in Table 3, the number of permissions has steadily increased over time, growing from 92 in Android 11 to 142 in Android

15. Of these, 88 permissions have remained unchanged across all five versions.

In contrast to AOSP, where permissions with normal and dangerous protection levels constitute the majority (Barzolevskaia et al., 2024), the most common protection level in AAOS is `signature|privileged`, which restricts access to privileged apps built into the system image. The number of permissions in this category rose from 79 in Android 11 to 129 in Android 15. In other word, vast majority of AAOS permissions can be only used by Google and OEM developers, and is not intended for third-party developers.

One permission, `RECEIVE_CAR_AUDIO_DUCKING_EVENTS` with `system|signature` level in AAOS 11 and AAOS 12 is not present anywhere beyond AndroidManifest file. However, the protection level of this permission changes to `signature|privileged` from AAOS 13. The presence of this permission may be a remnant of outdated or legacy source code.

Lower-risk normal permissions remain low (6 in AAOS 11 and 8 in AAOS 15) while the two dangerous permissions remained constant across all versions. These permissions are `CAR_ENERGY` that gives an application access to vehicle energy information, and `CAR_SPEED` that allows an application to read the vehicle speed.

In addition to these base levels, Android defines additional flags that can be added to the base level separated by '|'. One of them is `vendorPrivileged` which can only be granted to privileged apps in vendor partition and is similarly not available to third-party developers. Three permissions with protection level `signature|privileged|vendorPrivileged` appear in AAOS 14 and 15, which are `CAR_POWER`, `CONTROL_SHUTDOWN_PROCESS`, and `CONTROL_CAR_POWER_POLICY`. These permissions allow an application to control the vehicle power state and adjust the shutdown process, effectively, limiting access to the power of a vehicle to OEMs only.

Interestingly, the permission `CONTROL_SHUTDOWN_PROCESS` was first introduced with protection level `signature|privileged` in AAOS 13, which later was changed to `signature|privileged|vendorPrivileged` in AAOS 14. In addition to this, 2 additional permissions were added in AAOS 13 to control car energy systems (`CONTROL_CAR_ENERGY`), and privileged information access (`PRIVILEGED_CAR_INFO`) respectively. In Android Automotive release 14, 24 more permissions were introduced for ADAS settings, driver monitoring, remote display mirroring, and system-UI management. At the same time, 5 legacy permissions, for input services, instrument clusters, projection, VMS clients, and storage encryption seeds were removed. With the release of AAOS 15, the platform had defined a total of 147 unique permissions across all versions.

In addition to categorization by types and protection levels, AAOS incorporates *functional domains*, i.e., groups of vehicle features or sub-systems organized by functionality, such as location or media. Permissions associated with these domains govern which apps can access or control the corresponding vehicle functions. For example, AAOS allows OEMs to define permissions for accessing specific vehicle functional domains.

In all cases, an Android application must declare the permissions it requires to access restricted resources in its `AndroidManifest.xml` file.

3.3. Guarded APIs

One of the key mechanisms for controlling access to vehicle resources is a set of APIs protected by permissions. We identify relevant permission-guarded APIs through static analysis of all public API elements exposed by Android Automotive SDK via the Car library (car-lib). Specifically, we recursively traverse the car-lib source tree in each AAOS release, extracting all public API methods and their associated annotations using regular expressions. These expressions capture both Javadoc comments and all attached @-annotations, including tags embedded in `/*...*/` blocks. For each `@RequiresPermission` entry, we record not only the presence of the annotation but also the listed

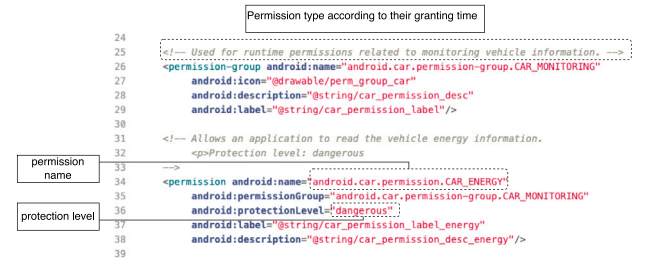


Fig. 2. An example of Android Automotive permission annotations in Android-Manifest.xml file.

constant(s). We then post-process the release's Car.java file to resolve these constants into their corresponding full permission strings. Finally, our parser maps each required permission to its protection level as declared in the manifest file.

The Android source code annotations serve as inline metadata that inform the build system, documentation generators, and platform tooling about how APIs should be exposed or restricted. They also describe the intended usage of permissions by Android developers and are used to generate Java code documentation in HTML format from the Android source code. Our analysis showed that the Android Automotive source code includes over 20 different tags. Most of these tags are undocumented, and even those that are documented lack clear explanations of their functionality.

Given a variety of undocumented labels, to facilitate the analysis we grouped annotation combinations into six higher-level categories. These categories reduce the complexity presenting a small, semantically meaningful set that directly reflects existing AAOS's security controls. Table 4 summarizes the API annotations present across different AAOS versions. The categories are:

- **FlaggedApi:** Captures all APIs annotated `@FlaggedApi`, with sub-groups indicating which are also `@SystemApi` or `@hide`. This isolates the new runtime feature-flag mechanism introduced in AAOS 15 and shows how many experimental interfaces are exposed or concealed.
- **SystemApi:** Includes every interface marked `@SystemApi`. Sub-combinations (`deprecated|hide`, `hide`, `RequiresApi|hide`, `TestApi|hide`) highlight the additional visibility or lifecycle constraints applied to system-only APIs.
- **RequiresPermission:** Collects all APIs that declare a permission requirement. Splitting by co-annotations (e.g. `deprecated`, `hide`, `FlaggedApi`, `SystemApi`, `TestApi`, `VisibleForTesting`, `removed`) reveals how enforcement overlaps with deprecation, hiding, and feature flags.
- **TestApi/VisibleForTesting:** Isolates APIs intended solely for testing, whether hidden or deprecated, to prevent mix with production interfaces.
- **Deprecated/hide / removed:** Aggregates all removal and hiding markers into one category which provides a single measure of the obsolete or concealed API surface.
- **No Annotations:** Represents the public APIs without any annotations, presumably highlighting their unrestricted nature.

As shown in Table 4, the evolution of Android Automotive is clearly reflected in the growth of API methods, from 2403 in AAOS 11 to 4963 in AAOS 15. The vast majority of these methods have no annotations (1669 in AAOS 11 and 3464 in AAOS 15), suggesting they are available to third-party developers. Among the annotated categories, deprecated or otherwise restricted methods (marked with `@hide` and/or `@removed`) constitute the largest portion. This category has steadily increased over time, from 338 methods (14%) in AAOS 11 to 510 methods (10%) in AAOS 15.

Table 4
API annotations across AAOS versions.

Annotation	AAOS 11	AAOS 12L	AAOS 13	AAOS 14	AAOS 15
FlaggedApi					
FlaggedApi	0	0	0	0	58
SystemApi	0	0	0	0	1
SystemApi hide	0	0	0	0	41
Total FlaggedApi	0	0	0	0	100
SystemApi					
SystemApi	1	1	1	35	12
deprecated hide	22	26	27	27	31
deprecated hide removed	0	0	1	1	1
hide	107	131	160	243	327
RequiresApi hide	0	0	0	3	0
TestApi	0	0	1	0	0
TestApi hide	11	15	21	0	0
Total SystemApi	141	173	211	309	371
RequiresPermission					
RequiresPermission	188	204	209	241	253
deprecated	1	1	9	10	20
deprecated hide	0	0	0	9	9
FlaggedApi	0	0	0	0	4
FlaggedApi SystemApi hide	0	0	0	0	29
FlaggedApi TestApi hide	0	0	0	0	4
hide	25	32	28	9	21
hide removed	0	0	0	2	2
SystemApi	0	0	0	1	0
SystemApi deprecated hide	0	0	3	3	4
SystemApi hide	26	45	43	61	132
SystemApi TestApi hide	2	2	10	0	0
TestApi hide	2	0	2	5	8
VisibleForTesting hide	0	1	1	1	1
Total RequiresPermission	244	285	305	342	487
TestApi, VisibleForTesting	11	12	19	17	31
Deprecated, hide, removed	338	388	431	439	510
No annotations	1669	1982	2322	3364	3464
Total API	2403	2840	3288	4471	4963

| indicates a combination of annotations.

Table 5
Overview of AAOS methods that require permissions (annotated @RequiresPermission).

Protection level	AAOS 11	AAOS 12L	AAOS 13	AAOS 14	AAOS 15
dangerous	7	7	10	10	13
normal	21	26	25	26	30
signature	0	1	1	2	2
signature privileged	211	245	256	278	410
signature privileged vendorPrivileged	0	0	0	13	16
signature privileged development	2	3	3	3	4
signature recents	1	1	5	5	7
signature installer	2	2	3	3	3
Not Defined	0	0	2	2	2
Total	244	285	305	342	487

System-only interfaces (@SystemApi), which are available exclusively to system developers, make up a growing fraction of the API library. The total number of @SystemApi entries increases from 141 in AAOS 11 to 371 in AAOS 15. The majority of these are also marked as hidden, which is consistent with their intended restricted use.

API methods that enforce Android security policies are annotated with the @RequiresPermission tag. Similar to other categories, the number of such methods increased from 244 in AAOS 11 to 487 in AAOS 15. Table 5 provides further details on this category, listing the permissions required to access these methods.

These methods mostly require signature|privileged permissions, e.g., 81% of methods in AAOS 15. This permission restricts access to API for only privileged apps built into the system image and signed with the same certificate, indicating that they come from the same developer. The number of methods that requires permissions of this protection level grows sharply in AAOS 15 (from 278 to 410), which

indicates a large influx of methods restricted to system signed, privileged apps in that release. On the other hand, the use of “dangerous” and “normal” permissions does not show much change (from 10 in AAOS 14 to 13 in AAOS 15 for dangerous permissions). The signature|privileged|vendorPrivileged category appears only in AAOS 14 (13 methods) and rises slightly in AAOS 15 (16 methods), showing that vendor level privileges were introduced in AAOS 14. Methods requiring just signature level permissions stay minimal over all versions, implying almost all signature protected methods also carry other restrictions.

During our analysis, we also encountered API methods that require permissions not documented in the official sources. We labeled them as ‘Not Defined’. These include two API methods: addCpuAvailabilityChangeListener and removeCpuAvailabilityChangeListener which require the COLLECT_CAR_CPU_INFO permission. According to the source code, this permission is necessary for accessing

Table 6
System vehicle properties.

Category	AAOS 11	AAOS 12L	AAOS 13	AAOS 14	AAOS 15
Display units	4	6	6	7	8
Doors/Seats/Windows/Mirrors	37	37	37	49	51
Driving & Chassis	7	7	7	17	19
Energy & EV	3	3	9	10	10
Engine	15	15	15	15	17
Exterior lights	12	12	16	16	16
HVAC	30	31	31	40	48
Info	11	11	11	11	11
Sensors & Status	4	5	5	5	10
Misc	14	27	29	54	64
Total	137	154	166	224	254

car CPU information, yet this permission is not mentioned in the official developer documentation and its protection level is not explicitly defined in codebase.

3.4. Vehicle properties

Similar to AOSP, which treats sensitive user data and components as privileged, Android Automotive protects access to privileged vehicle data and hardware controls. These controls are abstracted through *vehicle properties*, which serve as the primary interface that car apps use to access vehicle functionalities via the Vehicle HAL. Each vehicle property represents a specific type of data or control function such as vehicle speed, engine RPM, HVAC temperature, or door lock status, and is identified by a unique property ID.

Vehicle properties are categorized based on the vehicle subsystem they interact with. These categories include:

- HVAC (Heating, Ventilation, and Air Conditioning): properties related to climate control, such as cabin temperature, fan speed, defroster state, A/C on/off, recirculation status, dual-zone activation, and vent/heating setpoints.
- Engine: properties measuring or controlling engine internals, including engine RPM, gear selection, powertrain, fuel level, fuel capacity, coolant and oil temperature, oil level, and hybrid/electric battery status.
- Driving and Chassis: properties reflecting vehicle motion and handling, such as vehicle speed, wheel tick counts, steering angle, acceleration/brake pedal position, ABS/traction control states, and tire pressure.
- Doors/Seats/Windows/Mirrors: properties for physical access and occupant comfort, including door open/lock status, window position, sunroof position, seat adjustment parameters, seatbelt buckled indicators, and mirror position or fold state.
- Display Units: properties controlling or reporting unit systems—distance, fuel volume, tire pressure and night-mode toggles.
- Sensors and Status: environmental and system sensors such as odometer, outside temperature, diagnostic trouble code status, vehicle epoch time, and encryption binding seeds.
- Exterior Lights: lighting control and status properties, including headlights, high-beam, fog lights, hazard lights, reading and cabin interior lights, and their switch positions.
- Info: static vehicle identifiers and manufacturer data, including VIN, make, model, model year, multi-port connector capabilities, and chassis dimensions.
- Energy & EV: properties specific to electrified vehicles, such as EV connector type, instantaneous charge rate, range remaining, charging port open/connect state, and battery capacity metrics.
- Misc: any properties not fitting the above subsystems, including vendor-specific extensions, low-level HAL flags, or experimental data points.

Table 6 gives an overview of system vehicle properties defined by Google across different categories. As we can see, the total number of

system vehicle properties grows steadily from 137 in AAOS 11 to 254 in AAOS 15. However, not all categories grow at the same rate. For example, HVAC category grows substantially from 30 in AAOS 11 to 48 in AAOS 15, reflecting more properties are being added gradually for climate controls. In contrast, the Info category (static identifiers like VIN, make/model) remains at 11 across all versions.

The core protection mechanism for accessing vehicle properties is the AAOS permission system that defines the static mapping between vehicle properties and their corresponding permissions. Each property is associated with its access control, i.e., read, write, or both, and the corresponding permissions required for read and write access. For example, reading the value of the vehicle's low tire pressure CRITICALLY_LOW_TIRE_PRESSURE requires CAR_TIRES permissions. Since this property is not writable, it does not have a corresponding write permission.

Table 7 provides an overview of system vehicle properties and their corresponding protection levels. As the Android Automotive platform has evolved, the number of defined properties increased from 137 in AAOS 11 to 254 in AAOS 15. A large portion of these properties is protected by `signature|privileged` permissions, indicating that access to most vehicle hardware resources is restricted to system apps. The number of properties protected by `signature|privileged` level permissions climbs from 93 to 173 between AAOS 11 and AAOS 15, indicating that properties are gradually gated behind system- or vendor-level permissions. We also see properties with “Dangerous” permissions increasing from 7 to 13. Meanwhile, properties with no permission requirement grow from 8 in AAOS 11 to 19 in AAOS 14, before dropping to 18 in AAOS 15. In AAOS 11 there are exactly eight properties that do not require any permission. By AAOS 12 and AAOS 13, seven of those remain unchanged (all except `VEHICLE_MAP_SERVICE`), and eleven new read-only flags are added, bringing the total to 19. Those eleven remain constant through AAOS 14. In AAOS 15, `VEHICLE_MAP_SERVICE` and `HW_KEY_INPUT` are deprecated, and one new entry, `CLUSTER_HEARTBEAT`, is added with no permission requirement. There are five properties in AAOS 14 that share the permission string `android.car.permission.CAR_POWER` with protection level `signature|privileged|vendorPrivileged`:

- `AP_POWER_STATE_REQ` (ID 289475072)
- `AP_POWER_STATE_REPORT` (ID 289475073)
- `AP_POWER_BOOTUP_REASON` (ID 289409538)
- `DISPLAY_BRIGHTNESS` (ID 289409539)
- `PER_DISPLAY_BRIGHTNESS` (ID 289475076)

In AAOS 15, each of these five properties is deprecated, contributing to a total of 12 deprecated properties. Additionally, protection level `signature|privileged|vendorPrivileged` does not leveraged by any properties in AAOS 15.

Despite the rapid evolution of the Android Automotive platform, the official documentation has not kept pace. The permissions defined in the Android Automotive codebase often do not align with those described in the official developer documentation. While the documentation states that the platform defines 160 properties, including

Table 7
Standard vehicle properties by protection levels.

Level	AAOS 11	AAOS 12L	AAOS 13	AAOS 14	AAOS 15	Official documentation
Does not require permission	8	19	19	19	18	1
Normal	20	22	22	23	23	23
Dangerous	7	7	10	12	13	13
Signature Privileged	93	97	104	150	173	96
Signature Privileged, Normal	8	8	8	11	11	11
Signature Privileged, Dangerous	1	1	3	3	4	4
Signature Privileged VendorPrivileged	0	0	0	5	0	0
Deprecated	0	0	0	1	12	12
Total	137	154	166	224	254	160

Table 8
Vehicle properties in different OEM system images.

Type	GM (AAOS 11)	Honda (AAOS 12L)	Volvo (AAOS 11)
System properties	111	68	85
Vendor properties	13	431	16
Total	124	499	101

one that does not require a permission, our analysis reveals notable discrepancies across AAOS versions. Specifically, AAOS 11 contains 137 properties (8 without permissions), AAOS 12 has 154 (19 without permissions), AAOS 13 includes 166 (19 without permissions), AAOS 14 contains 224 (19 without permissions), and AAOS 15 defines 254 properties (18 without permissions). These figures diverge substantially from the documented total, highlighting inconsistencies between the official documentation and the actual implementation.

Although previous studies have highlighted the issue of inconsistent and outdated documentation in the Android ecosystem (Barzolevskaia et al., 2024; Silva et al., 2024), this problem appears to persist even in its most recent development — Android Automotive.

In addition to system properties, OEMs can introduce additional custom properties to support proprietary features unique to their vehicles. These OEM-introduced properties are known as *vendor properties*. Although Google does not explicitly reserve a specific range of property IDs for OEMs, it requires that these IDs be unique and not conflict with existing system-defined properties. To identify vendor vehicle properties, we compared all properties defined by Google against those found in OEM images.

Table 8 provides an overview of the system and vendor vehicle properties utilized in OEM AAOS images. While OEMs do not publicly disclose details about proprietary features, it is interesting to note that Honda relies heavily on custom properties to access vehicle hardware components, 431 out of the 499 properties used in their AAOS image are vendor-defined. OEMs can customize the property-to-permission mapping by introducing custom permissions and the corresponding custom vehicle property-to-permission mappings.

These OEM images also show how AAOS is configured in real production deployments. In practice, car makers do not ship AAOS exactly as it appears in AOSP. Instead, they select a subset of Google defined system properties, add their own vendor properties, and define custom property to permission mappings to support vehicle specific features. As a result, the security boundary seen in production vehicles depends not only on the base AAOS version but also on OEM integration choices. Our comparison of GM, Honda, and Volvo shows that these choices can vary significantly across vendors. This means that the exposed API surface, protected resources, and effective access control model in deployed vehicles may differ substantially from the reference AAOS source tree.

3.5. Summary

In this survey we provide a systematic security focused analysis of Android Automotive OS versions 11 through 15, with a focus on

architectural changes, security mechanisms, and emerging threat surfaces. Our findings reveal increasing complexity across versions and significant gaps between implementation and official documentation.

- **Documentation–Implementation Mismatch.** AAOS 15 defines 254 vehicle properties in source code (up from 137 in AAOS 11), yet the official documentation still lists only 160. Several permissions, such as `RECEIVE_CAR_AUDIO_DUCKING_EVENTS`, appear in AAOS 11–13 with elevated protection but are undocumented. Moreover, the number of properties that require no permissions increased from 8 (AAOS 11) to 19 (AAOS 12–14) and 18 (AAOS 15), while the documentation still references only one. These discrepancies can result in misconfigured apps and overlooked security enforcement. For example, such cases were recently demonstrated by Jumana et al. (2025), who identified anomalies in AAOS where different APIs access the same resource but enforce inconsistent access control policies (e.g., one API requires a specific permission, whereas another permits access to the same resource without enforcing that requirement).
- **Increasing Restrictiveness of Permissions.** The number of platform-defined permissions has grown from 92 (AAOS 11) to 142 (AAOS 15), with most now requiring signature or privileged access. As a result, third-party apps are left with minimal access to “normal” or “dangerous” permissions. OEM customizations further complicate the environment—for instance, Honda’s AAOS 12L build includes 431 vendor-defined properties compared to Google’s 68, many with custom and undocumented permissions.
- **Protected APIs and Feature Flags.** The count of APIs annotated with `@RequiresPermission` has nearly doubled, from 244 (AAOS 11) to 487 (AAOS 15). Many of these APIs are also marked with `@FlaggedApi`, making their availability and long-term support uncertain. Over 81% of protected APIs now require signature or privileged permissions, leaving only 13 as “dangerous” in AAOS 15 (up slightly from 7 in AAOS 11). We also identified undocumented permissions, such as `COLLECT_CAR_CPU_INFO`, which is required by two methods but lacks an explicit protection-level definition and is absent from the official documentation.
- **Expansion of Vehicle Properties and Hidden APIs.** The number of vehicle properties has increased from 137 to 254, covering areas such as HVAC, ADAS, and EV functions. While “dangerous” property permissions have grown from 7 to 13, those requiring signature or privileged access rose sharply from 93 to 173. AAOS documentation fails to reflect 94 of these properties, some of which expose sensitive vehicle states. Notably, five vehicle properties introduced in AAOS 14 with `signature | privileged | vendorPrivileged` permissions were removed in AAOS 15, breaking backward compatibility. In addition, AAOS 15 introduces over 100 undocumented `@FlaggedApi` methods and numerous `@hide` interfaces, expanding the system’s hidden attack surface.

While AAOS continues to expand its capabilities, its documentation, permission model, and vendor integration practices have not kept pace. These inconsistencies and undocumented elements although do not

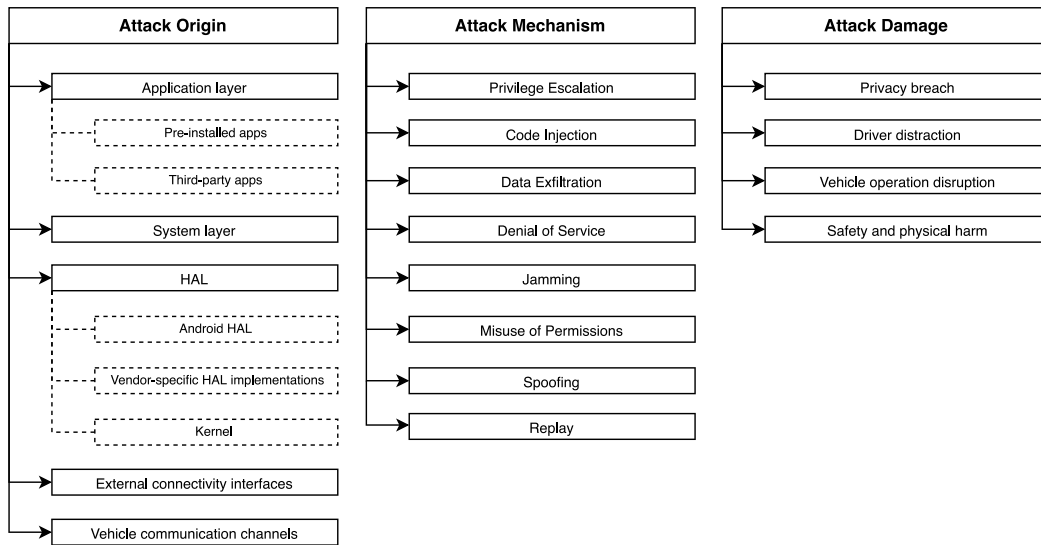


Fig. 3. Classification of attacks targeting Android Automotive.

constitute directly exploitable vulnerabilities, increase the likelihood of security risks by making it more difficult for developers and vendors to correctly implement and enforce access control policies. As a result, greater standardization, clearer specifications, and stronger oversight are needed in future AAOS releases. The anomalies resulting from inconsistent access control checks in AAOS were also recently pointed out by Jumana et al. (2025). The study showed that many of these inconsistencies stem from the evolution of the Android platform into AAOS, which introduced artifacts and gaps in access control enforcement, further indicating that AAOS developers lack sufficiently clear and comprehensive security specifications.

4. Classification of Android automotive attacks

In the past decade, researchers have demonstrated a variety of successful attacks on vehicle infrastructure. Existing classifications of these attacks tend to be broad, for instance, based on the CIA triad (Lu et al., 2019), targeting the vehicle's physical components, such as sensors or CAN-bus systems (Aliwa et al., 2021), or focusing on where the attacks are injected (Human interface, internal system processing, or internal in-vehicle network) (Gong et al., 2025). Taking a more software-oriented approach, specifically focused on Android Automotive, allows for a more practical and detailed analysis of the threat landscape.

The existing regulations, UN Regulation No. 155 (UN R155) (United Nations Economic Commission for Europe (UNECE), 2021) and ISO/SAE 21434 (International Organization for Standardization (ISO) and SAE International, 2021), primarily enumerate discrete security threats and corresponding mitigation measures that automotive manufacturers should consider. However, neither provides a comprehensive understanding of the overall automotive attack surface.

While research specifically targeting attacks on Android Automotive remains limited, several studies have proposed categorizations of these threats. For instance, Eriksson et al. (2019) classified attacks based on their impact on assets into three categories: privacy, disturbance, and availability of assets. Mandal et al. (2019) offered classification of Android infotainment threats by attack type and proposed the following categories: virus/malware, authentication, illegal setting, false information, jamming, tracking, driver distraction.

Despite these attempts, there is currently no universally accepted taxonomy tailored specifically to AAOS and its unique integration of Android systems with automotive functionality.

To support a more structured and comprehensive analysis of emerging threats in Android Automotive, we propose a new classification

framework. This framework organizes attacks along three orthogonal dimensions: *attack origin*, *attack mechanism* and *attack damage*. This classification is presented in Fig. 3.

4.1. Attack origin

The attacks can be classified based on their origin within the Android Automotive software and hardware stack, reflecting the component or interface from which the attack is launched or exploited. At the *Applications layer*, third-party apps, typically installed from app distribution platforms such as Google Playstore, and pre-installed (OEM or system) apps serve as a primary entry point. Malicious or compromised third-party apps may abuse exposed APIs, request excessive permissions, or bypass sandboxing mechanisms. Pre-installed apps and typically granted elevated system privileges and may be misused if improperly configured or exploited via vulnerabilities.

At the *System layer*, Android Automotive OS introduces several critical entry points that can serve as vectors for disruption of vehicle-integrated functionality. One of the primary components is the *SystemService*, which hosts core Android services such as the *ActivityManagerService*, *PackageManagerService*, and AAOS-specific services *CarService*. A uniquely critical system entry point in AAOS is the *CarService*, which orchestrates access to vehicle-related functionality through a suite of APIs, e.g., *CarPropertyManager*, *CarSensorManager*. These system services serve as interfaces to vendor-specific HALs to expose access to vehicle-specific functionality, and thus could be leveraged to influence vehicle behavior.

The *Hardware Abstraction Layer (HAL)* category includes Android, Vendor-custom HALs and kernel layer. Attacks originating at this level are particularly dangerous because they can bypass higher-layer security mechanisms by interacting directly with hardware or system drivers. Vulnerabilities in poorly isolated or misconfigured vendor HALs may allow unauthorized access to critical vehicle functions (e.g., speed, gear, or ignition control) or leakage of sensitive sensor data.

External connectivity interfaces expand the attack surface by enabling remote or nearby attacks, e.g., through malicious file injection, diagnostic port access, or network-based exploits. These interfaces connect vehicle to the outside and are intended for connectivity, diagnostics, or user interaction. This category includes attacks leveraging Bluetooth, WiFi, USB ports, diagnostics ports (OBD-II), remote Over-the-Air (OTA) updates, and Vehicle-to-Vehicle (V2V) communication often used by the Telematics Control Unit (TCU). Note that device interoperability platforms such as Android Auto also leverage these interfaces.

Finally, *vehicle communication channels* are internal, in-vehicle networks used by Electronic Control Units (ECUs) to communicate with each other. In AAOS, some of these channels may be exposed via HALs and Car APIs. These include CAN bus, FlexRay, Ethernet-based vehicle network, Attacks originating here may involve message spoofing, or misuse of interfaces exposed through AAOS integration with in-vehicle networks. These attacks are particularly sensitive as they can affect safety-critical systems and real-time vehicle behavior.

4.2. Attack mechanisms

AAOS attacks leverage a range of mechanisms. We have defined the primary mechanisms by which attackers compromise Android Automotive systems. Each mechanism targets different layers ranging from the application layer to system services and illustrates how vulnerabilities can be exploited to escalate privileges, inject code, exfiltrate data, disrupt services, or abuse platform features.

- *Privilege escalation* is a broad category that attacks aiming to bypass the operating system's permission boundaries and gain elevated access, for example, by exploiting vulnerabilities in system services or by circumventing Android's SELinux enforcement or app sandboxing to access protected resources.
- *Code injection* attacks inject or execute arbitrary code within privileged system components by utilizing memory corruption bugs, e.g., buffer overflows, in native code libraries, system or HAL modules, exploiting unsecured IPC interfaces or vulnerable third-party services.
- *Data exfiltration* are leakage attacks aim to extract sensitive information from the vehicle environment by for example accessing vehicle telemetry, location data, or personal identifiers through improperly protected interfaces, or abusing logging mechanisms or misconfigured data storage to gather private information.
- *Denial of service* attacks aim to reduce vehicle system reliability and availability by overloading system resources, for example, by crashing the IVI system, freezing the UI, or overloading vehicle core services, repeatedly triggering system exceptions or excessive resource consumption to impair functionality.
- *Jamming* attacks disrupt communication by interfering with wireless or network signals, typically through the emission of radio frequency interference. These attacks target wireless communication channels such as Bluetooth, Wi-Fi, cellular, and Vehicle-to-Everything (V2X) protocols, effectively blocking or degrading data transmission. In the context of Android Automotive, jamming can hinder essential functions like smartphone pairing, internet connectivity, and remote diagnostics. More critically, it can disable safety-related features such as emergency calls or remote software updates by blocking communication links.
- *Misuse of permissions*. This is an Android-specific category that focuses on attacks arising from the abuse of the AAOS permission model. If improperly managed, the permission system can expose critical system resources to underprivileged or misconfigured applications. For example, attacks may exploit custom OEM-defined permissions which often lack consistent enforcement or sufficient documentation creating opportunities for unauthorized access to sensitive vehicle functions or data.
- *Spoofing* attacks in Android Automotive involve an adversary impersonating a trusted component, device, or user to inject false data or gain unauthorized access to vehicle functions. These attacks often exploit weaknesses in authentication mechanisms or data validation processes within the vehicle's software architecture. For example, attackers may spoof CAN bus messages to simulate sensor values or control commands, potentially influencing decisions made by the vehicle's control systems. Spoofed location data can mislead navigation apps, while false speed readings may affect driver behavior or be used to manipulate insurance assessments. Additionally, malicious apps may mimic system dialogs or notifications to trick users into granting sensitive permissions or inputting credentials, resulting in privacy breaches or unauthorized privilege escalation.
- *Replay* attacks in Android Automotive involve intercepting and reusing valid data transmissions to perform unauthorized actions. These attacks exploit the absence of message freshness checks, such as timestamps, nonces, or sequence numbers, allowing an adversary to record a legitimate command or message and replay it at a later time to achieve the same effect, often without detection. In the context of Android Automotive, replay attacks can target several layers:
 - VHAL. If an attacker gains access to the vehicle's internal communication network (e.g., via compromised IVI firmware or exposed debug interfaces), they can replay CAN signals routed through the Vehicle HAL (VHAL). For example, previously captured commands for unlocking doors, adjusting HVAC settings, or changing drive modes can be resent to manipulate the system. Since many CAN messages lack built-in authentication or anti-replay mechanisms, such attacks are feasible without deep protocol knowledge.
 - Inter-component communication (ICC). In Android Automotive apps, attackers may attempt to replay previously intercepted intents if components do not validate the origin or content freshness. While Android's permission system may mitigate some of these risks, improperly secured components remain vulnerable if they accept untrusted inputs.
 - External communication. For example, if OTA update mechanisms or remote diagnostic services do not enforce session validation or replay protection, attackers could potentially resend previously captured update or configuration commands. This is especially critical in scenarios where the IVI system interfaces with cloud services over improperly authenticated channels.

4.3. Attack damage

Classifying Android Automotive attacks based on potential damage focuses on the impact rather than the technical surface or method. This approach is often used in risk assessment. Based on type and severity of damage they could cause, we classify attacks into four categories:

- *Privacy breaches*, these compromise user data, location, or personal vehicle settings. For example, these attacks can harvest user behavior (e.g., media habits, climate preferences, call history), access GPS, telemetry, or trip history, leak information through compromised third-party apps.
- *Driver distraction* represents a class of lower-severity attacks that divert the driver's attention, degrade the user experience, or act as a precursor to more harmful exploits. These may include disruptions to media playback, or driver assistance interfaces, e.g., changing themes, settings, or user interface colors, triggering repeated notifications, pop-ups, or sounds.
- *Vehicle operation disruption*, these attacks degrade vehicle functionality, usability, but may not cause immediate harm. For example, crashing or freezing the IVI system, causing vehicle operating system reboot, blocking legitimate AAOS updates or triggering downgrades.
- *Safety and physical harm* attacks represent the most severe category, as they directly impact driver or passenger safety and may cause physical harm. These attacks include manipulation of critical ECUs via the CAN bus from compromised AAOS components, the injection of false speed or position data, and unauthorized control of HVAC systems, defoggers, lights, that in hazardous conditions can pose serious risks to vehicle occupants and others on the road.

Table 9
Classification of attacks based on the damage they cause.

Category	Example attack types	Severity
Privacy breach	GPS tracking, profile theft, trip data leakage	Low–High
Driver distraction	Audio disruptions, visual clutter, UI abuse	Moderate
Vehicle operation disruption	IVI crash, OTA corruption, app freezing	High
Safety and physical harm	Steering override, brake spoofing, sensor manipulation	Catastrophic

Table 10
Attacks on Android automotive.

Attack	Year	Attack origin	Attack mechanism	Attack damage	Test env.
Insecure OTA update mechanism (Oh et al., 2025)	2025	External connectivity interfaces	Code injection	Vehicle operation disruption	AAOS (Android version not specified)
Exploiting custom permission at run time (Moiz and Alalfi, 2022)	2022	Application layer (3d party apps)	Data exfiltration	Privacy breach	Android Auto and emulation (Android 9.0)
Deriving speed from location information (Moiz and Alalfi, 2022)	2022	Application layer (3d party apps)	Data exfiltration	Privacy breach	Android Auto and Automotive Emulators (Android 9.0)
Bluetooth HCI snoop log exploit (Renganathan et al., 2022)	2022	System layer	Data exfiltration	Privacy breach	Production vehicle with Android 6.0.1 (OEM undisclosed) and an emulation of AAOS 11
Intent collusion (Moiz and Alalfi, 2020)	2020	Application layer (3d party apps)	Data exfiltration	Privacy breach	Theoretical analysis
Intent eavesdropping (Moiz and Alalfi, 2020)	2020	Application layer (3d party apps)	Data exfiltration	Privacy breach	Theoretical analysis
Intent hijacking (Moiz and Alalfi, 2020)	2020	Application layer (3d party apps)	Data exfiltration	Privacy breach	Theoretical analysis
Intent spoofing (Moiz and Alalfi, 2020)	2020	Application layer (3d party apps)	Spoofing	Privacy breach	Theoretical analysis
Insecure OTA update mechanism (Pesé et al., 2020)	2020	External connectivity interfaces	Code injection	Vehicle operation disruption	AAOS (Android Oreo 8.1)
Permissionless exfiltration (Eriksson et al., 2019)	2019	Application layer	Data exfiltration	Privacy breach	Volvo's IVI emulators and physical test bed (Android 8.0 and 8.1)
Indirect speed data leakage (Eriksson et al., 2019)	2019	Application layer	Data exfiltration	Privacy breach	Volvo's IVI emulators, physical test bed (Android 8.0 and 8.1)
Fork bomb attack (Eriksson et al., 2019)	2019	Application layer (3d party apps)	Denial of service	Vehicle operation disruption	Volvo's IVI emulators and physical test bed (Android 8.0 and 8.1)
Intent storm attack (Eriksson et al., 2019)	2019	System layer	Denial of service	Vehicle operation disruption	Volvo's IVI emulators and physical test bed (Android 8.0 and 8.1)
SoundBlast (Audio channel hijacking) (Eriksson et al., 2019)	2019	Application layer (3d party apps)	Misuse of permissions	Driver distraction	Volvo's IVI emulators and physical test bed (Android 8.0 and 8.1)
Climate control API covert channel (Eriksson et al., 2019)	2019	Application layer (3d party apps)	Misuse of permissions	Privacy breach	Volvo's IVI emulators and physical test bed (Android 8.0 and 8.1)
Android app-repackaging attack (Lee et al., 2019)	2019	Application layer (3d party apps)	Code injection	Vehicle operation disruption	Production vehicle (OEM undisclosed)
WebView-based JavaScript injection (Mandal et al., 2018, 2019)	2019	Application layer (3d party apps)	Code injection	Privacy breach	Theoretical analysis

The overview of attack damage categories with examples is shown in Table 9.

5. Survey of existing AAOS threats

Building on the proposed classification, this section groups existing research on Android Automotive attacks based on their execution mechanisms. We categorize based on attack mechanism as it reveals recurring patterns across different attack origins and also highlights weaknesses that enable similar exploits across OEM implementations (see Table 10).

5.1. Code injection

Code injection attacks in Android Automotive systems exploit vulnerabilities across multiple architectural layers, ranging from application layer to low-level hardware interfaces, enabling adversaries to

disrupt vehicle operations through unauthorized command execution. These attacks leverage memory-unsafe code, insufficient input validation, and insecure communication channels to inject malicious payloads into critical subsystems.

An illustrative example of a code injection attack was presented by Lee et al. (2019). As shown in several prior studies, vehicle CAN bus communication is unprotected and therefore susceptible to code injection attacks (Bozdal et al., 2018). In their work, Lee et al. introduced a repackaged Android app as a new entry point to deliver the malicious payload to the in-vehicle's internal communication networks. The attack leveraged wireless communication module connected to a vehicle's diagnostic On-board diagnostics (OBD)-II interface to send arbitrary commands to CAN bus. OBD communication requires knowledge of car manufacturer's OBD parameter IDs (PIDs), which can be extracted from an official vehicle diagnostic app, as demonstrated in the study. By obtaining such an app from an official app distribution platform, an attacker can retrieve the necessary information to prepare

a malicious payload in advance. This payload can then be embedded into a repackaged Android app, which may be distributed through Android marketplaces and installed by unsuspecting victims. Experimental validation was conducted on an undisclosed vehicles equipped with Android infotainment systems showed the critical consequences of such attack. The authors were able to successfully unlock the door, turn the engine off, and turn the engine off at a specified vehicle speed.

The potential for code injection attacks in Android Auto apps was also explored by Mandal et al. (2018, 2019). Their studies identified vulnerabilities related to the improper use of WebView instances, which could enable arbitrary JavaScript injection. Such issues allow malicious scripts to invoke an app's methods or silently inject code by loading remote HTML content leading to unauthorized code execution. Of the 23 Android Auto apps analyzed, 6 were found to be vulnerable to at least one of these JavaScript-related code injection threats (Mandal et al., 2019).

Another category of code injection attacks targets the remote over-the-air (OTA) update mechanisms in Android Automotive, potentially exposing vehicles to firmware tampering and system compromise. Security challenges related to OTA updates in connected vehicles have been widely discussed in the literature (Halder et al., 2020). Unlike typical code injection attacks that exploit individual apps or communication channels, this type of attack focuses on AAOS's update infrastructure. The resulting damage can cause disruption to vehicle operation, since a compromised firmware image can modify or disable critical subsystems. Pesé et al. (2020) presented a driver safety attack stemming from an availability of Vehicle HAL (VHAL) mapping table in the IVI firmware. An attacker with access to the firmware image can reverse-engineer the embedded configuration files to identify the CAN signal for a critical function, e.g., the gas pedal position, they can capture and modify OTA update by swapping critical signal with a benign command (e.g., a dashboard display update), resulting in unintended vehicle acceleration while the instrument cluster continues to display normal behavior. Similar attacks on OTA updates were introduced by Oh et al. (2025). Their work focuses on insufficient security in network communication, demonstrating that OTA updates can be modified through man-in-the-middle (MitM) attacks, for example by downgrading network protocols or manipulating TLS certificates.

5.2. Data exfiltration

Data exfiltration attacks in Android Automotive systems compromise user privacy through unauthorized extraction of sensitive data via insecure inter-component communication, improper storage practices, and covert transmission channels. The existing studies explored threats that originate at both the application and system layers.

Renganathan et al. (2022) demonstrated a system-layer information leakage attack by exploiting the Bluetooth Host Controller Interface (HCI) snoop log. The attack leverages Android's developer options, specifically the HCI snoop log, which records data from Bluetooth profiles such as the Phone Book Access Profile (PBAP) and Message Access Profile (MAP) in plaintext. As a result, the log may contain sensitive information, including contact lists, text messages, and call histories of users who previously paired their phones with the IVI system. This log is generated upon Bluetooth pairing and persists until the vehicle is powered off. Due to improper memory clearing by the CarPowerManager service, the log remains accessible throughout the vehicle's operation. Consequently, an attacker with brief physical access to a vehicle, given that developer options are enabled, can extract the unencrypted log via USB or the Android Debug Bridge (adb). The attack was validated on a production vehicle running Android 6.0.1 (OEM undisclosed) and on an AAOS 11 build emulated on a Raspberry Pi 4B. The vulnerability was found to be present in all Android versions from 4.2 onwards that support developer options.

Moiz and Alalfi (2020) presented several attacks targeting Android Automotive. Among these, intent hijacking, intent eavesdropping, and

intent collusion are examples of data exfiltration threats. Each of these attacks leverages the platform's inter-component communication (ICC) mechanism to leak sensitive vehicle data. While Android apps are protected by a sandbox environment that prevents direct access by other apps, ICC provides a system-level mechanism that enables different components (within or across apps) to communicate and share data using messages called intents.

In *intent hijacking*, a malicious app registers one of its components with intent filter attributes that match those of an authorized receiver. When another application dispatches an intent containing sensitive information, such as vehicle speed, location, or contact details, without properly restricting the set of permitted receivers, the attacker's component can intercept the intent and gain access to the data, even without holding the required permissions.

In *intent eavesdropping*, a legitimate application broadcasts an intent intended solely for its own components but fails to enforce receiver permissions. A malicious app installed on the same system can register for the same broadcast action and receive the sensitive data. This allows the malicious app to escalate its privileges by passively listening to and capturing incoming intents, again without requiring explicit permissions.

Lastly, unlike hijacking or eavesdropping, where a single malicious app both obtains and transmits data, *intent collusion* involves two cooperating apps. One app holds permissions to access sensitive vehicle properties, while the other possesses internet access but lacks permission to read vehicle data. The first app sends an intent containing the sensitive information, which the second app, having registered for the same broadcast action, receives and then transmits to a malicious server.

Although these presented threats remained theoretical, the authors demonstrated the widespread presence of intent leaks by statically analyzing 273 Android Auto apps (236 from Google Play and 40 reverse-engineered), uncovering 402 leaks across 121 apps.

Data leakage attacks leveraging ICC were also examined by Eriksson et al. (2019). In the permissionless exfiltration attack, the authors demonstrated how intents can be abused to extract sensitive information without requiring direct permissions. One such approach involves passing sensitive data through an intent URI. A malicious app lacking the INTERNET permission can encode data in the URI while masquerading as a benign component, for example, by setting the MIME type (e.g., audio/wav) to trigger a browser or media player that already has network access. In response, the system launches the appropriate handler, which automatically retrieves the URI from an attacker-controlled server, thereby leaking data without the malicious app ever requesting network access. The proof of concept was demonstrated on Volvo IVI emulators with Android 8.0–8.1. An Android app with only the RECORD_AUDIO permission recorded five seconds of audio, which was then exfiltrated via browser, the stock music player, video player, and image viewer.

Eriksson et al. (2019) also presented an attack for indirect speed data leakage. Although vehicle speed is considered privacy-sensitive and typically requires permissions, apps can infer it indirectly. By using permissionless sensor data such as engine RPM and gear, along with knowledge of the wheel size, speed can be approximated. However, this evaluation was limited to the IVI unit and did not involve a full vehicle system.

Moiz and Alalfi (2022) introduced *Deriving speed from location information* attack. This attack exploits the fact that while Android Automotive restricts direct access to vehicle speed data, apps can still infer speed using location data. By calculating the distance between successive GPS coordinates over time, a malicious app can derive a vehicle's speed, acceleration patterns, and braking behavior. Since many apps commonly request location access (with user consent), this covert method of speed estimation bypasses the need for privileged car data permissions presenting privacy risks. The attack is practical and has been successfully reproduced in both Android Auto and Android Automotive emulators.

5.3. Denial of service

Denial-of-service (DoS) attacks targeting Android Automotive OS exploit systemic weaknesses in resource management and inter-process communication to disrupt vehicle operations by overwhelming critical system components. These attacks originate at the system layer, where insufficient safeguards against resource exhaustion allow adversaries to degrade or disable infotainment and safety-critical functions.

Eriksson et al. (2019) demonstrated that AAOS remains vulnerable to *Fork bomb* attacks despite process creation limits. The authors showed that by leveraging a shell context, it is possible to spawn recursive child processes that exhaust CPU and memory resources. Tests conducted on both emulators and physical IVI units resulted in complete system freeze, rendering infotainment system unresponsive until the system was rebooted.

The Android Intent system can also become a vector for DoS attacks when abused at scale. Eriksson et al. (2019) introduced the *Intent Storm attack*, which demonstrated that launching an app's main activity from dozens of threads forces the system server, a privileged process responsible for managing app lifecycles, into 100% CPU utilization. The authors observed that this caused the system server to become overwhelmed, rendering Volvo's IVI unit emulators unusable. In some cases, the emulators automatically restarted after a few minutes.

Both DoS attacks are critical because they target essential components of the vehicle's software infrastructure, leading to IVI system instability or complete unresponsiveness. Although these attacks do not directly affect the vehicle's driving systems, the resulting disruption of the infotainment system undermines driver focus presenting a safety concern.

5.4. Misuse of permissions

The misuse of permissions in Android Automotive enables attackers to exploit APIs beyond their intended scope, bypassing security controls and compromising privacy and safety.

Eriksson et al. (2019) demonstrated the *SoundBlast* attack, where a malicious app abuses the permissionless AudioManager API to hijack audio channels. This API, which controls audio streams, can be exploited to monitor volume changes. Once a predefined vehicle speed threshold is exceeded, the app can force the volume to maximum, taking advantage of the API's lack of permission checks. This abrupt and unauthorized volume manipulation at high speeds creates driver distractions. The authors validated the attack on Volvo IVI emulators and infotainment testbeds, confirming that the volume override persists even when using hardware controls (e.g., steering wheel buttons or IVI knobs). Full control can only be restored by terminating the malicious app.

A more subtle attack described by Eriksson et al. (2019) leverages the *Climate Control* API as a covert channel, also relying on misused permissions. In this scenario, the API is repurposed to transmit data between colluding malicious apps to leak sensitive information. For example, one app may have permission to access sensitive data but not the Internet, while another has Internet access but no permission to read sensitive data. The two apps can communicate by encoding information into floating-point temperature values. The sender app writes these values using the Climate Control API, while the receiver app reads them and exfiltrates the data, e.g., via the Internet, without ever directly accessing the protected sensor data itself.

5.5. Spoofing

An example of a spoofing attack was presented by Moiz and Alalfi (2020), focusing on Android *Intent spoofing*, which exploits the platform's Inter-Component Communication (ICC) model. In this attack, a malicious app forges intents to impersonate system events or trusted messages typically sent by other legitimate applications. Moiz et al.

highlight that if an app component does not verify the origin of an incoming intent, e.g., through permission checks or signature validation, it can mistakenly treat a forged intent as legitimate.

This allows an attacker to send a crafted intent to a vulnerable component within another app, tricking it into executing unintended operations. As a result, the malicious app may be able to trigger actions like leaking sensitive data, altering application behavior, or even crashing the target app. Because the receiving component assumes the intent came from a trusted source, the attacker gains control over certain app behaviors without needing elevated permissions.

5.6. Summary

Research on Android Automotive threats remains limited, largely due to the relative immaturity of the field. However, our review highlights a growing body of work that has uncovered a range of security vulnerabilities in Android Automotive systems. These findings emphasize the increasing complexity and expanding attack surface of modern vehicles as they become more reliant on Android-based architectures.

- **Attack origin** The vast majority of documented attacks originate from the application layer, particularly through third-party apps delivered through app distribution platforms. System-level attacks that exploit deeper integration points in AAOS are rare.
- **Common Attack Mechanisms** Across all layers, common mechanisms include misuse of permissions, code injection, spoofing, and data exfiltration. These attacks primarily result in privacy breaches, though some, like OTA tampering and DoS attacks, can disrupt core vehicle functions.
- **Common Attack Damage** The majority of attacks targeting AAOS result in privacy breaches, leaking sensitive vehicle and driver information. A smaller subset of attacks can disrupt vehicle operation, representing a more severe form of damage. However, none of the documented attacks so far have demonstrated catastrophic impact, such as causing safety hazards or physical harm.
- **Test Environments** Most of these attacks have been validated in emulators or IVI testbeds, highlighting their feasibility in real-world deployments without requiring access to production vehicles. However, the attacks documented in this survey were primarily demonstrated on early AAOS versions (e.g., AAOS 8.0–8.1) and on limited OEM testbeds (e.g., Volvo IVI emulators). Future research should systematically port these proof-of-concept exploits to newer AAOS releases to evaluate whether the vulnerabilities have been effectively mitigated. Extending these attacks across newer AAOS versions and multiple OEM implementations would help determine which flaws remain exploitable.

The existing body of research highlights that Android Automotive inherits many of the security weaknesses of mobile Android, while also introducing new risks stemming from its integration with vehicle subsystems. Addressing these issues requires not only stronger platform-level protections but also stricter validation mechanisms for both applications and OEM-specific extensions. However, research on defensive strategies for Android Automotive remains largely absent and warrants significant attention from the research community.

6. Case study of the existing AAOS vulnerabilities

To evaluate the effectiveness of our attack categorization framework, we analyzed all Android Automotive OS vulnerabilities published in the CVE database over the past five years (2022–2026). The overview is presented in Table 11. Based on the provided CVE description, we classified the vulnerability following its origin, employed primary attack mechanism, and the potential attack damage. This process allowed us to validate that our categories cover the spectrum of real-world AAOS threats.

Table 11
Classification of published Android Automotive vulnerabilities.

CVE	Description	Attack origin	Attack mechanism	Attack damage
CVE-2023-21260	Misleading information can be presented to user as a system message.	Application layer (apps)	Privilege escalation	Driver distraction
CVE-2023-20927	App can be granted signature permissions due to a permissions bypass.	Application layer (apps)	Privilege escalation, Misuse of permissions	Depending on the permission obtained, damage can range from privacy breach to driver's safety concerns
CVE-2022-20458	User's sensitive information (e.g., user's account name) can be printed in logs	Application layer (apps)	Privilege escalation	Privacy breach
CVE-2022-20429	A confused-deputy vulnerability in Android's CarSettings component that allows a malicious app to trigger privileged Bluetooth configuration operations	Application layer (apps)	Privilege escalation, Misuse of permissions	Privacy breach, Driver distraction
CVE-2022-20234	An app can trick users into granting notification access, enabling it to read all notifications.	Application layer (apps)	Privilege escalation, Misuse of permissions	Privacy breach
CVE-2022-20215	An app can bypass permission checks in the CarSettings component to invoke privileged operations causing system instability.	Application layer (apps)	Denial of service	Vehicle operation disruption
CVE-2022-20214	Tapjacking attack. An app can overlay the toggle button to enable apps to modify system settings without user consent	Application layer (apps)	Privilege escalation	Driver distraction
CVE-2022-20213	Vulnerability in Android Automotive's CarSettings component that allows a malicious app to bypass permission checks and access privileged operations	Application layer (apps)	Denial of service	Vehicle operation disruption
CVE-2022-20212	Tapjacking attack. An app can invoke operations reserved for privileged apps.	Application layer (apps)	Privilege escalation	Driver distraction, Vehicle operation disruption
CVE-2022-20144	a vulnerability in Android Automotive OS that allows a malicious app to bypass permission checks in the system settings or CarSettings component, enabling unauthorized access to privileged operations.	Application layer (apps)	Privilege escalation	Vehicle operation disruption

Despite the diversity of potential attack origins, the officially reported vulnerabilities are predominantly located at the application layer and are typically exploitable through third-party applications. While prior academic research has demonstrated the feasibility of system-level attacks, the vulnerability data suggests that apps remain the most common and practical attack vector in the Android ecosystem.

Evaluating the potential impact of these attacks is more challenging. The corresponding entries in the CVE database rarely include detailed descriptions of the exact exploit mechanisms or their real-world consequences. Instead, they typically report impacts in terms of the CIA triad, which does not clearly indicate the type of target damage in the automotive context. As a result, the available CVE information provides limited insight into the actual damage that could result from successful exploitation.

Our analysis indicates that the majority of CVEs affect driver and vehicle operation, leading to system instability, crashes, or unintentional configuration changes. Fewer vulnerabilities target privacy, such as unauthorized notification or Bluetooth access. This distribution confirms that our categorization not only aligns with observed attack patterns but also highlights areas of potential risk that warrant further attention, such as system resilience against DoS-type attacks and the protection of sensitive user data. Overall, the case study supports the practical applicability of our classification in assessing AAOS security threats.

7. Limitations of current approaches to AAOS security

Android Automotive OS (AAOS) reveals critical gaps in its permission model that warrant deeper exploration. Future research should prioritize identifying systemic weaknesses in how permissions are structured, validated, and enforced across AAOS layers, particularly focusing on low-level interactions between applications, APIs, and services.

While Android Automotive OS (AAOS) inherits many security features from Android, its integration with vehicle systems introduces

unique challenges that current approaches do not fully address. Key limitations include:

- (1) **Incomplete and Outdated Documentation.** AAOS's public documentation often lags behind actual implementations. For example, our analysis revealed that the number of defined vehicle properties and platform permissions in source code far exceeds what is documented. This mismatch leads to misconfigurations, overlooked security checks, and difficulty for developers and auditors trying to assess the system's real security posture.
- (2) **Limited Access Control Granularity.** AAOS relies heavily on Android's permission model, which lacks the fine-grained access control needed for complex automotive systems. Many sensitive APIs and vehicle properties are protected only by signature or privileged permissions, which can be bypassed by compromised system components or improperly signed vendor apps. Furthermore, the lack of context-aware access control (e.g., based on driving state or user profile) limits dynamic security enforcement.
- (3) **Overreliance on OEM Customizations.** OEMs frequently extend AAOS with proprietary services, vehicle properties, and permissions. However, these customizations are often undocumented and inconsistently implemented, creating an opaque and fragmented security landscape. Without clear guidelines or validation requirements, critical vulnerabilities may be introduced at the vendor layer, often undetected by Google or third-party auditors.
- (4) **Insufficient Protection Against Physical and Debug Access.** Although AAOS supports protections like SELinux and verified boot, many systems expose powerful debug interfaces (e.g., ADB over USB or Wi-Fi) in production builds. If not properly disabled or secured, these interfaces can allow attackers to gain root access, modify system files, or tamper with firmware bypassing all app-level restrictions.

- (5) **Weaknesses in Inter-Component Communication (ICC).** Android's ICC model introduces known risks such as intent spoofing and privilege escalation that remain unaddressed in Android Automotive. Despite security improvements like component export restrictions and runtime permissions, misconfigured components in AAOS can still be exploited. Given that many system apps interact directly with vehicle functions, such vulnerabilities present significant safety concerns.
- (6) **Lack of Defenses for Real-Time Automotive Threats.** Existing Android security mechanisms are designed for mobile environments, not for safety-critical automotive use cases. As AAOS-specific attacks continue to evolve, the lack of real-time anomaly detection and response mechanisms leaves the system vulnerable to threats that require immediate mitigation.
- (7) **Limited Transparency and Review of Hidden APIs.** AAOS contains hundreds of hidden APIs (e.g., @FlaggedApi) that are undocumented and not included in standard SDKs. These APIs are often accessible to system or privileged apps, creating a large, unreviewed attack surface. Without transparency or external review, vulnerabilities in these interfaces may go unnoticed.

8. Conclusion

In this survey, we presented a systematic, security-focused analysis of Android Automotive OS (AAOS), covering its architectural foundations, permission model, guarded APIs, and vehicle property framework. By examining five consecutive AAOS releases (versions 11 to 15), we showed how Android Automotive builds on standard Android mechanisms such as sandboxing, permissions, and system APIs, while integrating automotive-specific components like the Vehicle HAL, CarService, and OEM-defined extensions. Our quantitative analysis reveals a clear trend: the number and complexity of permissions, APIs, and vehicle properties have grown substantially over time. We introduced a threat categorization structured by attack origin, mechanism, and impact, which captures the unique vulnerability landscape of AAOS. By mapping existing attacks to this framework, we identified weaknesses and showed how even localized flaws can escalate into privacy violations or safety-critical failures. Our findings provide a foundation for future research. Addressing the identified gaps will require collaborative efforts among researchers, Google, and OEMs. This work offers a roadmap for improving the security posture of Android Automotive, helping ensure safer and more privacy-respecting deployments as the platform continues to evolve.

CRedit authorship contribution statement

Faiyaz Khan: Writing – original draft, Software, Methodology. **Hanbo Yu:** Writing – original draft, Methodology, Formal analysis. **Junjie Wu:** Writing – review & editing, Investigation. **Natalia Stakhanova:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Investigation, Funding acquisition, Conceptualization. **Benjamin C.M. Fung:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Natalia Stakhanova reports financial support was provided by The National Cybersecurity Consortium (NCC). If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research was supported by the National Cybersecurity Consortium (NCC) funding.

Data availability

No data was used for the research described in the article.

References

- Ahmad, U., Han, M., Jolfaei, A., Jabbar, S., Ibrar, M., Erbad, A., Herbert Song, H., Alkhrijah, Y., 2024. A comprehensive survey and tutorial on smart vehicles: Emerging technologies, security issues, and solutions using machine learning. *IEEE Trans. Intell. Transp. Syst.* 25 (11), 15314–15341.
- Aliwa, E., Rana, O., Perera, C., Burnap, P., 2021. Cyberattacks and countermeasures for in-vehicle networks. *ACM Comput. Surv.* 54 (1), [Online]. Available: <https://doi.org/10.1145/3431233>.
- Android, 2025a. Android developers reference: <permission>. <https://developer.android.com/guide/topics/manifest/permission-element#plevel>.
- Android, 2025b. The vehicle hardware abstraction layer (VHAL) overview. <https://source.android.com/docs/automotive/vhal?hl=en>.
- Android Open Source Project, 2024. Android OS documentation. [Online]. Available: <https://source.android.com/docs>. (Accessed 31 December 2025).
- Barzolevskaia, A., Branca, E., Stakhanova, N., 2024. Measuring and characterizing (mis)compliance of the Android permission system. *IEEE Trans. Softw. Eng.* (01), 1–23.
- Bozda, M., Samie, M., Jennions, I., 2018. A survey on can bus protocol: Attacks, challenges, and potential solutions. In: 2018 International Conference on Computing, Electronics & Communications Engineering. ICCECE, IEEE, pp. 201–205.
- Buscemi, A., Turcanu, I., Castagnani, G., Panchenko, A., Engel, T., Shin, K.G., 2023. A survey on controller area network reverse engineering. *IEEE Commun. Surv. & Tutorials* 25 (3), 1445–1481.
- Chatzoglou, E., Kambourakis, G., Kouliaridis, V., 2021. A multi-tier security analysis of official car management apps for android. *Future Internet* 13 (3).
- Cheng, B.H., Doherty, B., Polanco, N., Pasco, M., 2019. Security patterns for automotive systems. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion. MODELS-C, pp. 54–63.
- De Vincenzi, M., Costantino, G., Matteucci, I., Fenzl, F., Plappert, C., Rieke, R., Zelle, D., 2024. A systematic review on security attacks and countermeasures in automotive ethernet. 56 (6), [Online]. Available: <https://doi.org/10.1145/3637059>.
- Developers, A., 2025. Permissions on Android. <https://developer.android.com/guide/topics/permissions/overview#system-components>.
- Douss, A.B.C., Abassi, R., Sauveron, D., 2023. State-of-the-art survey of in-vehicle protocols and automotive ethernet security and vulnerabilities. *Math. Biosci. Eng.* 20 (9), 17057–17095, [Online]. Available: <https://www.aimspress.com/article/doi/10.3934/mbe.2023761>.
- El-Rewini, Z., Sadatsharan, K., Selvaraj, D.F., Plathottam, S.J., Ranganathan, P., 2020. Cybersecurity challenges in vehicular communications. *Veh. Commun.* 23, 100214, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S221420961930261X>.
- Elkhail, A.A., Refat, R.U.D., Habre, R., Hafeez, A., Bacha, A., Malik, H., 2021. Vehicle security: A survey of security issues and vulnerabilities, malware attacks and defenses. *IEEE Access* 9, 162401–162437.
- Eriksson, B., Groth, J., Sabelfeld, A., 2019. On the road with third-party apps: Security analysis of an in-vehicle app platform. In: VEHTS. pp. 64–75.
- Fernandez de Arroyabe, I., Watson, T., Angelopoulou, O., 2023. Cybersecurity in the automotive industry: A systematic literature review (SLR). *J. Comput. Inf. Syst.* 63 (3), 716–734, [Online]. Available: <https://doi.org/10.1080/08874417.2022.2103853>.
- Gong, H., Hong, S., Yang, S., Chang, R., Shen, W., Yuan, Z., Yu, C., Zhou, Y., 2025. Harness: transparent and lightweight protection of vehicle control on untrusted android automotive operating system. In: Proceedings of the 34th USENIX Conference on Security Symposium. USENIX Association, USA.
- Gözübüyük, B., Tang, B., Shin, K.G., Pesé, M.D., 2024. Analyzing privacy implications of data collection in android automotive OS. *arXiv preprint arXiv:2409.15561*.
- Grimm, D., Stang, M., Sax, E., 2021. Context-aware security for vehicles and fleets: A survey. *IEEE Access* 9, 101809–101846.
- Halder, S., Ghosal, A., Conti, M., 2020. Secure over-the-air software updates in connected vehicles: A survey. *Comput. Netw.* 178, 107343, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1389128619314963>.
- Hao, J., Han, G., 2020. On the modeling of automotive security: A survey of methods and perspectives. *Future Internet* 12 (11), [Online]. Available: <https://www.mdpi.com/1999-5903/12/11/198>.

- Huber, M., Brunner, M., Sauerwein, C., Carlan, C., Breu, R., 2018. Roadblocks on the highway to secure cars: An exploratory survey on the current safety and security practice of the automotive industry. In: Gallina, B., Skavhaug, A., Bitsch, F. (Eds.), *Computer Safety, Reliability, and Security*. Springer International Publishing, Cham, pp. 157–171.
- International Organization for Standardization (ISO), SAE International, 2021. ISO/SAE 21434:2021 Road Vehicles — Cybersecurity Engineering. Std. ISO/SAE 21434:2021, ISO and SAE International, International Standard. [Online]. Available: <https://www.iso.org/standard/70918.html>.
- Jadhav, S., Kshirsagar, D., 2018. A survey on security in automotive networks. In: 2018 Fourth International Conference on Computing Communication Control and Automation. ICCUBEA, pp. 1–6.
- Jo, H.J., Choi, W., 2022. A survey of attacks on controller area networks and corresponding countermeasures. *IEEE Trans. Intell. Transp. Syst.* 23 (7), 6123–6141.
- Jumana, Vyas, P., Aafer, Y., 2025. Red light for security: Uncovering auto feature check and access control gaps in AAOS. In: Egele, M., Moonsamy, V., Gruss, D., Carminati, M. (Eds.), *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Nature Switzerland, Cham, pp. 147–166.
- Kifor, C.V., Popescu, A., 2024. Automotive cybersecurity: A survey on frameworks, standards, and testing and monitoring technologies. *Sensors* 24 (18), [Online]. Available: <https://www.mdpi.com/1424-8220/24/18/6139>.
- Kim, S., Shrestha, R., 2020. *Automotive Cyber Security*. Vol. 34, Springer, Singapur.
- Lampe, B., Meng, W., 2023. A survey of deep learning-based intrusion detection in automotive applications. *Expert Syst. Appl.* 221, 119771, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0957417423002725>.
- Lauser, T., Zelle, D., Krauß, C., 2020. Security analysis of automotive protocols. In: *Proceedings of the 4th ACM Computer Science in Cars Symposium. CSCS '20*, Association for Computing Machinery, New York, NY, USA, [Online]. Available: <https://doi.org/10.1145/3385958.3430482>.
- Le, V.H., den Hartog, J., Zannone, N., 2018. Security and privacy for innovative automotive applications: A survey. *Comput. Commun.* 132, 17–41, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S014036641731174X>.
- Lee, Y., Woo, S., Lee, J., Song, Y., Moon, H., Lee, D.H., 2019. Enhanced Android app repackaging attack on in-vehicle network. *Wirel. Commun. Mob. Comput.* 2019 (1), 5650245.
- Li, X., Yu, Y., Sun, G., Chen, K., 2018. Connected vehicles' security from the perspective of the in-vehicle network. *IEEE Netw.* 32 (3), 58–63.
- Limbsiya, T., Teng, K.Z., Chattopadhyay, S., Zhou, J., 2022. A systematic survey of attack detection and prevention in connected and autonomous vehicles. *Veh. Commun.* 37, 100515, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2214209622000626>.
- Lu, Z., Qu, G., Liu, Z., 2019. A survey on recent advances in vehicular network security, trust, and privacy. *IEEE Trans. Intell. Transp. Syst.* 20 (2), 760–776.
- Luo, F., Jiang, Y., Zhang, Z., Ren, Y., Hou, S., 2021. Threat analysis and risk assessment for connected vehicles: A survey. *Secur. Commun. Networks* 2021 (1), 1263820, [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/1263820>.
- Luo, F., Zhang, X., Yang, Z., Jiang, Y., Wang, J., Wu, M., Feng, W., 2022. Cybersecurity testing for automotive domain: A survey. *Sensors* 22 (23), [Online]. Available: <https://www.mdpi.com/1424-8220/22/23/9211>.
- Mahmood, S., Nguyen, H.N., Shaikh, S.A., 2021. Automotive cybersecurity testing: Survey of testbeds and methods. In: Tagarev, T., Atanassov, K.T., Kharchenko, V., Kacprzyk, J. (Eds.), *Digital Transformation, Cyber Security and Resilience of Modern Societies*. Springer International Publishing, Cham, pp. 219–243, [Online]. Available: https://doi.org/10.1007/978-3-030-65722-2_14.
- Malhi, A.K., Batra, S., Pannu, H.S., 2020. Security of vehicular ad-hoc networks: A comprehensive survey. *Comput. Secur.* 89, 101664, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167404818312872>.
- Mandal, A.K., Cortesi, A., Ferrara, P., Panarotto, F., Spoto, F., 2018. Vulnerability analysis of Android auto infotainment apps. In: *Proceedings of the 15th ACM International Conference on Computing Frontiers. CF '18*, Association for Computing Machinery, New York, NY, USA, pp. 183–190.
- Mandal, A.K., Panarotto, F., Cortesi, A., Ferrara, P., Spoto, F., 2019. Static analysis of Android Auto infotainment and on-board diagnostics II apps. *Softw.: Pract. Exp.* 49 (7), 1131–1161.
- Mawonde, K., Isong, B., Lugayizi, F., Abu-Mahfouz, A.M., 2018. A survey on vehicle security systems: Approaches and technologies. In: *Proceedings of Th 44th Annual Conference of the IEEE Industrial Electronics Society. IECON*, pp. 4633–4638.
- Moiz, A., Alalfi, M.H., 2020. An approach for the identification of information leakage in automotive infotainment systems. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 110–114.
- Moiz, A., Alalfi, M.H., 2022. A survey of security vulnerabilities in Android automotive apps. In: *Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems. EnCyCris '22*, Association for Computing Machinery, New York, NY, USA, pp. 17–24, [Online]. Available: <https://doi.org/10.1145/3524489.3527300>.
- Oh, I., Sahlabadi, M., Yim, K., Lee, S., 2025. Threat classification and vulnerability analysis on 5G firmware over-the-air updates for mobile and automotive platforms. *Electronics* 14 (10).
- Pesé, M., Shin, K., Bruner, J., Chu, A., 2020. Security analysis of Android Automotive. *SAE Int. J. Adv. Curr. Pract. Mobil.* 2.
- Rajapaksha, S., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Madzudzo, G., Cheah, M., 2023. AI-based intrusion detection systems for in-vehicle networks: A survey. *ACM Comput. Surv.* 55 (11).
- Rathore, R.S., Hewage, C., Kaiwartya, O., Lloret, J., 2022. In-vehicle communication cyber security: Challenges and solutions. *Sensors* 22 (17), [Online]. Available: <https://www.mdpi.com/1424-8220/22/17/6679>.
- Renganathan, V., Yurtsever, E., Ahmed, Q., Yener, A., 2022. Valet attack on privacy: a cybersecurity threat in automotive Bluetooth infotainment systems. *Cybersecurity* 5 (1), [Online]. Available: <http://dx.doi.org/10.1186/s42400-022-00132-x>.
- Sharma, C., Moylan, S., Amariucal, G.T., Vasserman, E.Y., 2019. An extended survey on vehicle security. *ArXiv abs/1910.04150*, [Online]. Available: <https://api.semanticscholar.org/CorpusID:203951961>.
- Silva, G., Ridley, N., Branca, E., Stakhanova, N., 2024. A Systematic Evaluation of Non-SDK Interface Restrictions in Android: Bridging the Gap Between Guidelines and Practice. In: *Proceedings of the 17th International Symposium on Foundations & Practice of Security. FPS2024*, Association for Computing Machinery, New York, NY, USA.
- Sommer, F., Dürrwang, J., Kriesten, R., 2019. Survey and classification of automotive security attacks. *Information* 10 (4), [Online]. Available: <https://www.mdpi.com/2078-2489/10/4/148>.
- Sun, X., Yu, F.R., Zhang, P., 2022. A survey on cyber-security of connected and autonomous vehicles (CAVs). *IEEE Trans. Intell. Transp. Syst.* 23 (7), 6240–6259.
- Takahashi, J., Iwamura, M., Tanaka, M., 2020. Security threat analysis of automotive infotainment systems. In: *Proceedings of the IEEE 92nd Vehicular Technology Conference. VTC2020-Fall*, pp. 1–7.
- Talpur, A., Gurusamy, M., 2022. Machine learning for security in vehicular networks: A comprehensive survey. *IEEE Commun. Surv. & Tutorials* 24 (1), 346–379.
- Tran, P., Nguyen, D.C., 2022. Advanced lightweight cryptography for automotive security: Surveys, challenges and solutions. In: *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security - Volume 1: IoTBDS. INSTICC. SciTePress*, pp. 304–311.
- United Nations Economic Commission for Europe (UNECE), 2021. UN Regulation No. 155: Uniform Provisions Concerning the Approval of Vehicles with Regards to Cyber Security and Cyber Security Management System. Tech. Rep., UNECE World Forum for Harmonization of Vehicle Regulations (WP.29), UN Regulation No. 155 (Cybersecurity and Cybersecurity Management System). [Online]. Available: <https://unece.org/sites/default/files/2023-02/R155e%20%282%29.pdf>.
- van der Sanden, B., Vasenev, A., 2020. Architectural guidance in automotive for privacy and security: Survey and classification. In: 2020 IEEE International Systems Conference. SysCon, pp. 1–8.
- Weiss, N., Schreieck, M., Wiesche, M., Krcmar, H., 2021. From product to platform: How can BMW compete with platform giants? *J. Inf. Technol. Teach. Cases* 11 (2), 90–100, [Online]. Available: <https://journals.sagepub.com/doi/abs/10.1177/2043886920944185>.
- Wu, W., Li, R., Xie, G., An, J., Bai, Y., Zhou, J., Li, K., 2020. A survey of intrusion detection for in-vehicle networks. *IEEE Trans. Intell. Transp. Syst.* 21 (3), 919–933.
- Young, C., Zambreno, J., Olufowobi, H., Bloom, G., 2019. Survey of Automotive Controller Area network intrusion detection systems. *IEEE Des. & Test* 36 (6), 48–55.
- Zheng, Y., Zhang, M., Li, X., Chen, X., Zhang, Z., Zhu, J., Shan, C., Wu, G., 2025. Automotive security in the digital era: A comprehensive survey of attacks and defenses for keyless entry system. In: Zhou, K. (Ed.), *Computational and Experimental Simulations in Engineering*. Springer Nature Switzerland, Cham, pp. 444–467.